aws

開発者ガイド

AWS Lambda

AWS Lambda: 開発者ガイド

Table of Contents

AWS Lambda とは

AWS Lambda を使用すると、サーバーをプロビジョニングまたは管理することなくコードを実行できます。

Lambda は可用性の高いコンピューティングインフラストラクチャでコードを実行し、コンピューティングリソースに関するすべての管理を行います。これには、サーバーおよびオペレーティングシステムのメンテナンス、容量のプロビジョニングおよび自動スケーリング、さらにログ記録などが含まれます。Lambda で必要なことは、サポートするいずれかの言語ランタイムにコードを与えることだけです。

コードを Lambda 関数に整理します。Lambda サービスは、必要な場合にのみ関数を実行し、自動的にスケーリングします。消費したコンピュート時間に対してのみ課金されます。コードが実行されていない間は料金は発生しません。詳細については、AWS Lambda 料金を参照してください。

> **Tip**
>
> サーバーレスソリューションを構築する方法については、「サーバーレスデベロッパーガイド」を参照してください。

Lambda を使用するタイミング

Lambda は、迅速にスケールアップが求められ、要求がないときはゼロにスケールダウンする必要があるアプリケーションシナリオに最適なコンピュートサービスです。例えば、Lambda は以下を実行するために使用できます。

- ファイル処理: Amazon Simple Storage Service (Amazon S3) を使用して、アップロード後に Lambda データ処理をリアルタイムでトリガーします。
- ストリーム処理: Lambda と Amazon Kinesis を使用して、アプリケーションアクティビティの追跡、取引注文の処理、クリックストリーム分析、データクレンジング、ログのフィルタリング、インデックス作成、ソーシャルメディア分析、モノのインターネット (IoT) デバイスデータのテレメトリ、および計測のためにリアルタイムのストリーミングデータを処理します。
- ウェブアプリケーション: Lambda と他の AWS サービスを組み合わせて、自動的にスケールアップおよびスケールダウンし、複数のデータセンターにまたがる高可用性設定で実行される強力なウェブアプリケーションを構築します。

- IoT バックエンド: Lambda を使用してサーバーレスバックエンドを構築し、ウェブ、モバイル、IoT、およびサードパーティの API リクエストを処理します。
- モバイルバックエンド: Lambda と Amazon API Gateway を使用してバックエンドを構築し、API リクエストを認証して処理します。AWS Amplify を使用すると、iOS、Android、ウェブ、React Native フロントエンドと簡単に統合できます。

Lambda を使用する際、ユーザーが責任を負うのはコードのみです。Lambda によって、コードを実行するメモリのバランス、CPU、ネットワーク、その他のリソースを提供するコンピューティングフリートが管理されます。Lambda がこれらのリソースを管理するため、コンピューティングインスタンスにログインしたり、提供されたランタイムのオペレーティングシステムをカスタマイズしたりすることはできません。Lambda は、容量の管理、モニタリング、Lambda 関数のログ記録など、運用および管理アクティビティをユーザーに代わって実行します。

主な特徴

次の主要な機能は、スケーラブルで安全で拡張が容易な Lambda アプリケーションの開発に役立ちます。

環境変数

: 環境変数を使用し、コードを更新することなく関数の動作を調整します。

バージョン

: たとえば、安定した本番環境バージョンのユーザーに影響を与えることなく、ベータテスト用に新しい関数を使用するために、関数のデプロイをバージョンで管理できます。

コンテナイメージ

: 既存のコンテナツールを再使用したり、かなり大量の依存関係 (機械学習など) を持つ大規模なワークロードをデプロイしたりすることができるように、AWS が提供するベースイメージまたは代替のベースイメージを使用して、Lambda 関数のコンテナイメージを作成します。

レイヤー

: ライブラリおよびその他の依存関係をパッケージ化し、デプロイアーカイブのサイズを削減し、コードをデプロイするスピードを速めます。

Lambda 拡張機能

: モニタリング、視認性、セキュリティ、ガバナンスに使用するツールで Lambda 関数を強化します。

関数 URL

Lambda 関数に専用の HTTP(S) エンドポイントを追加します。

レスポンスストリーミング

Lambda 関数 URL を設定し、レスポンスペイロードを Node.js 関数からクライアントにストリーミングで返したり、最初のバイトまでの時間 (TTFB) のパフォーマンスを向上させたり、より大きなペイロードを返したりします。

同時実行とスケーリングの制御

本番環境のアプリケーションのスケーリングおよび応答性を細かく制御します。

コード署名

承認されたデベロッパーのみが Lambda 関数で未変更で信頼できるコードを公開していることを検証します。

プライベートネットワーク設定

データベース、キャッシュインスタンス、内部サービスなどのリソースのプライベートネットワークを作成します。

ファイルシステムへのアクセス

Amazon Elastic File System (Amazon EFS) をローカルディレクトリにマウントするように関数を設定し、安全かつ高い同時実行レベルで関数コードが共有リソースをアクセスして変更できるようにします。

Lambda SnapStart for Java

Java ランタイムの起動パフォーマンスを追加コストなしで最大 10 倍向上させることができ、通常は関数コードを一切変更しません。

Lambda の開始方法

Lambda の使用を開始するには、Lambda コンソールを使用して関数を作成します。数分で関数を作成およびデプロイして、それをコンソールでテストできます。

チュートリアルを進めていくうちに、Lambda のイベントオブジェクトを使用して関数に引数を渡す方法など、Lambda の基本的な概念を学んでいきます。また、関数から ログ出力を返す方法と、CloudWatch Logs で関数の呼び出しログを表示する方法についても学習します。

簡単のために、関数の作成には Python または Node.js ランタイムのいずれかを使用します。これらはインタープリター言語なので、コンソールの組み込みコードエディタで関数のコードを直接編集できます。Java や C# などのコンパイル型言語では、ローカルのビルドマシン上でデプロイパッケージを作成し、それを Lambda にアップロードする必要があります。他のランタイムを使用して Lambda に関数をデプロイする方法については、「the section called “その他のリソースと次のステップ”」セクションにあるリンクを参照してください。

> **Tip**
>
> サーバーレスソリューションを構築する方法については、「サーバーレスデベロッパーガイド」を参照してください。

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があり

ます。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] を選んで、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理できます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウントのメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

コンソールで Lambda の関数の作成

この例の関数は、`"length"` および `"width"` のラベルが付けられた 2 つの整数値を含む JSON オブジェクトを取り込みます。関数はこれらの値を乗算して面積を計算し、これを JSON 文字列として返します。

またこの関数は、計算された面積に加え CloudWatch ロググループの名前の表示も行います。チュートリアルの後半では、関数の呼び出しの記録を表示するための、CloudWatch Logs の使用方法を学習します。

自分の関数を作成するには、まずコンソールを使用して基本の Hello world 関数を作成します。その後のステップで、独自の関数コードを追加します。

コンソールでLambda 関数 Hello world を作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. [Create function (関数の作成)] を選択します。

3. [ゼロから作る] を選択します。
4. [基本情報] ペインで、[関数名] に「**myLambdaFunction**」を入力します。
5. [ランタイム] では、[Node.js 20.x] または [Python 3.12] のいずれかを選択します。
6. [アーキテクチャ] の設定は [86_64] のままにし、[関数を作成] を選択します。

Lambda はメッセージ `Hello from Lambda!` を返す関数を作成します。また Lambda は、ユーザーの関数のための実行ロールも作成します。実行ロールとは、AWS のサービス およびリソースに対するアクセス許可を Lambda 関数に付与する AWS Identity and Access Management (IAM) のロールです。Lambda が作成するこのロールは、CloudWatch Logs に書き込むための基本的なアクセス許可を、ユーザーの関数に付与します。

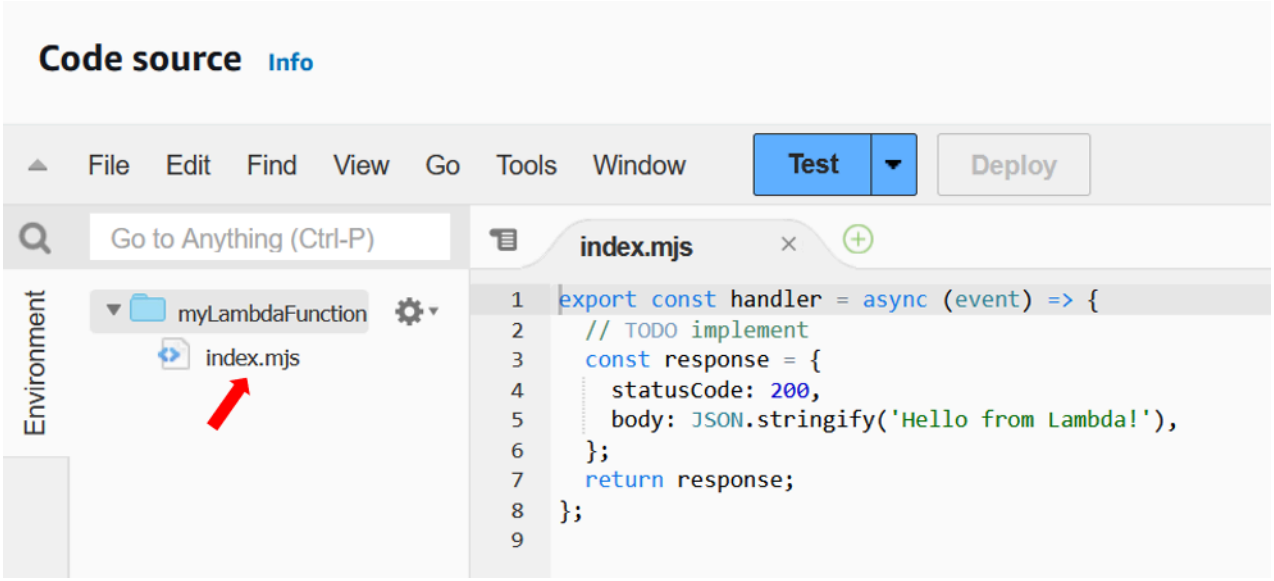
これで、コンソールの組み込みコードエディターを使用して、Lambda が作成した Hello world コードを独自の関数コードに置き換えることができるようになります。

Node.js

コンソールでコードを変更するには

1. [コード] タブを選択します。

   Lambda が関数コードを作成すると、それがコンソールの組み込みコードエディタに表示されます。コードエディターに index.mjs タブが表示されない場合は、次の図に示すように、ファイルエクスプローラーで index.mjs を選択します。



2. 次のコードを index.mjs タブに貼り付け、Lambda が作成したコードを置き換えます。

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. [デプロイ] を選択して、関数のコードを更新します。変更内容が Lambda によりデプロイされると、関数が正常に更新されたことをユーザーに知らせるバナーが、コンソールに表示されます。

関数コードを把握する

次のステップに進む前に、関数コードを読む時間を作り、Lambda の主要な概念を把握しましょう。

- Lambda ハンドラー:

  ユーザーの Lambda 関数は、`handler` という名前の Node.js 関数を含んでいます。Node.js の Lambda 関数には複数の Node.js 関数を含めることができますが、コードへのエントリポイントは、常に handler 関数です。関数が呼び出されると、Lambda はこのメソッドを実行します。

  コンソールを使用して Hello world 関数を作成すると、Lambda は自動的に、関数のハンドラーメソッドの名前を `handler` に設定します。この Node.js 関数の名前は編集しないでください。編集すると、関数を呼び出しても Lambda はコードを実行できなくなります。

Node.js の Lambda ハンドラーの詳細については、「the section called “Handler”」を参照してください。

- Lambda のイベントオブジェクト:

  関数 `handler` は 2 つの引数 (`event` および `context`) を受け取ります。Lambda のイベントとは JSON 形式のドキュメントであり、関数で処理するためのデータが含まれています。

  関数が別の AWS のサービス によって呼び出された場合、イベントオブジェクトには、その呼び出しの原因となったイベントに関する情報が含まれています。例えば、オブジェクトのアップロード時に Amazon Simple Storage Service (Amazon S3) バケットが関数を呼び出した場合、イベントは Amazon S3 バケットの名前とオブジェクトキーを含んでいます。

  この例では、キーと値のペア 2 つを含む JSON 形式のドキュメントをコンソールに入力することで、イベントを作成しています。

- Lambda のコンテキストオブジェクト:

  この関数が受け取る 2 番目の引数は`context`。Lambda は、ユーザーの関数に対し、自動的にコンテキストオブジェクトを渡します。コンテキストオブジェクトは、関数の呼び出しおよび実行環境に関する情報を含んでいます。

  モニタリング目的として、コンテキストオブジェクトを使用して関数の呼び出しに関する情報を出力できます。この例では、関数が `logGroupName` パラメータを使用して、CloudWatch ロググループの名前を出力しています。

  Node.js の Lambda コンテキストオブジェクトの詳細については、「the section called “Context”」を参照してください。

- Lambda でのログ記録:

  Node.js では、関数のログに情報を送信するために、`console.log` や `console.error` などのコンソールメソッドを使用できます。このサンプルコードでは、計算された面積と関数の CloudWatch Logs グループの名前を出力するために、`console.log` ステートメントを使用します。また、`stdout` または `stderr` に書き込みを行う任意のロギングライブラリも使用できます。

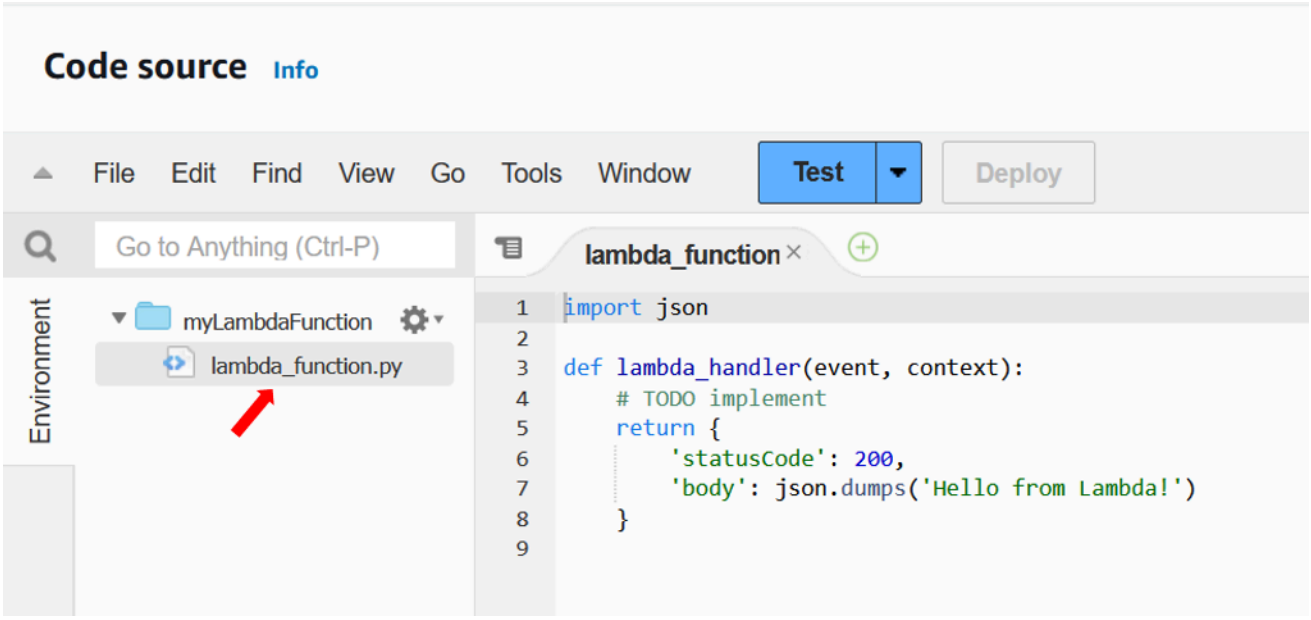
  詳細については、「the section called “ログ記録”」を参照してください。他のランタイムでのログ記録については、関連するランタイムの「構築方法」のページを参照してください。

Python

コンソールでコードを変更するには

1. [コード] タブを選択します。

   Lambda が関数コードを作成すると、それがコンソールの組み込みコードエディタに表示されます。コードエディターに lambda_function.py タブが表示されない場合は、次の図に示すように、ファイルエクスプローラーで lambda_function.py 選択します。



2. 次のコードを lambda_function.py タブに貼り付け、Lambda が作成したコードを置き換えます。

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']

    area = calculate_area(length, width)
```

```
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. [デプロイ] を選択して、関数のコードを更新します。変更内容が Lambda によりデプロイされると、関数が正常に更新されたことをユーザーに知らせるバナーが、コンソールに表示されます。

関数コードを把握する

次のステップに進む前に、関数コードを読む時間を作り、Lambda の主要な概念を把握しましょう。

- Lambda ハンドラー:

  ユーザーの Lambda 関数には、`lambda_handler` という名前の Python 関数が含まれています。Python の Lambda 関数には複数の Python 関数を含めることができますが、コードへのエントリポイントは、常に handler 関数です。関数が呼び出されると、Lambda はこのメソッドを実行します。

  コンソールを使用して Hello world 関数を作成すると、Lambda は自動的に、関数のハンドラーメソッドの名前を `lambda_handler` に設定します。この Python 関数の名前は編集しないでください。編集すると、関数を呼び出しても Lambda はコードを実行できなくなります。

  Python の Lambda ハンドラーの詳細については、「the section called “Handler”」を参照してください。

- Lambda のイベントオブジェクト:

  関数 `lambda_handler` は 2 つの引数 (`event` および `context`) を受け取ります。Lambda のイベントとは JSON 形式のドキュメントであり、関数で処理するためのデータが含まれています。

関数が別の AWS のサービス によって呼び出された場合、イベントオブジェクトには、その呼び出しの原因となったイベントに関する情報が含まれています。例えば、オブジェクトのアップロード時に Amazon Simple Storage Service (Amazon S3) バケットが関数を呼び出した場合、イベントは Amazon S3 バケットの名前とオブジェクトキーを含んでいます。

この例では、キーと値のペア 2 つを含む JSON 形式のドキュメントをコンソールに入力することで、イベントを作成しています。

- Lambda のコンテキストオブジェクト:

  この関数が受け取る 2 番目の引数は`context`。Lambda は、ユーザーの関数に対し、自動的にコンテキストオブジェクトを渡します。コンテキストオブジェクトは、関数の呼び出しおよび実行環境に関する情報を含んでいます。

  モニタリング目的として、コンテキストオブジェクトを使用して関数の呼び出しに関する情報を出力できます。この例では、関数が `log_group_name` パラメータを使用して、CloudWatch ロググループの名前を出力しています。

  Python の Lambda コンテキストオブジェクトの詳細については、「the section called “Context”」を参照してください。

- Lambda でのログ記録:

  Python では、関数のログに情報を送信するために、`print` ステートメントまたは Python ログ記録ライブラリのいずれかを使用できます。キャプチャされる内容の違いを説明するために、このサンプルコードでは両方の方法を使用しています。本番用のアプリケーションでは、ログ記録ライブラリを使用することをお勧めします。

  詳細については、「the section called “ログ記録”」を参照してください。他のランタイムでのログ記録については、関連するランタイムの「構築方法」のページを参照してください。

コンソールを使用して Lambda 関数を呼び出す

Lambda コンソールを使用して自分の関数を呼び出すには、まずテストイベントを作成して、それを関数に送信します。このイベントは、キー `"length"` および `"width"` を使用するキーと値のペア 2 つを含む、JSON 形式のドキュメントです。

テストイベントを作成するには

1. [コードソース] ペインで、[テスト] を選択します。

2. [新しいイベントを作成] を選択します。
3. [イベント名] に、「**myTestEvent**」を入力します。
4. [イベント JSON] パネルに、以下を貼り付けてデフォルト値を置き換えます。

```
{
  "length": 6,
  "width": 7
}
```

5. [Save] を選択します。

これで、関数をテストし、Lambda コンソールと CloudWatch Logs を使用して関数呼び出しのレコードを表示できるようになります。

関数をテストして呼び出しのレコードをコンソールに表示するには

- [コードソース] ペインで、[テスト] を選択します。関数の実行が終了すると、[実行結果] タブに応答と関数ログが表示されます。次のような結果が表示されます。

  Node.js

```
Test Event Name
myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2023-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is
 42
2023-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch
 log group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration:
 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

Python

```
Test Event Name
myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2023-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74
 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

この例では、コードの呼び出しにコンソールのテスト機能を使用しました。つまり、関数の実行結果は、コンソールに直接表示できます。関数呼び出しがコンソールの外部で行われる場合は、CloudWatch Logs を使用する必要があります。

CloudWatch Logs で関数の呼び出しレコードを表示するには

1. Amazon CloudWatch コンソールの [[Log groups (ロググループ)] ページ] を開きます。
2. 関数のロググループの名前を選択します (`/aws/lambda/myLambdaFunction`) 。これは関数がコンソールに出力したロググループ名です。
3. [ログストリーム] タブで、関数呼び出しに対応するログストリームを選択します。

   次のような出力が表示されます:

   Node.js

```
INIT_START Runtime Version: nodejs:20.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
```

```
2023-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2023-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.12.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2023-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

クリーンアップ

作業が完了したサンプル関数は削除しておきます。また、関数のログを保存するロググループと、コンソールが作成した実行ロールも削除できます。

Lambda 関数を削除するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [Actions] (アクション)、[Delete] (削除) の順に選択します。
4. [Delete function] (関数の削除) ダイアログボックスに「削除」と入力してから、[Delete] (削除) を選択します。

ロググループを削除するには

1. Amazon CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。

2. 関数のロググループ (`/aws/lambda/my-function`) を選択します。
3. [アクション]、[ロググループの削除] の順にクリックします。
4. ロググループの削除ダイアログボックスで、[削除] をクリックします。

実行ロールを削除するには

1. AWS Identity and Access Management (IAM) コンソールの [Roles (ロール)] ページを開きます。
2. 関数の実行ロールを選択します (`myLambdaFunction-role-31exxmpl` など)。
3. [削除] を選択します。
4. [Delete role] (ロールの削除) ダイアログボックスにロール名を入力し、[Delete] (削除) を選択します。

AWS CloudFormation と AWS Command Line Interface (AWS CLI) を使用して、関数、ロググループ、ロールの作成とクリーンアップを自動化できます。

その他のリソースと次のステップ

ここまでで、コンソールを使用し簡単な Lambda 関数を作成してテストを行ったので、次の各ステップを実行します。

- 自分のコードに依存関係を追加し、.zip デプロイパッケージを使用してデプロイする方法を学習します。以下のリンクから、関心のある言語を選択してください。

  Node.js

  「the section called “.zip ファイルアーカイブをデプロイする”」を参照してください。

  Typescript

  「the section called “.zip ファイルアーカイブをデプロイする”」を参照してください。

  Python

  「the section called “.zip ファイルアーカイブをデプロイする”」を参照してください。

  Ruby

  「the section called “.zip ファイルアーカイブをデプロイする”」を参照してください。

Java

「the section called ".zip ファイルアーカイブをデプロイする"」を参照してください。

Go

「the section called ".zip ファイルアーカイブをデプロイする"」を参照してください。

C#

「the section called "デプロイパッケージ"」を参照してください。

- 別の AWS のサービス から Lambda 関数を呼び出すための設定方法を学習するには、「チュートリアル: Amazon S3 トリガーを使用して Lambda 関数を呼び出す」を実行します。
- これより複雑な、他の AWS のサービス で Lambda を使用する例については、次のチュートリアルのいずれかを選択してください。
  - Amazon API Gateway で AWS Lambda を使用する: Lambda 関数を呼び出す Amazon API Gateway REST API を作成します。
  - チュートリアル: Lambda 関数を使用して Amazon RDS にアクセスする: Lambda 関数を使用して RDS Proxy 経由で Amazon Relational Database Service (Amazon RDS) データベースにデータを書き込みます。
  - チュートリアル: Amazon S3 トリガーを使用してサムネイル画像を作成する: Lambda 関数を使用して、イメージファイルが Amazon S3 バケットにアップロードされるたびにサムネイルを作成します。

AWS Lambda の基礎

Lambda 関数は Lambda サービスの主要リソースです。

関数を設定するには、Lambda コンソール、Lambda API、AWS CloudFormation または AWS SAM を使用します。関数のコードを作成し、デプロイパッケージを使用してコードをアップロードします。イベントが発生すると、Lambda が関数を呼び出します。Lambda は、関数の複数のインスタンスを並列に実行します。それらのインスタンスは、同時実行数とスケーリング制限によって管理されています。

トピック

Lambda の概念

Lambda が関数のインスタンスを実行してイベントを処理します。Lambda API を使用して関数を直接呼び出すことができます。または、AWS のサービスあるいはリソースを設定して関数を呼び出すことができます。

概念

- 機能
- Trigger トリガー)
- イベント
- 実行環境
- 命令セットアーキテクチャ
- デプロイパッケージ
- ランタイム
- Layer
- 拡張機能
- 同時実行
- Qualifier
- デスティネーション

機能

関数とは、Lambda でコードを実行するために呼び出すことができるリソースです。関数には、他の処理から渡された、もしくは他の AWS のサービスから送信されたイベントを、処理するためのコードが記述されています。

Trigger トリガー)

トリガーは、Lambda 関数を呼び出すリソースまたは設定です。トリガーの例としては、関数を呼び出すように設定できる AWS のサービスやイベントソースマッピングなどがあります。イベントソースマッピングは、ストリームまたはキューからアイテムを読み取り、関数を呼び出す Lambda のリソースです。詳細については、「Lambda 関数の呼び出しメソッドについて」および「他の AWS サービスからのイベントを使用した Lambda の呼び出し」を参照してください。

イベント

イベントは、処理する Lambda 関数のデータを含む JSON 形式のドキュメントです。イベントは、ランタイムによりオブジェクトに変換された上で、関数のコードに渡されます。関数を呼び出すときは、イベントの構造とコンテンツを決定します。

Example カスタムイベント - 気象データ

```
{
  "TemperatureK": 281,
  "WindKmh": -3,
  "HumidityPct": 0.55,
  "PressureHPa": 1020
}
```

AWS のサービスで関数を呼び出す場合、そのイベントのシェイプはサービスによって定義されます。

Example サービスイベント — Amazon SNS 通知

```
{
  "Records": [
    {
      "Sns": {
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        ...
```

AWS サービスからのイベントの詳細については、「他の AWS サービスからのイベントを使用した Lambda の呼び出し」を参照してください。

実行環境

実行環境は、Lambda 関数のための、安全で分離されたランタイム環境を提供します。関数の実行に必要なプロセスとリソースが、実行環境により管理されます。実行環境からは、関数と、その関数に関連付けられた任意の拡張機能のために、ライフサイクルのサポートが提供されます。

詳細については、「Lambda 実行環境」を参照してください。

命令セットアーキテクチャ

命令セットアーキテクチャは、Lambda が関数の実行に使用するコンピュータプロセッサタイプを決定します。Lambda は、命令セットアーキテクチャの選択肢を提供します。

- arm64 — AWS Graviton2 プロセッサ用の 64 ビット ARM アーキテクチャです。
- x86_64 — x86 ベースプロセッサ用の 64 ビット x86 アーキテクチャです。

詳細については、「Lambda 関数の命令セットアーキテクチャの設定」を参照してください。

デプロイパッケージ

Lambda 関数コードをデプロイするには、デプロイパッケージを使用します。Lambda では、次の 2 種類のデプロイパッケージがサポートされます。

- 関数コードとその依存関係を含む .zip ファイルアーカイブ。Lambda は、関数のためのオペレーティングシステムとランタイムを提供します。
- Open Container Initiative (OCI) の仕様に準拠したコンテナーイメージ。関数のコードと依存関係をイメージに追加します。また、オペレーティングシステムと Lambda ランタイムを含める必要があります。

詳細については、「Lambda デプロイパッケージ」を参照してください。

ランタイム

ランタイムでは、実行環境で実行される言語固有の環境が提供されます。ランタイムは、呼び出しイベント、コンテキスト情報、およびレスポンスを Lambda と関数の間で中継します。Lambda が提供するランタイムを使用することも、独自に構築することもできます。コードを .zip ファイルアーカイブとしてパッケージ化する場合は、プログラミング言語に適合したランタイムを使用するように、その関数を設定する必要があります。コンテナイメージの場合は、イメージをビルドするときにランタイムをインクルードします。

詳細については、「Lambda ランタイム」を参照してください。

Layer

Lambda レイヤーは、追加のコードまたはデータを含むことができる .zip ファイルアーカイブです。レイヤーには、ライブラリ、カスタムランタイム、データ、または設定ファイルを含めることができます。

レイヤーにより、Lambda 関数で使用するライブラリとその他の依存関係をパッケージ化する便利な方法が利用できます。レイヤーを使用することで、アップロードされたデプロイメントアーカイブのサイズを削減し、コードをデプロイするスピードを速めることができます。レイヤーを使用すると、コードの共有と責任の分離を促進し、ビジネスロジックの記述をより迅速に繰り返すことができます。

各関数につき最大 5 つのレイヤーを含めることができます。レイヤーは、標準の Lambda デプロイサイズクォータに対してカウントされます。関数にレイヤーを含むと、実行環境においてコンテンツが `/opt` ディレクトリに抽出されます。

デフォルトでは、作成したレイヤーは AWS アカウントに対してプライベートになります。レイヤーを他のアカウントと共有するか、またはパブリックにするか選ぶことができます。別のアカウントによって公開されたレイヤーを関数が消費する場合、関数は削除後にレイヤーバージョンを引き続き使用することができます。または、レイヤーへのアクセス権限が呼び出されます。しかし、削除されたレイヤーバージョンを使用して新しい関数を作成したり、関数を更新することはできません。

コンテナイメージとしてデプロイされた関数はレイヤーを使用しません。代わりに、コンテナイメージをビルドする際、必要なランタイム、ライブラリ、およびその他の依存関係を、そのイメージ内にパッケージ化します。

詳細については、「Lambda レイヤー」を参照してください。

拡張機能

Lambda 拡張機能を使用すると、関数を拡張できます。例えば、拡張機能を使用して、任意のモニタリングツール、オブザーバビリティツール、セキュリティツール、およびガバナンスツールに関数を統合できます。AWS Lambda パートナーが提供する幅広いツールセットから選択することも、独自の Lambda 拡張機能を作成することもできます。

内部拡張機能はランタイムプロセスで実行され、ランタイムと同じライフサイクルを共有します。外部拡張機能は、実行環境で別のプロセスとして実行されます。外部拡張機能は、関数が呼び出される前に初期化されます。また、関数のランタイムと並行して実行され、関数の呼び出しが完了した後も引き続き実行されます。

詳細については、「Lambda 拡張機能を使用して Lambda 関数を補強する」を参照してください。

同時実行

同時実行数とは、ある時点で関数が処理しているリクエストの数を指します。関数が呼び出されると、Lambda はその関数のインスタンスをプロビジョニングしてイベントを処理します。関数コードの実行が完了すると、別のリクエストを処理できます。リクエストの処理中に関数が再度呼び出されると、別のインスタンスがプロビジョンされるため、関数の同時実行数が増加します。

同時実行数は、AWS リージョンレベルのクォータの対象となります。個々の関数を設定して、同時実行数を制限したり、特定の同時実行数を達成できるようにしたりすることもできます。詳細については、「関数に対する予約済み同時実行数の設定」を参照してください。

Qualifier

関数を呼び出したり表示したりするときに、バージョンまたはエイリアスを指定するための修飾子を含めることができます。バージョンは、数値修飾子を持つ関数のコードと設定の変更不可能なスナップショットです。たとえば、`my-function:1` と指定します。エイリアスは、バージョンを指すポインタです。このポインタでは、別のバージョンにマップしたり、2 つのバージョン間でトラフィックを分割したりするための更新が可能です。たとえば、`my-function:BLUE` と指定します。バージョンとエイリアスを一緒に使用して、クライアントが関数を呼び出すための安定したインターフェイスを提供することができます。

詳細については、「Lambda 関数のバージョン」を参照してください。

デスティネーション

送信先は、Lambda が非同期呼び出しからイベントを送信できる AWS リソースです。処理に失敗したイベントの送信先を設定できます。一部のサービスでは、正常に処理されたイベントの宛先もサポートします。

詳細については、「非同期呼び出しの送信先の設定」を参照してください。

Lambda プログラミングモデル

Lambda では、すべてのランタイムに共通のプログラミングモデルを提供しています。プログラミングモデルとは、コードと Lambda システムとの間のインターフェイスを定義するものです。関数設定のハンドラを定義して、関数へのエントリポイントを Lambda に伝えます。ランタイムは、呼び出しイベント、および関数名やリクエスト ID などのコンテキストを含むオブジェクトをハンドラーに渡します。

ハンドラーが最初のイベントの処理を終了すると、ランタイムは別のイベントを送信します。関数のクラスはメモリ内にとどまるため、初期化コードにおいて、ハンドラーメソッドの外部で宣言されたクライアントおよび変数は再利用が可能です。後続のイベントの処理時間を短縮するには、初期化中に AWS SDK クライアントなどの再利用可能なリソースを作成します。初期化されると、関数の各インスタンスは数千件のリクエストを処理できます。

また、関数は、`/tmp` ディレクトリ内のローカルストレージにもアクセスできます。ディレクトリのコンテンツは、実行環境が停止された際に維持され、複数の呼び出しに使用できる一時的なキャッシュを提供します。詳細については、「Lambda 実行環境」を参照してください。

AWS X-Ray トレースが有効な場合、ランタイムは初期化と実行のために、別個のサブセグメントを記録します。

ランタイムは、関数からのログ出力をキャプチャし、Amazon CloudWatch Logs に送信します。ランタイムは、関数の出力をログに記録するだけでなく、関数の呼び出しの開始時と終了時にエントリも記録します。これには、リクエスト ID、請求期間、初期化期間、およびその他の詳細を含むレポートログが含まれます。関数によりエラーがスローされた場合、そのエラーは、ランタイムにより呼び出し元に返信されます。

> **Note**
>
> ログ記録にはCloudWatch 、ログクォータ が適用されます。ログデータは、スロットリングが原因で失われることがあります。また、場合によっては、関数のインスタンス停止時に失われることがあります。

Lambda は、需要の増加に応じて追加のインスタンスを実行し、需要の減少に応じてインスタンスを停止することで関数をスケーリングします。このモデルは、次のようなアプリケーションアーキテクチャにおいてばらつきが生じます。

- 特に明記されていない限り、受信リクエストは、順不同または同時に処理されます。

- 関数のインスタンスが長く存続することを想定せず、アプリケーションの状態を別の場所に保存します。
- ローカルストレージとクラスレベルのオブジェクトを使用することで、パフォーマンスを向上させられます。その場合でも、デプロイパッケージのサイズと実行環境に転送するデータの量は最小限に抑えてください。

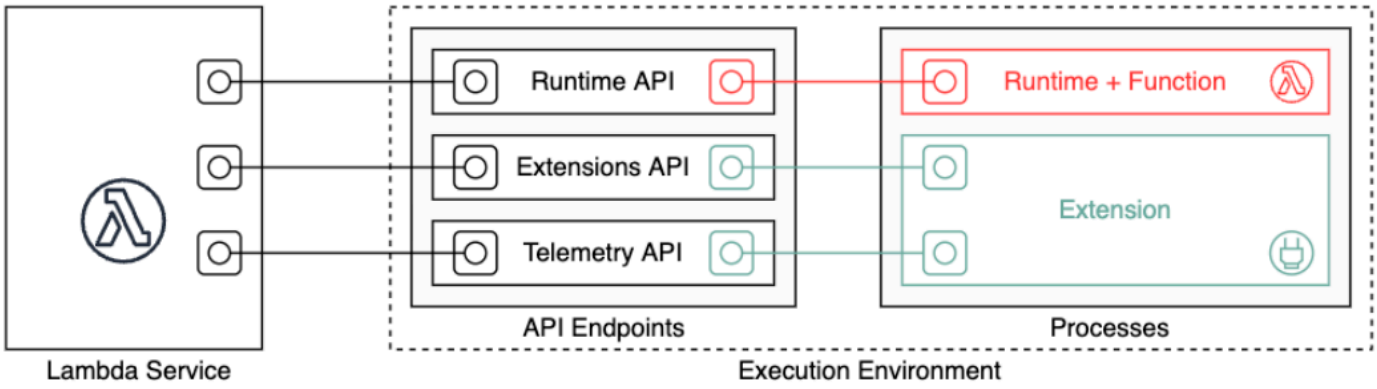
プログラミング言語別のプログラミングモデルの実践的な入門ガイドについては、以下の章を参照してください。

- Node.js による Lambda 関数の構築
- Python による Lambda 関数の構築
- Ruby による Lambda 関数の構築
- Java による Lambda 関数の構築
- Go による Lambda 関数の構築
- C# による Lambda 関数の構築
- PowerShell による Lambda 関数の構築

Lambda 実行環境

Lambda は、実行環境で関数を呼び出します。これにより、安全で分離されたランタイム環境が提供されます。実行環境は、関数の実行に必要なリソースを管理します。また、関数のランタイム、および関数に関連付けられた外部拡張機能のライフサイクルサポートも提供します。

関数のランタイムは、ランタイム API を使用して Lambda と通信します。拡張機能は、拡張機能 API を使用して Lambda と通信します。拡張機能は、Telemetry API を使用することで、関数からログメッセージとその他のテレメトリを受け取ることもできます。



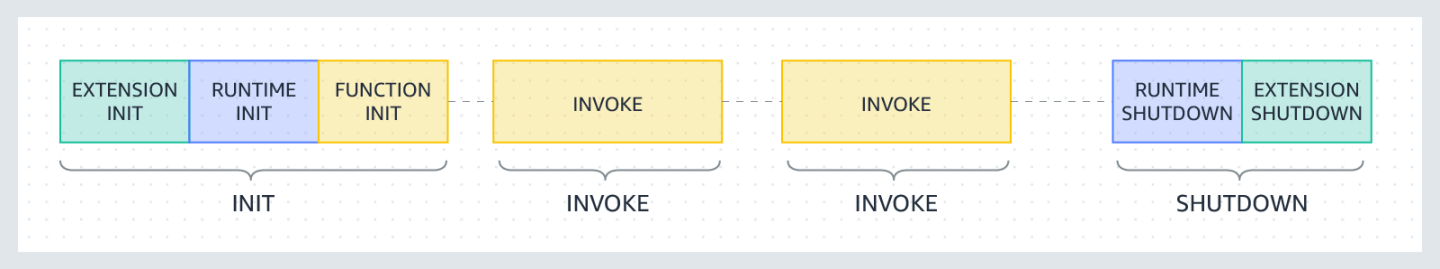
Lambda 関数を作成する際に、関数に許可するメモリ容量や最大実行時間など、設定情報を指定します。Lambda はこの情報を使用して実行環境をセットアップします。

関数のランタイムと各外部拡張機能は、実行環境内で実行されるプロセスです。アクセス許可、リソース、認証情報、および環境変数は、関数と拡張機能の間で共有されます。

トピック

- Lambda 実行環境のライフサイクル
- 関数にステートレスを実装する

Lambda 実行環境のライフサイクル



各フェーズは、ランタイムと、登録されているすべての拡張機能に Lambda を送信するイベントから始まります。ランタイムと各拡張機能は、Next API リクエストを送信することで完了を示します。Lambda は、ランタイムと各拡張機能が完了して保留中のイベントがなくなると、実行環境をフリーズします。

トピック

初期化フェーズ

Init フェーズでは、Lambda は次の 3 つのタスクを実行します。

- すべての拡張機能を起動する (Extension init)
- ランタイムをブートストラップする (Runtime init)
- 関数の静的コード (Function init) を実行する
- 任意の beforeCheckpoint ランタイムフックを実行する (Lambda SnapStart のみ)

この Init フェーズは、ランタイムとすべての拡張機能が Next API リクエストを送信して準備完了を示したときに終了します。Init フェーズは 10 秒に制限されています。3 つのタスクすべてが 10 秒以内に完了しない場合、Lambda は最初の関数呼び出し時に、設定された関数タイムアウトで Init フェーズを再試行します。

Lambda SnapStart がアクティブ化されると、関数バージョンの発行時に Init フェーズが開始されます。Lambda は、初期化された実行環境のメモリとディスク状態のスナップショットを保存し、暗号化されたスナップショットを永続化して、低レイテンシーアクセスのためにスナップショットをキャッシュします。beforeCheckpoint ランタイムフックがある場合、コードは Init フェーズの最後に実行されます。

> **Note**
>
> 10 秒のタイムアウトは、プロビジョニングされた同時実行または SnapStart を使用している関数には適用されません。プロビジョニングされた同時実行関数と SnapStart 関数の場合、初期化コードは最大 15 分間実行できます。制限時間は 130 秒、または設定されている関数のタイムアウト (最大 900 秒) のいずれか長い方です。

プロビジョニング済み同時実行を使用する場合、関数の PC 設定を行う際、Lambda は実行環境を初期化します。また Lambda は、初期化された実行環境が呼び出し前に常に使用できるようにします。関数の呼び出しフェーズと初期化フェーズの間に予期しないギャップが発生する場合があります。関数のランタイムとメモリ設定によっては、初期化された実行環境での最初の呼び出しでレイテンシーの変動が発生する場合があります。

オンデマンド同時実行を使用する関数の場合、Lambda は呼び出しリクエストの前に実行環境を初期化することがあります。これが発生すると、関数の初期化フェーズと呼び出しフェーズの間に時間のギャップが発生することがあります。この動作に依存しないことをお勧めします。

初期化フェーズ中の失敗

`Init` フェーズ中に関数がクラッシュするかタイムアウトすると、Lambda は `INIT_REPORT` ログにエラー情報を出力します。

Example — タイムアウトの INIT_REPORT ログ

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example — 拡張が失敗したときの INIT_REPORT ログ

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

`Init` フェーズが成功した場合、SnapStart がアクティブ化されていない限り、Lambda は `INIT_REPORT` ログを生成しません。SnapStart 関数は常に `INIT_REPORT` を生成します。詳細については、「Lambda のモニタリング SnapStart」を参照してください。

復元フェーズ (Lambda SnapStart のみ)

SnapStart 関数を初めて呼び出し、その関数がスケールアップすると、Lambda は関数をゼロから初期化するのではなく、永続化されたスナップショットから新しい実行環境を再開しま

す。afterRestore() ランタイムフックがある場合、コードは Restore フェーズの最後に実行されます。ユーザーには、afterRestore() ランタイムフックの所要時間分の料金が請求されます。タイムアウト制限 (10 秒) 内にランタイム (JVM) がロードされ、afterRestore() ランタイムフックが完了される必要があります。その時間を超えると、SnapStartTimeoutException が発生します。Restore フェーズが完了すると、Lambda が関数ハンドラーを呼び出します (呼び出しフェーズ)。

復元フェーズ中の失敗

Restore フェーズが失敗した場合、Lambda は RESTORE_REPORT ログにエラー情報を出力します。

Example — タイムアウトの RESTORE_REPORT ログ

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example — ランタイムフック障害の RESTORE_REPORT ログ

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

RESTORE_REPORT ログの詳細については、「Lambda のモニタリング SnapStart」を参照してください。

呼び出しフェーズ

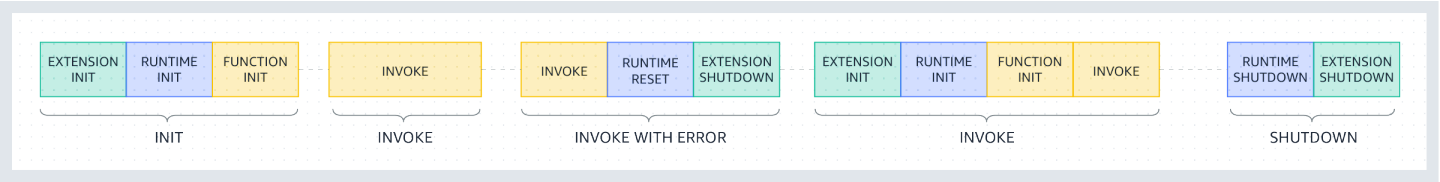
Next API リクエストに応答して Lambda 関数が呼び出されると、Lambda はランタイムと各拡張機能に Invoke イベントを送信します。

関数のタイムアウト設定は、Invoke フェーズ全体の所要時間を制限します。例えば、関数のタイムアウトを 360 秒に設定した場合、関数とすべての拡張機能は 360 秒以内に完了する必要があります。独立した呼び出し後フェーズはないことに注意してください。所要時間は、すべての呼び出し時間 (ランタイム + 拡張機能) の合計であり、関数とすべての拡張機能の実行が終了するまで計算されません。

呼び出しフェーズはランタイム後に終了し、すべての拡張機能は Next API リクエストを送信して、完了したことを示します。

呼び出しフェーズ中の失敗

Lambda 関数が Invoke フェーズ中にクラッシュするかタイムアウトすると、Lambda は実行環境をリセットします。次の図は、呼び出しに失敗した場合の Lambda 実行環境の動作を示しています。



前示の図では次のとおりです。

- 最初のフェーズは、エラーなしで実行される INIT フェーズです。
- 2 番目のフェーズは、エラーなしで実行される INVOKE フェーズです。
- ある時点で、関数で呼び出しエラー (関数のタイムアウトやランタイムエラーなど) が発生したとします。INVOKE WITH ERROR というラベルの付いた 3 番目のフェーズは、このシナリオを示しています。これが発生すると、Lambda サービスはリセットを実行します。リセットは Shutdown イベントのように動作します。まず、Lambda はランタイムをシャットダウンし、登録された各外部拡張機能に Shutdown イベントを送信します。イベントには、シャットダウンの理由が含まれます。この環境が新しい呼び出しに使用される場合、Lambda は次の呼び出しとともに拡張機能とランタイムを再び初期化します。

  > **Note**
  >
  > Lambda のリセットでは、次の init フェーズより前の/tmpディレクトリコンテンツはクリアされません。この動作は、通常のシャットダウンフェーズと一致しています。

- 4 番目のフェーズは、呼び出しが失敗した直後の INVOKE フェーズを表します。ここで、Lambda は INIT フェーズを再実行して環境を再び初期化します。これは、抑制された初期化と呼ばれます。抑制された初期化が発生した場合、Lambda は CloudWatch Logs で追加の INIT フェーズを明示的にレポートしません。代わりに、REPORT 行の期間に追加の INIT 期間 + INVOKE 期間が含まれる場合があります。例えば、CloudWatch で次のログが表示されたとします。

  ```
  2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
  2022-12-20T01:00:02.500-08:00 END RequestId: XXX
  2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
  Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
  ```

  この例では、REPORT タイムスタンプと START タイムスタンプの差は 2.5 秒です。これは、報告された 3022.91 ミリ秒の期間とは一致しません。なぜなら、Lambda が実行した追加の INIT (抑制された init) が考慮されていないからです。この例では、実際の INVOKE フェーズでは 2.5 秒かかったと推測できます。

この動作に関するより多くのインサイトを得るために、Lambda Telemetry API を使用できます。Telemetry API は、抑制された初期化が呼び出しフェーズの間で発生するたびに、phase=invoke の INIT_START、INIT_RUNTIME_DONE、および INIT_REPORT イベントを発行します。

- 5 番目のフェーズは、エラーなしで実行される SHUTDOWN フェーズを表します。

シャットダウンフェーズ

Lambda は、ランタイムをシャットダウンしようとする際、Shutdown イベントを登録された各外部拡張機能に送信します。拡張機能は、この時間を最終的なクリーンアップタスクに使用できます。Shutdown イベントは、Next API リクエストに対するレスポンスです。

期間: Shutdown フェーズ全体の上限は 2 秒です。ランタイムまたは拡張機能が応答しない場合、Lambda は通知 (SIGKILL) によりそれを終了します。

関数とすべての拡張機能が完了した後、Lambda は別の関数の呼び出しを想定して、実行環境をしばらく維持します。ただし、Lambda は数時間ごとに実行環境を終了し、ランタイムの更新とメンテナンスを許可します。継続的に呼び出される関数であっても、この終了は発生します。実行環境が無期限に保持されると想定すべきではありません。詳細については、「関数にステートレスを実装する」を参照してください。

関数が再び呼び出されると、再利用のため、Lambda によって環境が解凍されます。実行環境の再利用には、次のような意味があります。

- 関数ハンドラーメソッドの外部で宣言されたオブジェクトは、初期化されたままとなり、関数が再度呼び出されると追加の最適化を提供します。例えば、Lambda 関数がデータベース接続を確立する場合、連続した呼び出しでは接続を再確立する代わりに元の接続が使用されます。新しい接続を作成する前に、接続が存在するかどうかを確認するロジックをコードに追加することをお勧めします。
- 各実行環境は、/tmp ディレクトリに 512 MB と 10,240 MB 間を 1 MB 刻みで提供します。ディレクトリのコンテンツは、実行環境が停止された際に維持され、複数の呼び出しに使用できる一時的なキャッシュを提供します。キャッシュに保存したデータが存在するかどうかを確認するための追加コードを追加できます。デプロイのサイズ制限の詳細については、「Lambda クォータ」を参照してください。
- Lambda 関数によって開始され、関数が終了したときに完了しなかったバックグラウンドプロセスまたはコールバックは、Lambda 関数が実行環境を再利用したときに再開します。コードのバックグラウンド処理またはコールバックは、コード終了までに完了させてください。

関数にステートレスを実装する

Lambda 関数コードを記述するときは、実行環境が 1 回の呼び出しに対してのみ存在すると仮定して、実行環境をステートレスとして扱います。Lambda は、継続的に呼び出される関数であっても、実行時の更新とメンテナンスを可能にするために、数時間ごとに実行環境を終了します。関数の起動時に、必要な状態 (Amazon DynamoDB テーブルからショッピングカートを取得するなど) を初期化します。終了する前に、Amazon Simple Storage Service (Amazon S3)、DynamoDB、Amazon Simple Queue Service (Amazon SQS) などの永続的なデータ変更を永続的なストアにコミットします。カウンターや集計など、複数の呼び出しにまたがる既存のデータ構造、一時ファイル、または状態に依存しないようにしてください。これにより、関数は各呼び出しを個別に処理できるようになります。

Lambda デプロイパッケージ

AWS Lambda 関数のコードは、スクリプトまたはコンパイルされたプログラム、さらにそれらの依存関係で構成されます。デプロイパッケージを使用して、Lambda に関数コードをデプロイします。Lambda は、コンテナイメージと .zip ファイルアーカイブの 2 種類のデプロイパッケージをサポートします。

トピック

コンテナイメージ

コンテナイメージには、基盤となるオペレーティングシステム、ランタイム、Lambda の拡張機能、アプリケーションコードとその依存関係が含まれています。また、これらのイメージには、機械学習モデルなどの静的データを追加することもできます。

Lambda では、コンテナイメージをビルドする際に使用できる、オープンソースのベースイメージのセットを提供しています。コンテナイメージを作成およびテストするには、AWS Serverless Application Model (AWS SAM) コマンドラインインターフェイス (CLI)、またはネイティブなコンテナツール (Docker CLI など) を使用できます。

コンテナイメージをマネージド型 AWS コンテナイメージのレジストリサービスである Amazon Elastic Container Registry (Amazon ECR) にアップロードします。関数にイメージをデプロイするには、Lambda コンソール、Lambda API、コマンドラインツール、AWS SDK を使用して Amazon ECR イメージの URL を指定します。

Lambda コンテナイメージの詳細については、「コンテナイメージを使用した Lambda 関数の作成」を参照してください。

.zip ファイルアーカイブ

.zip ファイルアーカイブには、アプリケーションコードとその依存関係が集録されています。Lambda コンソールまたはツールキットを使用して関数を作成する際、コードの.zip ファイルアーカイブが、Lambda により自動的に作成されます。

Lambda API、コマンドラインツール、または AWS SDK を使用して関数を管理する場合は、デプロイパッケージを作成する必要があります。関数がコンパイルされた言語を使用しているか、関数に依存関係を追加する場合、デプロイパッケージを作成する必要があります。関数のコードをデプロイするには、Amazon Simple Storage Service (Amazon S3) またはローカルマシンからデプロイパッケージをアップロードします。

Lambda コンソール、AWS Command Line Interface (AWS CLI) を使用してデプロイパッケージとして .zip ファイルをアップロードしたり、Amazon Simple Storage Service (Amazon S3) バケットにアップロードしたりできます。

Lambda コンソールの使用

次の手順は、Lambda コンソールを使用して .zip ファイルをデプロイパッケージとしてアップロードする方法を示します。

Lambda コンソールに .zip ファイルをアップロードするには

1. Lambda コンソールで [Functions (関数)] ページを開きます。
2. 関数を選択します。
3. [Code Source] (コードソース) ペインで、[Upload from] (からアップロード)、[.zip file] (.zip ファイル) の順に選択します。
4. [Upload] (アップロード) を選択して、ローカルの .zip ファイルを選択します。
5. [Save] を選択します。

AWS CLI の使用

AWS Command Line Interface (AWS CLI) を使用して、.zip ファイルをデプロイパッケージとしてアップロードできます。言語固有の手順については、以下のトピックを参照してください。

Node.js

- .zip ファイルアーカイブで Node.js Lambda 関数をデプロイする

Python

- Python Lambda 関数で .zip ファイルアーカイブを使用する

Ruby

- Ruby Lambda 関数で .zip ファイルアーカイブを使用する

Java

- .zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする

Go

- .zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする

C#

- .zip ファイルアーカイブを使用して C# Lambda 関数を構築し、デプロイする

PowerShell

- .zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする

Amazon S3 の使用

Amazon Simple Storage Service (Amazon S3) を使用して、.zip ファイルをデプロイパッケージとしてアップロードできます。詳細については、「」を参照してください。

レイヤー

.zip ファイルアーカイブを使用して関数コードをデプロイする場合は、ライブラリ、カスタムランタイム、関数についてのその他の依存関係の配信メカニズムとして Lambda レイヤーを使用できます。レイヤーを使用することで、開発中の関数コードを、変更されることのないコードやリソースとは区別しながら管理できるようになります。関数が使用するレイヤーを、お客様ご自身で作成したレイヤー、AWS により提供されるレイヤー、または他の AWS のユーザーから提供されるレイヤーの中から選択して設定できます。

コンテナイメージにレイヤーは使用できません。代わりに、コンテナイメージをビルドする際、任意のランタイム、ライブラリ、その他の依存関係をコンテナイメージ内にパッケージ化します。

レイヤーの詳細については、「Lambda レイヤー」を参照してください。

他の AWS サービスを使用したデプロイパッケージの構築

次のセクションでは、Lambda 関数の依存関係をパッケージ化するために使用できる他の AWS のサービスについて説明します。

C または C++ ライブラリを備えたデプロイパッケージ

デプロイパッケージにネイティブライブラリが含まれている場合は、AWS Serverless Application Model(AWS SAM) でデプロイパッケージを構築できます。AWS SAMCLI `sam build` コマンド

を `--use-container` と共に使用して、デプロイパッケージを作成できます。このオプションでは、Lambda 実行環境と互換性のある Docker イメージ内にデプロイパッケージを構築します。

詳細については、AWS Serverless Application Model デベロッパーガイドの sam build を参照してください。

50 MB を超えるデプロイパッケージ

デプロイパッケージのサイズが 50 MB を超える場合は、関数のコードと依存関係を Amazon S3 バケットにアップロードしてください。

デプロイパッケージを作成し、Lambda 関数を作成する AWS リージョンの Amazon S3 バケットに .zip ファイルをアップロードできます。Lambda 関数を作成する際は、Lambda コンソール上または AWS CLI を使用して S3 バケット名とオブジェクトキー名を指定します。

Amazon S3 コンソールを使用してバケットを作成するには、「Amazon Simple Storage Service ユーザーガイド」の「バケットを作成する」を参照してください。

Lambda と Infrastructure as code (IaC) の使用

Lambda には、コードをデプロイして関数を作成する方法がいくつか用意されています。例えば、Lambda コンソールまたは AWS Command Line Interface (AWS CLI) を使用して、Lambda 関数を手動で作成または更新できます。このような手動オプションに加えて、AWS では Infrastructure as Code (IaC) を使用して Lambda 関数とサーバーレスアプリケーションをデプロイするためのソリューションを多数提供しています。IaC を使用すると、プロセスや設定を手動で行う代わりに、コードを使用して Lambda 関数やその他の AWS リソースをプロビジョニングおよび維持できます。

ほとんどの場合、Lambda 関数は単独では実行されません。代わりに、データベース、キュー、ストレージなどの他のリソースと共に、サーバーレスアプリケーションの一部を形成します。IaC を使用すると、デプロイプロセスを自動化して、多数の個別の AWS リソースを含むサーバーレスアプリケーション全体を迅速かつ繰り返しデプロイおよび更新できます。このアプローチにより、開発サイクルが短縮され、構成管理が容易になり、リソースを毎回同じ方法でデプロイできるようになります。

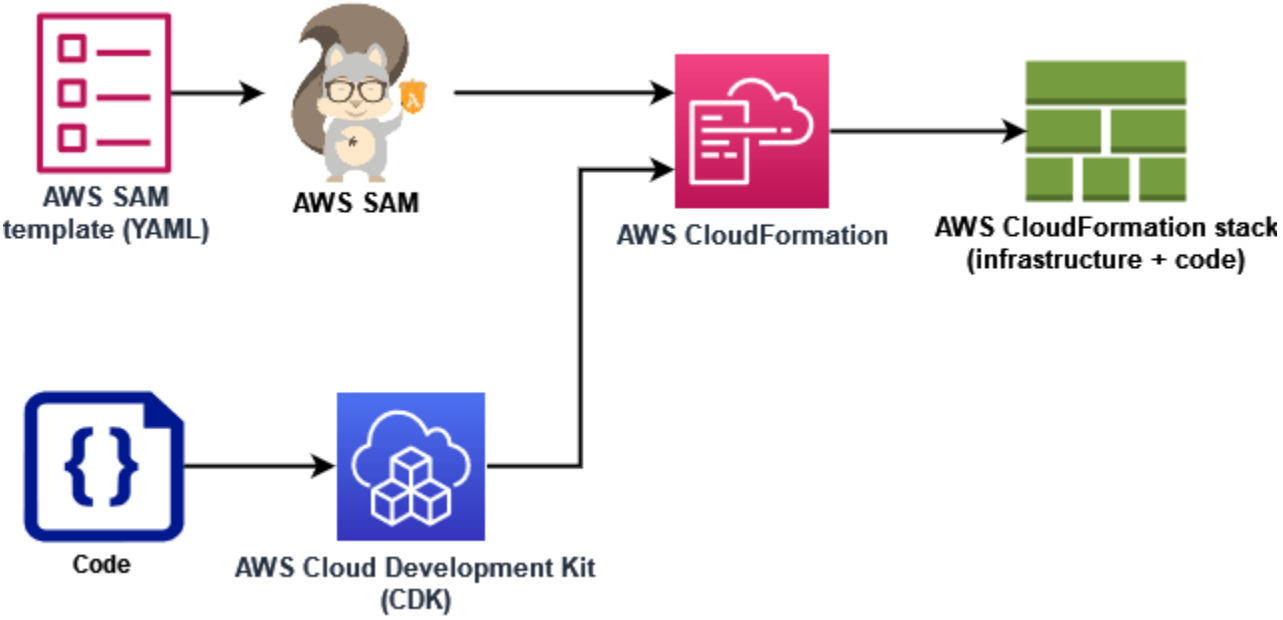
トピック

Lambda 用 IaC ツール

IaC を使用して Lambda 関数とサーバーレスアプリケーションをデプロイするために、AWS ではさまざまなツールやサービスを提供しています。

AWS CloudFormation は、AWS がクラウドリソースを作成および設定するために提供した最初のサービスでした。AWS CloudFormation を使用して、インフラストラクチャとコードを定義するテキストテンプレートを作成します。AWS が新しいサービスを導入し、AWS CloudFormation テンプレートの作成が複雑になっていくにつれて、さらに 2 つのツールがリリースされました。AWS SAM はサーバーレスアプリケーションを定義するための、もう一つのテンプレートベースのフレームワークです。AWS Cloud Development Kit (AWS CDK) は、多くの一般的なプログラミング言語のコード構文を使用してインフラストラクチャを定義およびプロビジョニングするための、コードを優先とするアプローチです。

AWS SAM と AWS CDK を両方使用すると、AWS CloudFormation がバックグラウンドで動作してインフラストラクチャを構築し、デプロイします。次の図は、これらのツールの関係を示しています。また、図の後に各ツールの主要な機能を説明しています。



- AWS CloudFormation - AWSリソースとそのプロパティを記述した YAML または JSON テンプレートを使用して、リソースを CloudFormation モデル化してセットアップします。 は、安全で反復可能な方法でリソースを CloudFormation プロビジョニングするため、手動の手順なしでインフラストラクチャとアプリケーションを頻繁に構築できます。設定を変更すると、 はスタックを更新するために実行する適切なオペレーション CloudFormation を決定します。変更をロールバック CloudFormation することもできます。
- AWS Serverless Application Model (AWS SAM) - AWS SAM は、サーバーレスアプリケーションを定義するためのオープンソースのフレームワークです。AWS SAM テンプレートは、短縮構文を使用して関数、API、データベース、イベントソースマッピングをリソースあたりわずか数行のテキスト (YAML) で定義します。デプロイ中に、AWS SAM は AWS SAM 構文を AWS CloudFormation 構文に変換および拡張します。このため、任意の CloudFormation 構文をAWS SAMテンプレートに追加できます。これにより、 AWS SAMのすべての能力が得られますがCloudFormation、設定行が少なくなります。
- AWS Cloud Development Kit (AWS CDK) - ではAWS CDK、コードコンストラクトを使用してインフラストラクチャを定義し、 を通じてプロビジョニングしますAWS CloudFormation。AWS CDK では、既存の IDE、テストツール、ワークフローパターンを使用して、TypeScript、、Python、Java、.NET、Go (デベロッパープレビュー) でアプリケーションインフラ

ストラクチャをモデル化できます。反復可能なデプロイ、簡単なロールバック、ドリフト検出など、AWS CloudFormation のすべてのメリットが得られます。

また、AWS ではシンプルなグラフィカルインターフェイスを使用して IaC テンプレートを開発する、AWS Application Composer というサービスも提供しています。Application Composer では、ビジュアルキャンバスで AWS のサービスをドラッグ、グループ化、接続することでアプリケーションアーキテクチャを設計します。次に、Application Composer は、アプリケーションのデプロイに使用できるデザインで AWS SAM または AWS CloudFormation テンプレートを作成します。

以下の「the section called “Lambda 用 IaC の開始方法”」セクションでは、Application Composer を使用して、既存の Lambda 関数に基づくサーバーレスアプリケーションのテンプレートを作成します。

Lambda 用 IaC の開始方法

このチュートリアルでは、既存の Lambda 関数で AWS SAM テンプレートを作成し、他の AWS リソースを追加して Application Composer でサーバーレスアプリケーションを構築することで、IaC を Lambda で使用開始できます。

Application Composer を使用せずにテンプレートを操作する方法を学ぶために、AWS SAM または AWS CloudFormation のチュートリアルを実行することから始める場合は、このページの最後にある「the section called “次のステップ”」セクションに他のリソースへのリンクがあります。

このチュートリアルを実行すると、AWS リソースの AWS SAM での指定方法など、いくつかの基本的な概念を習得できます。また、Application Composer で、AWS SAM または AWS CloudFormation を使用してデプロイできるサーバーレスアプリケーションを構築する方法についても習得します。

このチュートリアルを完了するには、次のステップを実行します。

- サンプル Lambda 関数の作成
- Lambda コンソールを使用して、関数の AWS SAM テンプレートを表示します
- 関数の設定を AWS Application Composer にエクスポートし、関数の設定に基づいてシンプルなサーバーレスアプリケーションを設計します。
- 更新した AWS SAM テンプレートを保存して、サーバーレスアプリケーションをデプロイするための基礎として使用できます。

「the section called “次のステップ”」セクションでは、AWS SAM および Application Composer についての詳細を学ぶために使用できるリソースを紹介しています。これらのリソースには、AWS SAM を使用してサーバーレスアプリケーションをデプロイする方法を説明する、より高度なチュートリアルへのリンクが含まれています。

前提条件

このチュートリアルでは、Application Composer のローカル同期機能を使用して、テンプレートとコードファイルをローカルビルドマシンに保存します。この機能を使用するには、File System Access API に対応するブラウザが必要です。これにより、Web アプリケーションでローカルファイルシステム内のファイルの読み取り、書き込み、保存を行うことができます。Google Chrome または Microsoft Edge の使用が推奨されます。File System Access API の詳細については、「What is the File System Access API?」を参照してください。

Lambda 関数を作成する

この最初のステップでは、チュートリアルの残りの部分を完了するために使用できる Lambda 関数を作成します。簡単に言うと、Lambda コンソールで Python 3.11 ランタイムを使用して、基本的な「Hello world」関数を作成します。

コンソールで「Hello world」Lambda 関数を作成するには

1. Lambdaのコンソールを開きます。
2. [機能の作成]を選択します。
3. [一から作成] を選択したままにし、[基本的な情報] の [関数名] に **LambdaIaCDemo** と入力します。
4. [ランタイム] で [Python 3.11] を選択します。
5. [関数の作成] を選択します。

関数の AWS SAM テンプレートを表示する

関数の設定を Application Composer にエクスポートする前に、Lambda コンソールを使用して関数の現在の設定を AWS SAM テンプレートとして表示します。このセクションのステップに従うことで、AWS SAM テンプレートの構造と、Lambda 関数などのリソースを定義してサーバーレスアプリケーションの指定を開始する方法について学びます。

関数の AWS SAM テンプレートを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 先ほど作成した関数 (LambdaIaCDemo) を選択します。
3. [関数の概要] ペインで、[テンプレート] を選択します。

   関数の設定を表す図の代わりに、その関数の AWS SAM テンプレートが表示されます。テンプレートは次のようになります。

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
```

```
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

作成した関数の YAML テンプレートを確認する時間を作り、いくつかの主要な概念について理解しておきましょう。

テンプレートは `Transform: AWS::Serverless-2016-10-31` を宣言することから始まります。この宣言は必須です。というのも、AWS SAM テンプレートはバックグラウンドで AWS CloudFormation を介してデプロイされるからです。`Transform` ステートメントを使用すると、テンプレートが AWS SAM テンプレートファイルとして識別されます。

`Transform` 宣言の後に `Resources` セクションがあります。ここで、AWS SAM テンプレートと共にデプロイする AWS リソースを定義します。AWS SAM テンプレートには、AWS SAM リソースと AWS CloudFormation リソースを組み合わせて含めることができます。これは、デプロイ中に AWS SAM テンプレートが AWS CloudFormation テンプレートに拡張されるためであり、有効な AWS CloudFormation 構文を AWS SAM テンプレートに追加できます。

現時点では、テンプレートの `Resources` セクションで定義されているリソースは Lambda 関数だけです。`LambdaIaCDemo`Lambda 関数を AWS SAM テンプレートに追加するには、`AWS::Serverless::Function` リソースタイプを使用します。Lambda 関数リソースの `Properties` で、関数のランタイム、関数ハンドラー、およびその他の設定オプションを定義します。AWS SAM で関数のデプロイに使用する必要のある関数のソースコードへのパスも、ここで定義されます。の Lambda 関数リソースの詳細についてはAWS SAM、「 AWS SAMデベロッパーガイドAWS::Serverless::Function」の「」を参照してください。

テンプレートでは、関数のプロパティと設定だけでなく、関数の AWS Identity and Access Management (IAM) ポリシーも指定されています。このポリシーは、Amazon CloudWatch Logs にログを書き込むアクセス許可を関数に付与します。Lambda コンソールで関数を作成すると、Lambda は自動的にこのポリシーを関数にアタッチします。AWS SAM テンプレート

で関数の IAM ポリシーを指定する方法の詳細については、 AWS SAMデベロッパーガイドの AWS::Serverless::Functionページの `policies`プロパティを参照してください。

AWS SAM テンプレートの構造の詳細については、「AWS SAM テンプレートの構造」を参照してください。

AWS Application Composer をサーバーレスアプリケーションの設計に使用する

関数の AWS SAM テンプレートを基にして単純なサーバーレスアプリケーションの構築を開始するには、関数の設定を Application Composer にエクスポートし、Application Composer のローカル同期モードを有効化します。ローカル同期で関数のコードと AWS SAM テンプレートがローカルビルドマシンに自動的に保存され、Application Composer に他の AWS リソースを追加しても保存したテンプレートは同期されたままになります。

関数を Application Composer にエクスポートするには

1. [関数の概要] ペインで、[Application Composer にエクスポート] を選択します。

   関数の設定とコードを Application Composer にエクスポートするには、Lambda でアカウントに Amazon S3 バケットを作成し、このデータを一時的に保存します。
2. ダイアログボックスで [プロジェクトの確認と作成] を選択し、このバケットのデフォルト名をそのまま使用して、関数の設定とコードを Application Composer にエクスポートします。
3. (オプション) Lambda で作成する Amazon S3 バケットに別の名前を選択する場合は、新しい名前を入力して [プロジェクトの確認と作成] を選択します。Amazon S3 バケットの名前は、グローバルに一意で、バケットの命名規則に従ったものである必要があります。

   [プロジェクトの確認と作成] を選択すると、Application Composer コンソールが開きます。キャンバスに、Lambda 関数が表示されます。
4. [メニュー] ドロップダウンから [ローカル同期をアクティブ化] を選択します。
5. 表示されたダイアログボックスで [フォルダを選択] を選び、ローカルビルドマシンのフォルダを選択します。
6. [アクティブ化] を選択してローカル同期をアクティブ化します。

関数を Application Composer にエクスポートするには、特定の API アクションを使用するためのアクセス許可が必要です。関数をエクスポートできない場合は、the section called “必要なアクセス許可” を参照して、必要な権限があることを確認してください。

> **Note**
>
> 関数を Application Composer にエクスポートするときは、Lambda が作成するバケットに標準の Amazon S3 の料金を適用します。Lambda がバケットに取り込むオブジェクトは 10 日後に自動的に削除されますが、Lambda でバケット自体が削除されることはありません。追加料金が AWS アカウントにかからないようにするには、関数を Application Composer にエクスポートしてから「バケットの削除」の手順に従ってください。Lambda で作成される Amazon S3 バケットの詳細については、「the section called “Application Composer”」を参照してください。

Application Composer でサーバーレスアプリケーションを設計するには

ローカル同期を有効化すると、Application Composer で行った変更がローカルビルドマシンに保存されている AWS SAM テンプレートに反映されます。追加の AWS リソースを Application Composer のキャンバスにドラッグアンドドロップして、アプリケーションを構築できるようになりました。この例では、Lambda 関数のトリガーとして Amazon SQS のシンプルなキューを追加し、データを書き込む関数に DynamoDB テーブルを追加します。

1. Amazon SQS トリガーを Lambda 関数に追加するには、次の手順を実行します。

   a. [リソース] パレットの検索フィールドに、「**SQS**」と入力します。
   b. [SQS キュー] リソースをキャンバスにドラッグし、Lambda 関数の左側に配置します。
   c. [詳細] を選択し、[論理 ID] に「**LambdaIaCQueue**」と入力します。
   d. [保存] を選択します。
   e. SQS キューカードの [サブスクリプション] ポートをクリックし、Lambda 関数カードの左側のポートにドラッグして、Amazon SQS リソースと Lambda リソースを接続します。2 つのリソースの間に線が表示されれば、接続に成功しています。また、Application Composer では、2 つのリソースが正常に接続されたことを示すメッセージもキャンバスの下部に表示されます。

2. 以下を実行して、Lambda 関数に Amazon DynamoDB テーブルを追加し、データを書き込みます。

   a. [リソース] パレットの検索フィールドに、「**DynamoDB**」と入力します。
   b. [DynamoDB テーブル] リソースをキャンバスにドラッグし、Lambda 関数の右側に配置します。
   c. [詳細] を選択し、[論理 ID] に「**LambdaIaCTable**」と入力します。

d. [保存] を選択します。

e. DynamoDB テーブルを Lambda 関数に接続するには、Lambda 関数カードの右側のポートをクリックして、DynamoDB カードの左側のポートにドラッグします。

これらのリソースを追加したところで、Application Composer が作成した更新済みの AWS SAM テンプレートを見てみましょう。

更新した AWS SAM テンプレートを表示するには

- Application Composer のキャンバスで [テンプレート] を選択し、キャンバスビューからテンプレートビューに切り替えます。

これで、AWS SAM テンプレートに以下の追加のリソースとプロパティが含まれているはずです。

- 識別子 `LambdaIaCQueue` を含む Amazon SQS キュー

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

Application Composer を使用して Amazon SQS キューを追加すると、Application Composer で `MessageRetentionPeriod` プロパティが設定されます。SQS キューカードで [詳細] を選択し、[FIFO キュー] をオンまたはオフにして `FifoQueue` プロパティを設定することもできます。

キューに他のプロパティを設定するには、テンプレートを手動で編集して追加できます。`AWS::SQS::Queue` のリソースと使用可能なプロパティの詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::SQS::Queue」を参照してください。

- Amazon SQS キューを関数のトリガーとして指定する Lambda 関数定義の `Events` プロパティ

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

Events プロパティは、1 つのイベントタイプと、そのタイプに依存する一連のプロパティで構成されます。Lambda 関数をトリガーするように設定AWS のサービスできるさまざまな と設定できるプロパティについては、「 AWS SAMデベロッパーガイドEventSource」の「」を参照してください。

- 識別子 LambdaIaCTable を含む DynamoDB テーブル

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Application Composer を使用して DynamoDB テーブルを追加するときに、DynamoDB テーブルカードで [詳細] を選択し、キー値を編集することでテーブルのキーを設定できます。Application Composer は、BillingMode や StreamViewType を含む他の多くのプロパティにもデフォルト値を設定します。

これらのプロパティやお使いの AWS SAM テンプレートに追加できるその他のプロパティの詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::DynamoDB::Table」を参照してください。

- 追加した DynamoDB テーブルで CRUD オペレーションを実行する権限を関数に付与する新しい IAM ポリシー。

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

以下の例は、最終的に完成した AWS SAM テンプレートを示しています。

```
AWSTemplateFormatVersion: '2010-09-09'
```

```
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
```

```
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

(オプション) AWS SAM を使用したサーバーレスアプリケーションのデプロイ

AWS SAM を使用し、Application Composer で作成したばかりのテンプレートを使用してサーバーレスアプリケーションをデプロイする場合は、まず AWS SAM CLI をインストールする必要があります。これを行うには、「AWS SAM CLI のインストール」の指示に従います。

アプリケーションをデプロイする前に、Application Composer がテンプレートと共に保存した関数コードも更新する必要があります。現時点では、Application Composer が保存した `lambda_function.py` ファイルには、関数を作成したときに Lambda によって提供された基本的な「Hello world」コードのみが含まれています。

関数コードを更新するには、以下のコードをコピーして、Application Composer がローカルビルドマシンに保存した `lambda_function.py` ファイルに貼り付けます。ローカル同期モードをアクティブ化したときのこのファイルの保存先として Application Composer のディレクトリを指定しました。

このコードによって、Application Composer で作成した Amazon SQS キューからのメッセージに含まれるキーと値のペアを受け入れます。キーと値の両方が文字列の場合、コードはそれらの文字列を使用して、テンプレートで定義されている DynamoDB テーブルに項目を書き込みます。

更新済みの Python 関数コード

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

サーバーレスアプリケーションをデプロイするには

AWS SAM CLI を使用してアプリケーションをデプロイするには、次の手順を実行します。関数が正しく構築およびデプロイされるように、Python バージョン 3.11 をお使いのビルドマシンと PATH にインストールする必要があります。

1. Application Composer で `template.yaml` ファイルおよび `lambda_function.py` ファイルを保存したディレクトリから、以下のコマンドを実行します。

   ```
   sam build
   ```

   このコマンドによって、アプリケーションのビルドアーティファクトが収集され、それらをデプロイする場所を適切な形式で配置します。
2. アプリケーションをデプロイし、AWS SAM テンプレートで指定された Lambda、Amazon SQS、DynamoDB の各リソースを作成するには、次のコマンドを実行します。

   ```
   sam deploy --guided
   ```

   `--guided` フラグを使用すると、AWS SAM にデプロイプロセスの手順が示されます。このデプロイでは、Enter キーを押してデフォルトのオプションをそのまま使用してください。

デプロイプロセス中に、AWS SAM でお使いの AWS アカウントに次のリソースが作成されます。

- `sam-app` という名前の AWS CloudFormation スタック
- `sam-app-LambdaIaCDemo-99VXPpYQVv1M` 名前形式の Lambda 関数
- `sam-app-LambdaIaCQueue-xL87VeKsGiIo` 名前形式の Amazon SQS キュー
- `sam-app-LambdaIaCTable-CN0S66C0VLNV` 名前形式の DynamoDB テーブル

また、Lambda 関数が Amazon SQS キューからメッセージを読み取り、DynamoDB テーブルで CRUD オペレーションを実行できるように、AWS SAM で必要な IAM ロールとポリシーを作成します。

AWS SAM を使用してサーバーレスアプリケーションをデプロイする方法の詳細については、「the section called “次のステップ”」セクションのリソースを参照してください。

デプロイしたアプリケーションのテスト (オプション)

サーバーレスアプリケーションが正しくデプロイされたことを確認するには、メッセージをキーと値のペアを含む Amazon SQS キューに送信し、Lambda がこれらの値を使用して DynamoDB テーブルに項目を書き込むことを確認します。

サーバーレスアプリケーションをテストするには

1. Amazon SQS コンソールの [キュー] ページを開き、お使いのテンプレートで AWS SAM が作成したキューを選択します。名前の形式は `sam-app-LambdaIaCQueue-xL87VeKsGiIo` になります。
2. [メッセージを送受信] を選択し、[メッセージの送信] セクションの [メッセージ本文] に次の JSON を貼り付けます。

   ```
   {
       "myKey": "myValue"
   }
   ```

3. [メッセージの送信] を選択します。

   メッセージをキューに送信すると、Lambda はお使いの AWS SAM テンプレートで定義されたイベントソースマッピングを通じて関数を呼び出します。Lambda が意図したとおりに関数を呼び出したことを確認するには、DynamoDB テーブルに項目が追加されていることを確認します。
4. DynamoDB コンソールの [テーブル] ページを開いてテーブルを選択します。名前の形式は `sam-app-LambdaIaCTable-CN0S66C0VLNV` になります。
5. [テーブルアイテムの探索] を選択します。[返された項目] ペインに、[id] `myKey` と [値] `myValue` がある項目が表示されるはずです。

次のステップ

Application Composer を AWS SAM と AWS CloudFormation で使用する方法を学習する際は、まず「Using Application Composer with AWS CloudFormation and AWS SAM」から始めてください。

AWS SAM を使用して Application Composer で設計されたサーバーレスアプリケーションをデプロイするためのガイド付きチュートリアルを実行する際は、「AWS Serverless Patterns Workshop」の「AWS Application Composer tutorial」も実行することをお勧めします。

AWS SAM では、AWS SAM テンプレートやサポートされているサードパーティーの統合と併用できるコマンドラインインターフェイス (CLI) を提供して、サーバーレスアプリケーションを構築し、実行します。AWS SAM CLI を使用すると、アプリケーションの構築とデプロイ、ローカルテストとデバッグの実行、CI/CD パイプラインの設定などを行うことができます。AWS SAM CLI の使用に関する詳細については、「AWS Serverless Application Model 開発者ガイド」の「AWS SAM の使用開始」を参照してください。

AWS CloudFormation コンソールを使用して AWS SAM テンプレートでサーバーレスアプリケーションをデプロイする方法を学習する際は、「AWS CloudFormation ユーザーガイド」の「AWS CloudFormation コンソールの使用」から始めてください。

Lambda と Application Composer との統合がサポートされているリージョン

Lambda と Application Composer との統合は、以下の AWS リージョン でサポートされています。

- 米国東部 (バージニア北部)
- 米国東部 (オハイオ)
- 米国西部 (北カリフォルニア)
- 米国西部 (オレゴン)
- アフリカ (ケープタウン)
- アジアパシフィック (香港)
- アジアパシフィック (ハイデラバード)
- アジアパシフィック (ジャカルタ)
- アジアパシフィック (メルボルン)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (大阪)
- アジアパシフィック (ソウル)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- アジアパシフィック (東京)
- カナダ (中部)
- 欧州 (フランクフルト)
- 欧州 (チューリッヒ)

- 欧州 (アイルランド)
- 欧州 (ロンドン)
- 欧州 (ストックホルム)
- 中東 (アラブ首長国連邦)

VPC によるプライベートネットワーク

Amazon Virtual Private Cloud (Amazon VPC) は、AWS アカウント専用の、AWS クラウド上の仮想ネットワークです。Amazon VPC を使用して、データベース、キャッシュインスタンス、または内部サービスなどのリソースのプライベートネットワークを作成することができます。Amazon VPC の詳細については、Amazon VPC とは?を参照してください。

Lambda 関数は、常に Lambda サービスが所有する VPC 内で実行されます。Lambda はこの VPC にネットワークアクセスとセキュリティルールを適用し、VPC を自動的に維持および監視します。Lambda 関数がアカウント VPC 内のリソースにアクセスする必要がある場合は、VPC にアクセスするための関数を設定します。Lambda は、Hyperplane ENI という名前のマネージドリソースを提供します。これは、Lambda 関数が Lambda VPC からアカウント VPC 内の ENI (Elastic Network Interface) に接続するために使用します。

VPC または Hyperplane ENI は追加料金なしで使用できます。NAT ゲートウェイなど、一部の VPC コンポーネントには料金が発生します。詳細については、「Amazon VPC の料金」を参照してください。

トピック

VPC のネットワーク要素

Amazon VPC ネットワークには、次のネットワーク要素が含まれます。

- Elastic Network Interface — Elastic Network Interface は、仮想ネットワークカードを表す VPC 内の論理ネットワーキングコンポーネントです。

- サブネット — VPC の IP アドレスの範囲。AWS リソースは、指定したサブネット内に追加できます。インターネットに接続する必要があるリソースにはパブリックサブネットを、インターネットに接続しないリソースにはプライベートサブネットを使用してください。
- セキュリティグループ — 各サブネットの AWS リソースへのアクセスを制御するために、セキュリティグループを使用します。
- アクセスコントロールリスト (ACL) — ネットワーク ACL を使用して、サブネット内のセキュリティを強化します。デフォルトのサブネット ACL では、すべてのインバウンドトラフィックとアウトバウンドトラフィックを許可します。
- ルートテーブル - AWS が VPC のネットワークトラフィックを指示するために使用するルートのセットが含まれています。サブネットを特定のルートテーブルに明示的に関連付けることができます。デフォルトでは、サブネットはメインルートテーブルと関連付けられています。
- ルート — ルートテーブル内の各ルートは、IP アドレスの範囲と、Lambda がその範囲のトラフィックを送信する宛先を指定します。ルートはターゲット (トラフィックの送信に使用するゲートウェイ、ネットワークインターフェイス、または接続) も指定します。
- NAT ゲートウェイ - VPC のプライベートサブネットからインターネットへのアクセスを制御する AWS ネットワークアドレス変換 (NAT) サービス。
- VPC エンドポイント - Amazon VPC エンドポイントを使用すると、インターネット経由または NAT デバイス、VPN 接続、AWS Direct Connect 接続を必要とせずに、AWS でホストされているサービスへのプライベート接続を作成することができます。詳細については、「AWS PrivateLink および VPC エンドポイント」を参照してください。

> **Tip**
>
> VPC とサブネットにアクセスするように Lambda 関数を設定するには、Lambda コンソールまたは API を使用します。
> 関数を設定するには、「CreateFunction」の「`VpcConfig`」セクションを参照してください。詳細なステップについては、「AWS アカウント の Amazon VPC への Lambda 関数のアタッチ」を参照してください。

Amazon VPC のネットワーキング定義の詳細については、「Amazon VPC デベロッパーガイド」の「Amazon VPC の仕組み」およびアマゾン VPC に関するよくある質問を参照してください。

Lambda 関数を VPC に接続する

Lambda 関数は、常に Lambda サービスが所有する VPC 内で実行されます。デフォルトでは、Lambda 関数はアカウントの VPC に接続されていません。アカウントの VPC に関数を接続すると、VPC からアクセス権が付与されない限り、関数はインターネットにアクセスできません。

Lambda は、Hyperplane ENI を使用して VPC 内のリソースにアクセスします。Hyperplane ENI は、VPC-to-VPC NAT (V2N) を使用して、Lambda VPC からアカウント VPC への NAT 機能を提供します。V2N は Lambda VPC からアカウント VPC への接続を提供しますが、その逆方向には接続できません。

Lambda 関数を作成する (または VPC 設定を更新すると)、Lambda は、関数の VPC のサブネット構成ごとに Hyperplane ENI を割り当てます。同じサブネットとセキュリティグループを共有する場合、複数の Lambda 関数が 1 つのネットワークインターフェイスを共有できます。

他の AWS サービスに接続する場合、VPC とサポートされた AWS サービス間のプライベート通信に VPC エンドポイントを使用することができます。代わりの方法は、NAT ゲートウェイを使用して、アウトバウンドトラフィックを別の AWS サービスにルーティングすることです。

インターネットへのアクセス権を関数に付与するには、アウトバウンドトラフィックをパブリックサブネットの NAT ゲートウェイにルーティングします。NAT ゲートウェイにはパブリック IP アドレスがあるため、VPC のインターネットゲートウェイを介してインターネットに接続できます。詳細については、VPC に接続された Lambda 関数にインターネットアクセスを有効にする を参照してください。

共有サブネット

VPC 共有を使用すると、複数の AWS アカウントで自分のアプリケーションリソース (Amazon EC2 インスタンス、Lambda 関数など) を、共有され一元管理されている仮想プライベートクラウド (VPC) 内に作成できます。このモデルでは、VPC を所有するアカウント (所有者) は、同じ AWS Organization に属する他のアカウント (参加者) と 1 つまたは複数のサブネットを共有します。

プライベートリソースにアクセスするには、関数を VPC 内のプライベート共有サブネットに接続します。関数をサブネットに接続するには、サブネットの所有者がユーザーにサブネットを共有している必要があります。サブネットの所有者は、後でサブネットの共有を解除して、接続を解除することもできます。共有サブネットの VPC リソースを共有、共有解除、管理する方法についての詳細は、Amazon VPC ガイドの「VPC を他のアカウントと共有する方法」を参照してください。

Lambda Hyperplane ENI

Hyperplane ENI は、Lambda サービスが作成および管理するマネージドネットワークリソースです。Lambda VPC 内の複数の実行環境では、Hyperplane ENI を使用して、アカウント内の VPC 内のリソースに安全にアクセスできます。Hyperplane ENI は、Lambda VPC からアカウント VPC への NAT 機能を提供します。

Lambda は、各サブネットでセキュリティグループの固有のセットごとにネットワークインターフェイスを作成します。同じサブネットとセキュリティグループの組み合わせを共有するアカウント内の関数は、同じネットワークインターフェイスを使用します。本来ならセキュリティグループの設定で追跡を指定する必要がある場合でも、Hyperplane レイヤーで確立された接続は自動的に追跡されます。確立された接続に対応しない VPC からのインバウンドパケットは、Hyperplane レイヤーでドロップされます。詳細については、「Amazon EC2 ユーザーガイド」の「セキュリティグループの接続の追跡」を参照してください。

アカウントの関数は ENI リソースを共有するため、ENI ライフサイクルは他の Lambda リソースよりも複雑です。以下のセクションでは、ENI のライフサイクルについて説明します。

ENI ライフサイクル

- ENI の作成
- ENI の管理
- ENI の削除

ENI の作成

Lambda は、新しく作成された VPC 対応関数、または既存の関数に対する VPC 設定の変更のために Hyperplane ENI リソースを作成することがあります。Lambda が必要なリソースを作成している間、関数は保留状態のままになります。Hyperplane ENI の準備が整ったら、関数がアクティブ状態に移行し、ENI が使用可能になります。Lambda は Hyperplane ENI を作成するのに数分かかる場合があります。

新しく作成された VPC 対応関数の場合、関数上で動作する呼び出しまたはその他の API アクションは、関数の状態がアクティブに移行するまで失敗します。

既存の関数への VPC 設定の変更の場合、関数の状態がアクティブに移行するまで、関数の呼び出しは、古いサブネットおよびセキュリティグループ設定に関連付けられた Hyperplane ENI を引き続き使用します。

Lambda 関数が 30 日間アイドル状態のままである場合、Lambda は未使用の Hyperplane ENI を回収し、関数の状態をアイドルに設定します 次の呼び出しにより、Lambda はアイドル関数を再アクティブ化します。呼び出しは失敗し、Lambda が Hyperplane ENI の作成または割り当てを完了するまで、関数は保留状態になります。

関数の状態の詳細については、「Lambda 関数の状態」を参照してください。

ENI の管理

Lambda は、関数実行ロールのアクセス許可を使用して、ネットワークインターフェイスを作成および管理します。アカウントで VPC 対応関数に一意のサブネットとセキュリティグループの組み合わせを定義すると、Lambda は Hyperplane ENI を作成します。Lambda は、同じサブネットとセキュリティグループの組み合わせを使用するアカウントの他の VPC 対応関数に Hyperplane ENI を再利用します。

同じ Hyperplane ENI を使用できる Lambda 関数の数にクォータはありません。ただし、各 Hyperplane ENI は最大 65,000 個の接続/ポートをサポートします。接続数が 65,000 を超えると、Lambda は新しい Hyperplane ENI を作成して、追加の接続を提供します。

関数設定を更新して別の VPC にアクセスすると、Lambda は以前の VPC の Hyperplane ENI への接続を終了します。新しい VPC への接続を更新するプロセスには、数分かかる場合があります。この間、関数への呼び出しは以前の VPC を使用し続けます。更新が完了すると、新しい VPC で Hyperplane ENI を使用して新しい呼び出しが開始されます。この時点で、Lambda 関数は以前の VPC に接続されなくなります。

ENI の削除

関数を更新してその VPC 設定を削除する場合、Lambda はアタッチされた Hyperplane ENI を削除するのに最大 20 分かかります。Lambda は、他の関数 (または発行された関数のバージョン) がその Hyperplane ENI を使用していない場合にのみ ENI を削除します。

Lambda は、関数実行ロールのアクセス許可により、Hyperplane ENI を削除します。Lambda が Hyperplane ENI を削除する前に実行ロールを削除すると、Lambda はHyperplane ENI を削除できません。削除は手動で実行できます。

Lambda は、アカウントの関数または関数バージョンで使用されているネットワークインターフェイスを削除しません。Lambda ENI Finder を使用して、Hyperplane ENI を使用している関数または関数のバージョンを特定することができます。不要になった関数または関数のバージョンについては、Lambda が Hyperplane ENI を削除するように VPC 設定を削除できます。

接続

Lambda では、TCP (伝送制御プロトコル) と UDP (ユーザーデータグラムプロトコル) の 2 種類の接続がサポートされています。

VPC を作成する際、Lambda は、DHCP オプションのセットを自動的に作成し、VPC に関連付けます。VPC 用に独自の DHCP オプションセットを設定できます。詳細については、Amazon VPC DHCP オプションを参照してください。

Amazon は、お客様の VPC 用の DNS サーバー (Amazon Route 53 Resolver) を提供しています。詳細については、VPC の DNS サポートを参照してください。

IPv6 サポート

Lambda は、Lambda のパブリックデュアルスタックエンドポイントへのインバウンド接続と、IPv6 経由のデュアルスタック VPC サブネットへのアウトバウンド接続をサポートします。

インバウンド

IPv6 経由で関数を呼び出すには、Lambda のパブリックデュアルスタックエンドポイントを使用します。IPv4 と IPv6 の両方をサポートするデュアルスタックのエンドポイント Lambda デュアルスタックエンドポイントは次の構文を使用します。

```
protocol://lambda.us-east-1.api.aws
```

また、Lambda 関数 URL を使用して IPv6 経由で関数を呼び出すこともできます。関数 URL のエンドポイントでは、次の形式を使用します。

```
https://url-id.lambda-url.us-east-1.on.aws
```

アウトバウンド

関数は IPv6 経由でデュアルスタック VPC サブネット内のリソースに接続することができます。このオプションはデフォルトでオフに設定されています。アウトバウンド IPv6 トラフィックを許可するには、コンソールを使用するか、`--vpc-config Ipv6AllowedForDualStack=true` オプションを create-function または update-function-configuration コマンドに指定します。

> **Note**
>
> VPC でアウトバウンド IPv6 トラフィックを許可するには、関数に接続されているすべてのサブネットがデュアルスタックサブネットである必要があります。Lambda は、VPC 内の IPv6 専用サブネットのアウトバウンド IPv6 接続、VPC に接続されていない関数のアウトバウンド IPv6 接続、または VPC エンドポイント (AWS PrivateLink) を使用するインバウンド IPv6 接続をサポートしていません。

IPv6 経由でサブネットリソースに明示的に接続するように関数コードを更新することができます。次の Python の例では、ソケットを開いて IPv6 サーバーに接続します。

Example — IPv6 サーバへの接続

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

セキュリティ

AWS は、VPC のセキュリティを高めるために、セキュリティグループとネットワーク ACL を提供します。セキュリティグループは、リソースのインバウンドトラフィックとアウトバウンドトラフィックをコントロールします。ネットワーク ACL は、サブネットのインバウンドトラフィックとアウトバウンドトラフィックをコントロールします。セキュリティグループは、ほとんどのサブネットに対して十分なアクセス制御を提供します。VPC に追加のセキュリティレイヤーが必要な場合は、ネットワーク ACL を使用できます。詳細については、Amazon VPC でのインターネットワークトラフィックのプライバシーを参照してください。作成するサブネットはすべて、VPC のデフォルトのネットワーク ACL に自動的に関連付けられます。この関連付けを変更し、デフォルトのネットワーク ACL のコンテンツを変更できます。

一般的なセキュリティのベストプラクティスについては、VPC のセキュリティのベストプラクティスを参照してください。Lambda API とリソースへのアクセスを管理するために IAM を使用する方法の詳細については、AWS Lambda アクセス権限を参照してください。

VPC 設定で Lambda 固有の条件キーを使用して、Lambda 関数に追加のアクセス許可コントロールを提供できます。VPC 条件キーの詳細については、VPC 設定で IAM 条件キーを使用するを参照してください。

> **Note**
>
> Lambda 関数は、パブリックインターネットまたは AWS PrivateLink エンドポイントから呼び出すことができます。関数 URL には、パブリックインターネット経由でしかアクセスできません。Lambda 関数は AWS PrivateLink をサポートしていますが、関数 URL ではサポートされません。

オブザーバビリティ

VPC フローログを使用して、ネットワークインターフェイス間で送受信される IP トラフィックに関する情報を取得できます。フローログデータは Amazon CloudWatch Logs または Amazon S3 に発行できます。フローログを作成すると、選択した送信先でそのデータを取得して表示できます。

注: 関数を VPC にアタッチすると、CloudWatch ログメッセージは VPC ルートを使用しません。Lambda はログの通常のルーティングを使用して送信します。

Lambda 関数の命令セットアーキテクチャの設定

Lambda 関数の命令セットアーキテクチャは、Lambda が関数の実行に使用するコンピュータプロセッサのタイプを決定します。Lambda は、命令セットアーキテクチャの選択肢を提供します。

- arm64 — AWS Graviton2 プロセッサ用の 64 ビット ARM アーキテクチャです。
- x86_64 — x86 ベースプロセッサ用の 64 ビット x86 アーキテクチャです。

> **Note**
>
> arm64 アーキテクチャは、ほとんどの AWS リージョン にあります。詳細については、AWS Lambda 料金表を参照してください。メモリの料金表で、[Arm 料金] タブを選択し、[リージョン] ドロップダウンリストを開いて、どの AWS リージョン が Lambda での arm64 の使用をサポートしているかを確認してください。
> arm64 アーキテクチャで関数を作成する方法の例については、「AWS Graviton2 プロセッサを搭載した AWS Lambda 関数」を参照してください。

トピック

arm64 アーキテクチャを使用する利点

arm64 アーキテクチャ (AWS Graviton2 プロセッサ) を使用するLambda 関数は、x86_64 アーキテクチャで実行される同等の関数よりも大幅に優れた料金とパフォーマンスを実現します。高性能コンピューティング、ビデオエンコーディング、シミュレーションワークロードなど、コンピュータ集約型のアプリケーションに arm64 を使用することを検討してください。

Graviton2 CPU は Neoverse N1 コアを使用し、Armv8.2 (CRCおよび暗号拡張機能を含む) に加えて、複数の他のアーキテクチャ拡張機能をサポートします。

Graviton2 は vCPU あたりでより大きな L2 キャッシュを提供することにより、メモリの読み取り時間を短縮します。これにより、ウェブおよびモバイルバックエンド、マイクロサービス、データ処理システムのレイテンシーパフォーマンスが向上します。Graviton2 は暗号化パフォーマンスも改善し、CPU ベースの機械学習推論のレイテンシーを改善する命令セットをサポートします。

AWS Graviton2 の詳細については、AWS Graviton プロセッサを参照してください。

arm64 アーキテクチャへの移行の要件

arm64 アーキテクチャに移行する Lambda 関数を選択する際は、スムーズな移行を確保するために、関数が次の要件を満たしていることを確認してください。

- この関数は現在、Lambda Amazon Linux 2 ランタイムを使用しています。
- デプロイパッケージには、ユーザーが管理するオープンソースコンポーネントとソースコードのみが含まれており、移行に必要な更新を行うことができます。
- 関数コードにサードパーティーの依存関係が含まれている場合、各ライブラリまたはパッケージは arm64 バージョンを提供します。

arm64 アーキテクチャとの関数コードの互換性

Lambda 関数コードは、関数の命令セットアーキテクチャと互換性がある必要があります。関数を arm64 アーキテクチャに移行する前に、現在の関数コードに関する次の点に注意してください。

- 組み込みコードエディタを使用して関数コードを追加した場合、コードはどちらのアーキテクチャでも変更なしで実行される可能性があります。
- 関数コードをアップロードした場合は、ターゲットアーキテクチャと互換性のある新しいコードをアップロードする必要があります。
- 関数でレイヤーを使用する場合は、各レイヤーをチェックして新しいアーキテクチャと互換性があることを確認する必要があります。レイヤーに互換性がない場合は、関数を編集して、現在のレイヤーバージョンを互換性のあるレイヤーバージョンに置き換えます。
- 関数で Lambda 拡張機能を使用する場合は、各拡張機能をチェックして、新しいアーキテクチャと互換性があることを確認する必要があります。
- 関数でコンテナイメージデプロイパッケージタイプを使用する場合は、関数のアーキテクチャと互換性のある新しいコンテナイメージを作成する必要があります。

arm64 アーキテクチャへの移行方法

Lambda 関数を arm64 アーキテクチャに移行するために、次のステップに従うことをお勧めします。

1. アプリケーションまたはワークロードの依存関係のリストを構築します。一般的な依存関係は次のとおりです。
   - 関数が使用するすべてのライブラリとパッケージ。
   - コンパイラー、テストスイート、継続的インテグレーション、継続的デリバリー (CI/CD) パイプライン、プロビジョニングツール、スクリプトなど、関数の構築、デプロイ、テストに使用するツール。
   - 本番環境で関数をモニタリングするために使用する Lambda 拡張機能およびサードパーティー製ツール。
2. 各依存関係についてバージョンをチェックし、次に、arm64 バージョンが使用可能かどうかをチェックします。
3. アプリケーションを移行する環境を構築します。
4. アプリケーションをブートストラップします。
5. アプリケーションをテストおよびデバッグします。
6. arm64 関数のパフォーマンスをテストします。x86_64 バージョンとパフォーマンスを比較します。
7. arm64 Lambda 関数をサポートするように、インフラストラクチャパイプラインを更新します。
8. デプロイを本番環境にステージングします。

   たとえば、エイリアスルーティング設定を使用して、関数の x86 バージョンと arm64 バージョン間でトラフィックを分割し、パフォーマンスとレイテンシーを比較します。

Java、Go、.NET、Python の言語固有の情報を含む arm64 アーキテクチャのコード環境を作成する方法の詳細については、GitHub レポジトリの「AWS Graviton の使用を開始する」を参照してください。

命令セットアーキテクチャの設定

Lambda コンソール、AWS SDK、AWS Command Line Interface (AWS CLI)、AWS CloudFormation を使用して、新規および既存の Lambda 関数の命令セットアーキテクチャを設定できます。コンソールから既存の Lambda 関数の命令セットアーキテクチャを変更するには、次の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。

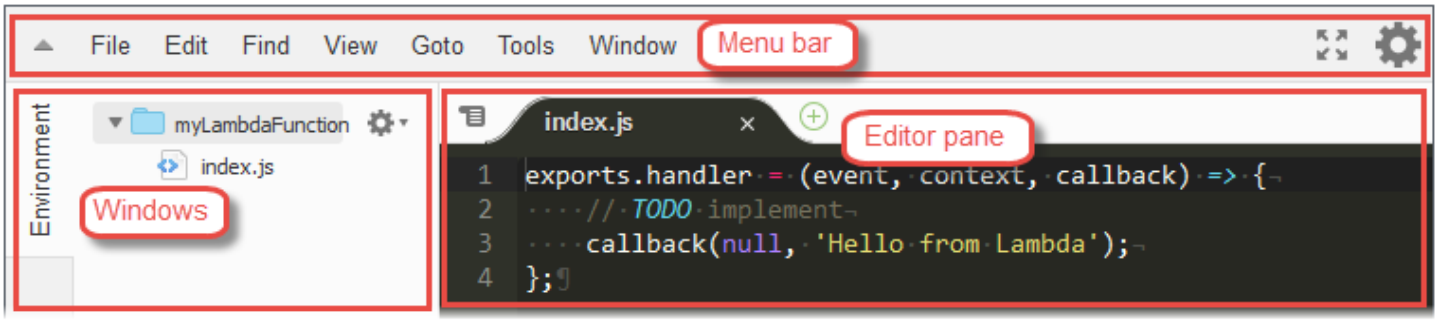
2. 命令セットアーキテクチャを設定する関数の名前を選択します。
3. メインの [コード] タブの [ランタイム設定] セクションで、[編集] を選択します。
4. [アーキテクチャ] で、関数が使用する命令セットアーキテクチャを選択します。
5. [保存] をクリックします。

> **Note**
>
> Amazon Linux 2 ランタイムは、すべて x86_64 と ARM CPU アーキテクチャの両方をサポートしています。
> Go 1.x などの Amazon Linux 2 を使用しないランタイムは、arm64 アーキテクチャをサポートしません。Go 1.x で arm64 アーキテクチャを使用するには、提供された .al2 ランタイムで関数を実行します。詳細については、.zip パッケージとコンテナイメージのデプロイ手順を参照してください。

Lambda コンソールエディタを使用したコードの編集

Lambda コンソールのコードエディタを使用すると、Lambda 関数のコードを記述し、テストして、実行結果を表示することができます。コードエディタは、Node.js や Python など、コンパイルを必要としない言語をサポートしています。コードエディタは .zip ファイルアーカイブのデプロイパッケージにのみ対応しています。また、デプロイパッケージのサイズは 3 MB 未満にする必要があります。

コードエディタには、メニューバー、ウィンドウ、およびエディタペインがあります。



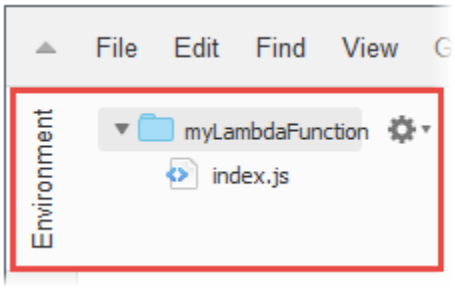
コマンドのリストについては、AWS Cloud9 ユーザーガイドの「メニューバーコマンドリファレンス」を参照してください。そのリファレンスに示しているコマンドの一部はコードエディタで使用できません。

トピック

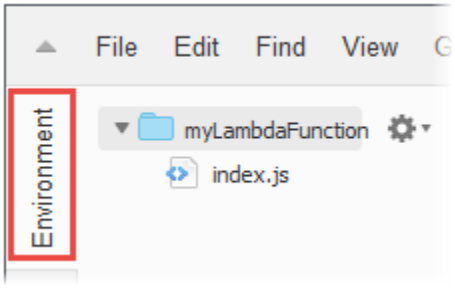
- ファイルとフォルダの操作
- コードの操作
- 全画面表示モードでの作業
- 設定の操作

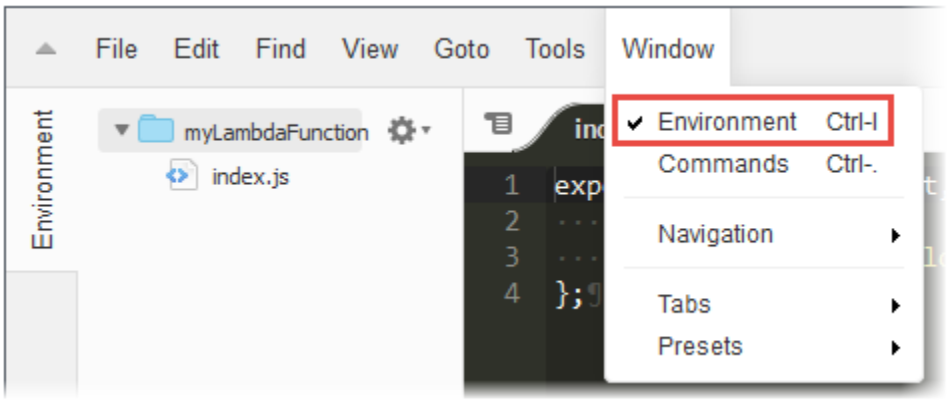
ファイルとフォルダの操作

コードエディタの [Environment] ウィンドウを使用して、関数のファイルを作成したり、開いたり、管理したりできます。



[Environment] ウィンドウを表示または非表示にするには、[Environment] ボタンを選択します。[Environment] ボタンが非表示になっている場合は、メニューバーで [Window]、[Environment] の順に選択します。





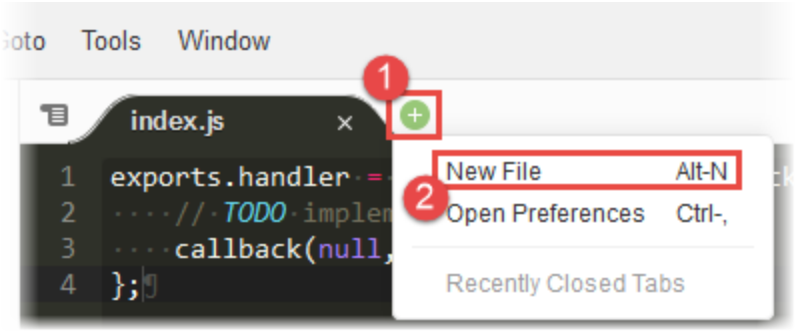
1 つのファイルを開き、その内容を エディタペインに表示するには、[Environment] ウィンドウでファイルをダブルクリックします。

複数のファイルを開き、その内容を エディタペインで表示するには、[Environment] ウィンドウでそれらのファイルを選択します。選択範囲を右クリックし、[Open] を選択します。

新しいファイルを作成するには、以下のいずれかの操作を行います。

- [Environment] ウィンドウで、新しいファイルを移動する先のフォルダを右クリックし、[New File] を選択します。ファイルの名前と拡張子を入力し、Enter キーを押します。

- メニューバーで [ファイル]、[新しいファイル] の順に選択します。ファイルを保存する準備ができたら、メニューバーで [File]、[Save] の順に選択するか、[File]、[Save As] の順に選択します。その後、[Save As] ダイアログボックスで、ファイルの名前を付け、保存場所を選択します。
- エディタペインのタブボタンバーで、[+] ボタンを選択してから、[New File] を選択します。ファイルを保存する準備ができたら、メニューバーで [File]、[Save] の順に選択するか、[File]、[Save As] の順に選択します。その後、[Save As] ダイアログボックスで、ファイルの名前を付け、保存場所を選択します。



新しフォルダを作成するには、[Environment] ウィンドウで、新しいフォルダを移動する先のフォルダを右クリックし、[New Folder] を選択します。フォルダの名前を入力し、Enter キーを押します。

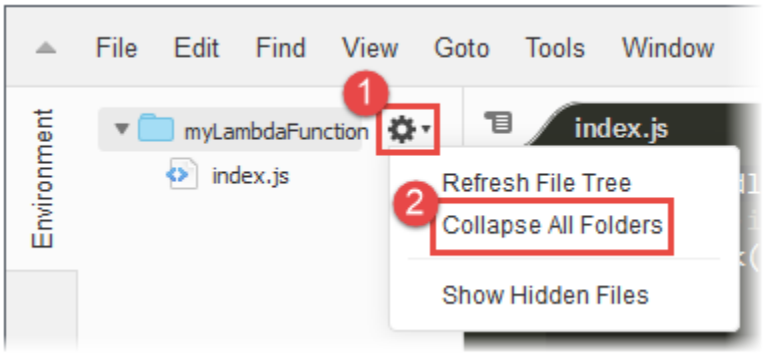
ファイルを保存するには、ファイルが開いてその内容がエディタペインに表示された状態で、メニューバーで [File]、[Save] の順に選択します。

ファイルまたはフォルダの名前を変更するには、[Environment] ウィンドウでファイルまたはフォルダを右クリックします。置換後の名前を入力し、Enter キーを押します。

ファイルまたはフォルダを削除するには、[Environment] ウィンドウでファイルまたはフォルダを選択します。選択範囲を右クリックし、[Delete] を選択します。その後、[Yes] (単一選択の場合) または [Yes to All] を選択して、削除を確定します。

ファイルまたはフォルダを切り取り、コピー、貼り付け、または複製するには、[Environment] ウィンドウでファイルまたはフォルダを選択します。選択範囲を右クリックし、[Cut] (切り取る)、[Copy] (コピー)、[Paste] (貼り付け)、[Duplicate] (重複) をそれぞれ選択します。

フォルダを折りたたむには、[Environment] ウィンドウで歯車アイコンを選択してから、[Collapse All Folders] を選択します。



隠しファイルを表示または非表示にするには、[Environment] ウィンドウで歯車アイコンを選択してから、[Show Hidden Files] を選択します。

関数に設定されている環境変数を確認するには、以下を実行してください。

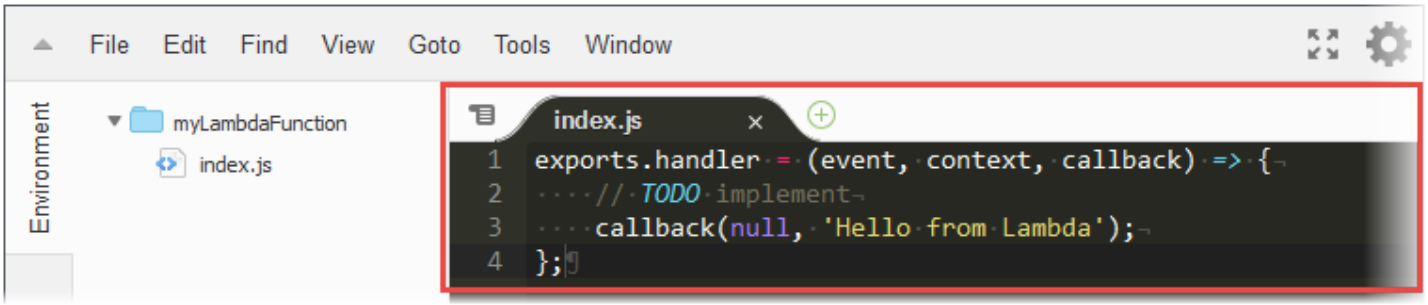
1. [コード] タブを選択します。
2. [環境変数] タブを選択します。
3. [ツール]、[環境変数を表示] の順に選択します。

環境変数は、コンソールのコードエディタに一覧表示されても暗号化されたままです。転送中の暗号化の暗号化ヘルパーを有効にした場合、それらの設定は変更されません。詳細については、「Lambda 環境変数の保護」を参照してください。

環境変数リストは読み取り専用で、Lambda コンソールでのみ使用できます。このファイルは、関数の .zip ファイルアーカイブをダウンロードしたときには含まれていません。また、このファイルをアップロードしても環境変数を追加することはできません。

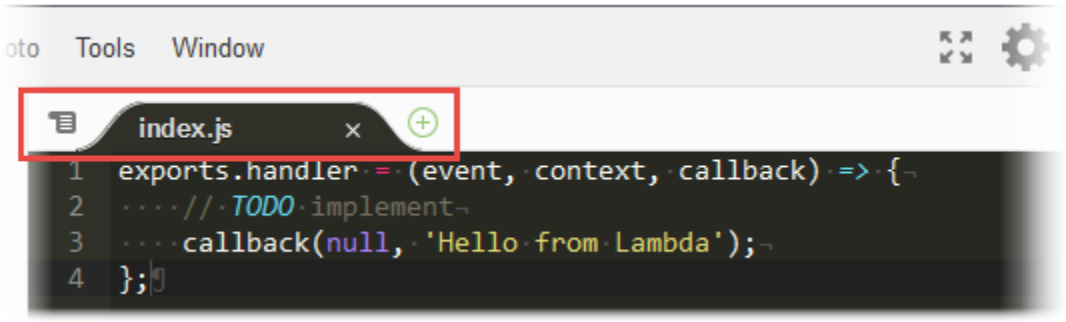
コードの操作

コードエディタのエディタペインを使用して、コードを表示したり記述したりします。



タブボタンの操作

タブボタンバーを使用して、ファイルを選択、表示、作成します。



開いているファイルの内容を表示するには、以下のいずれかの操作を行います。

- ファイルのタブを選択します。
- タブボタンバーでドロップダウンメニューボタンを選択してから、ファイルの名前を選択します。

開いているファイルを閉じるには、以下のいずれかの操作を行います。

- ファイルのタブで [X] アイコンを選択します。
- ファイルのタブを選択します。その後、タブボタンバーでドロップダウンメニューボタンを選択してから、[Close Pane] を選択します。

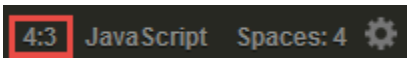
複数の開いているファイルを閉じるには、タブボタンのバーでドロップダウンメニューを選択し、必要に応じて [Close All Tabs in All Panes] または [Close All But Current Tab] を選択します。

新しいファイルを作成するには、タブボタンバーで [+] ボタンを選択してから、[New File] を選択します。ファイルを保存する準備ができたら、メニューバーで [File]、[Save] の順に選択するか、[File]、[Save As] の順に選択します。その後、[Save As] ダイアログボックスで、ファイルの名前を付け、保存場所を選択します。

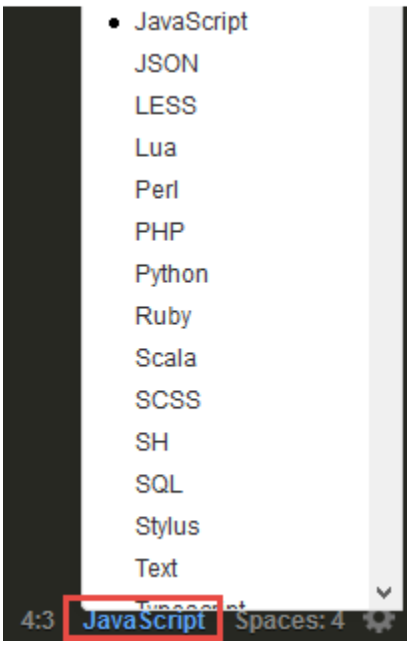
ステータスバーの操作

ステータスバーを使用して、アクティブなファイルの行にすばやく移動し、コードの表示方法を変更します。

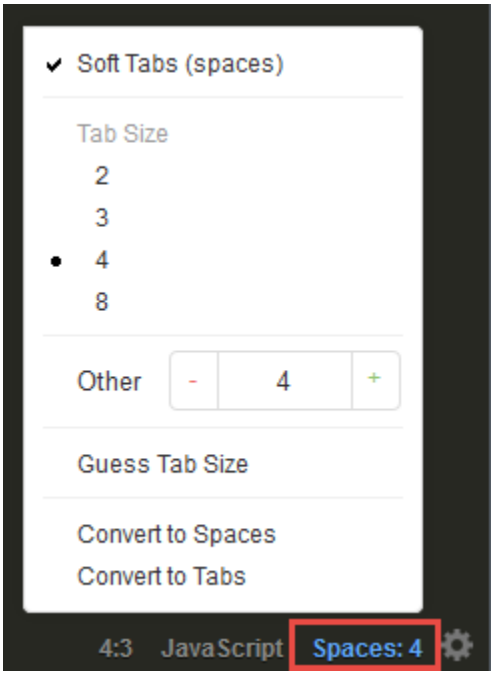
アクティブなファイルの行にすばやく移動するには、行セレクターを選択し、移動する先の行の番号を入力して、Enter キーを押します。



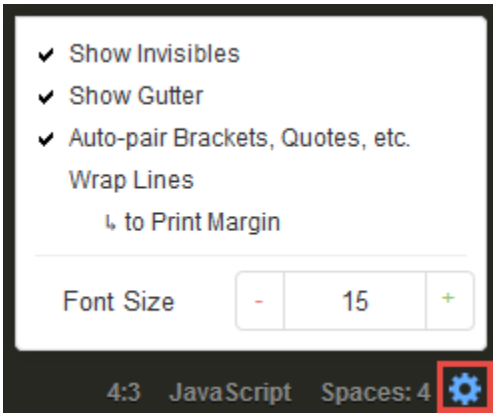
アクティブなファイルのコードカラースキームを変更するには、コードカラースキームセレクターを選択してから、新しいコードカラースキームを選択します。



アクティブなファイルで、ソフトタブを使用するかスペースを使用するかを変更したり、タブサイズを変更したり、スペースに変換するかタブに変換するかを変更したりするには、スペースおよびタブセレクターを選択してから、新しい設定を選択します。



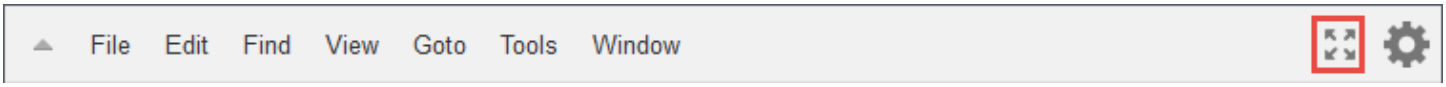
すべてのファイルで、編集上の文字や余白の表示/非表示、かっこや引用符の自動補完、行の折り返し、フォントサイズの変更を行うには、歯車アイコンを選択してから、新しい設定を選択します。



全画面表示モードでの作業

コードエディタを全画面表示にして、コードで作業するための領域を広げることができます。

コードエディタをウェブブラウザウィンドウの端まで広げるには、メニューバーの [Toggle fullscreen] ボタンを選択します。



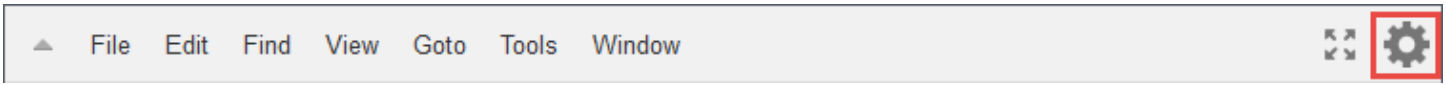
コードエディタを元のサイズに戻すには、もう一度 [Toggle fullscreen] ボタンを選択します。

全画面表示モードでは、メニューバーに [保存] と [テスト] という追加オプションが表示されます。[Save] を選択すると、関数コードが保存されます。[Test] または [Configure Events] を選択すると、関数のテストイベントを作成または編集できます。

設定の操作

表示されるコーディングのヒントや警告、コードの折りたたみ動作、コードの自動補完動作など、コードエディタのさまざまな設定を変更できます。

コードエディタの設定を変更するには、メニューバーで [Preferences] 歯車アイコンを選択します。



設定のリストについては、『AWS Cloud9 ユーザーガイド』の以下のリファレンスを参照してください。

- 変更可能なプロジェクト設定

- 使用可能なユーザー設定

それらのリファレンスで示している設定の一部はコードエディタで使用できません。

追加の Lambda 機能

Lambda では、関数の管理と呼び出しを行うためのマネジメントコンソールと APIを提供しています。また、必要に応じて言語間やフレームワーク間の切り替えを簡単にできるように、標準の機能セットをサポートするランタイムが提供されています。関数に加えて、バージョンやエイリアス、レイヤー、カスタムランタイムを作成することもできます。

高度な機能

- スケーリング
- 同時実行制御
- 関数 URL
- 非同期呼び出し
- イベントソースマッピング
- 送信先
- 関数ブループリント
- テストおよびデプロイのツール
- アプリケーションテンプレート

スケーリング

Lambda は、コードを実行するインフラストラクチャを管理し、受信リクエストに応じて自動的にスケーリングします。関数の呼び出しが速くて関数の単一のインスタンスではイベントを処理できない場合、Lambda は追加のインスタンスを実行することでスケールアップします。トラフィックが低下すると、非アクティブなインスタンスはフリーズまたは停止されます。料金は、関数の初期化中、ならびに関数がイベントを処理している時間に対してのみ発生します。

詳細については、「Lambda 関数のスケーリングについて」を参照してください。

同時実行制御

同時実行数の設定を使用して、本番稼働用アプリケーションの可用性と応答性を高めます。

関数が過度の同時実行数を使用するのを防ぎ、アカウントの使用可能な同時実行数の一部を関数に予約するには、予約された同時実行数を使用します。同時実行数のリザーブは、使用可能な同時実行のプールをサブセットに分割します。同時実行数のリザーブがある関数は、専用サブセットからの同時実行のみを使用します。

レイテンシーの変動なしに関数をスケーリングできるようにするには、プロビジョニングされた同時実行数を使用します。初期化に時間がかかる関数や、すべての呼び出しできわめて低いレイテンシーを必要とする関数の場合、プロビジョニングされた同時実行数により、関数のインスタンスを事前に初期化し、常に実行状態にしておくことができます。Lambda はApplication Auto Scaling と統合されており、使用率に基づいてプロビジョニングされた同時実行数のオートスケーリングをサポートします。

詳細については、「関数に対する予約済み同時実行数の設定」を参照してください。

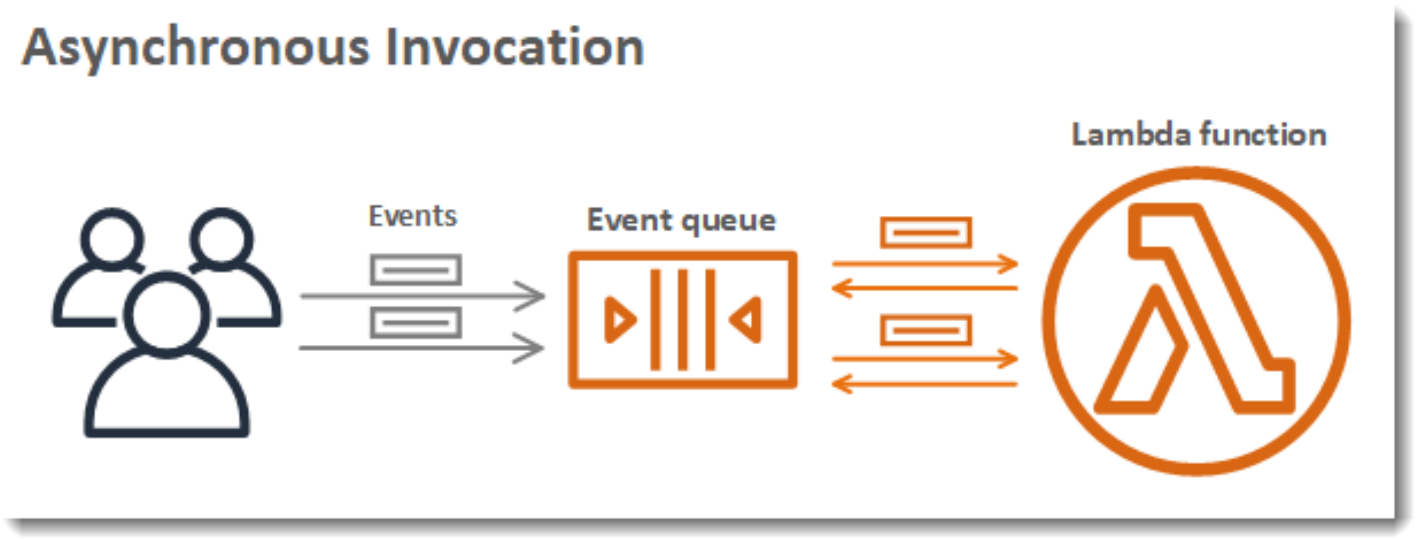
関数 URL

Lambda は、組み込まれた HTTP エンドポイントサポートを、関数 URL経由で提供します。関数 URL を使用すると、専用の HTTP エンドポイントを Lambda 関数に割り当てることができます。設定された関数 URL を使用することで、ウェブブラウザ、curl、Postman、または任意の HTTP クライアントから、関数を呼び出すことができます。

関数 URL は、既存の関数に追加することができます。あるいは、関数 URL を指定しながら新しい関数を作成します。詳細については、「Lambda 関数 URL の呼び出し」を参照してください。

非同期呼び出し

関数を呼び出す際は、同期的に呼び出すか非同期的に呼び出すかを選択できます。同期呼び出しでは、イベントを処理する関数を待ってレスポンスを返します。非同期呼び出しでは、Lambda はイベントをキューに入れて処理し、すぐにレスポンスを返します。



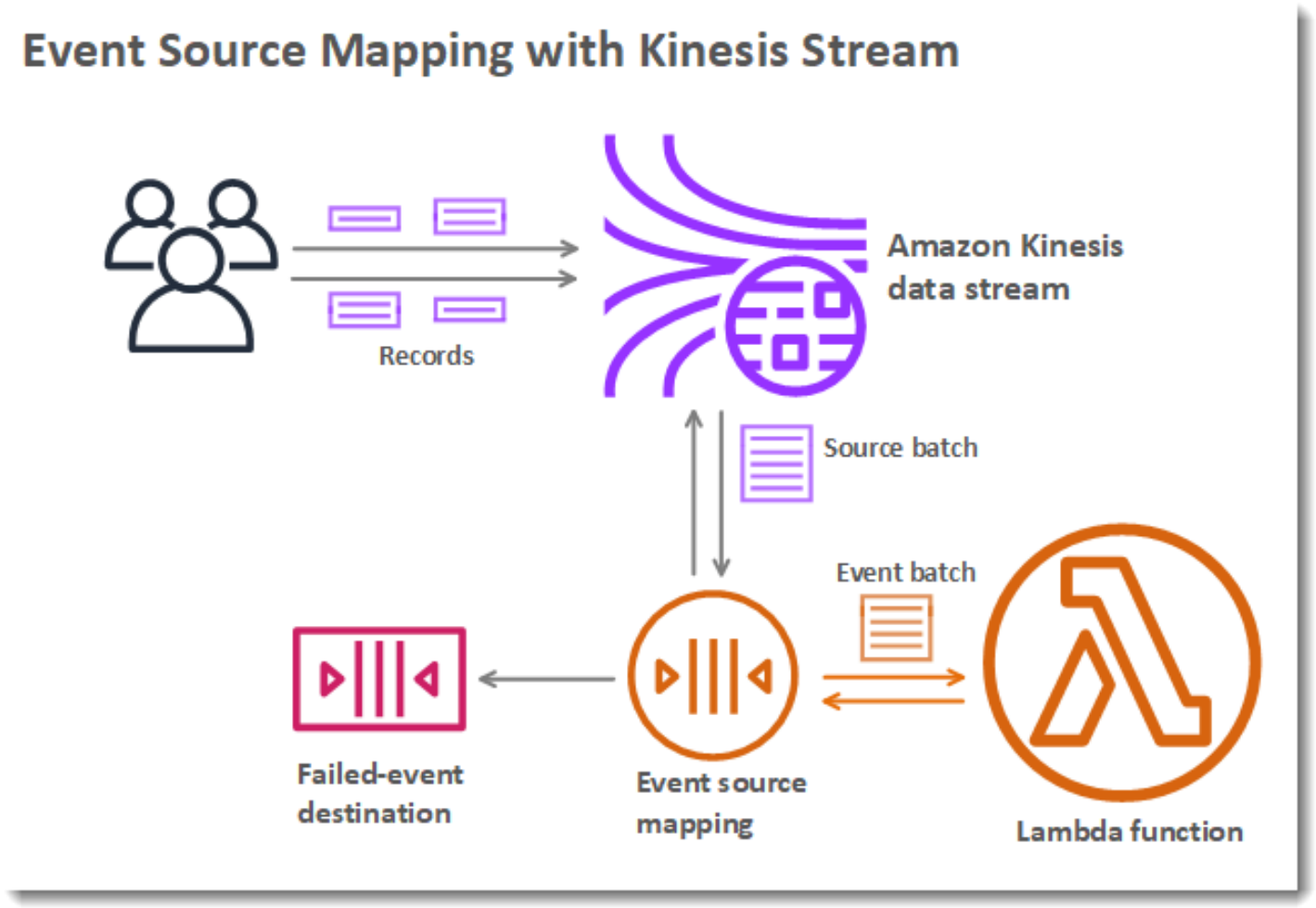
非同期呼び出しの場合、関数がエラーを返すか、スロットリングされると Lambda は再試行の処理を行います。この動作をカスタマイズするため、関数、バージョン、またはエイリアスのエラー処理

設定を定義できます。また、処理に失敗したイベントをデッドレターキューに送信したり、任意の呼び出しのレコードを送信先に送信したりするように Lambda を設定することもできます。

詳細については、「非同期呼び出し」を参照してください。

イベントソースマッピング

イベントソースマッピングを作成することで、ストリームまたはキューからの項目を処理できます。イベントソースマッピングは、Amazon Simple Queue Service (Amazon SQS) キュー、Amazon Kinesis ストリーム、または Amazon DynamoDB ストリームから項目を読み取り、バッチとして関数に送信するための Lambda のリソースです。関数が処理する各イベントには、数百または数千の項目を含めることができます。

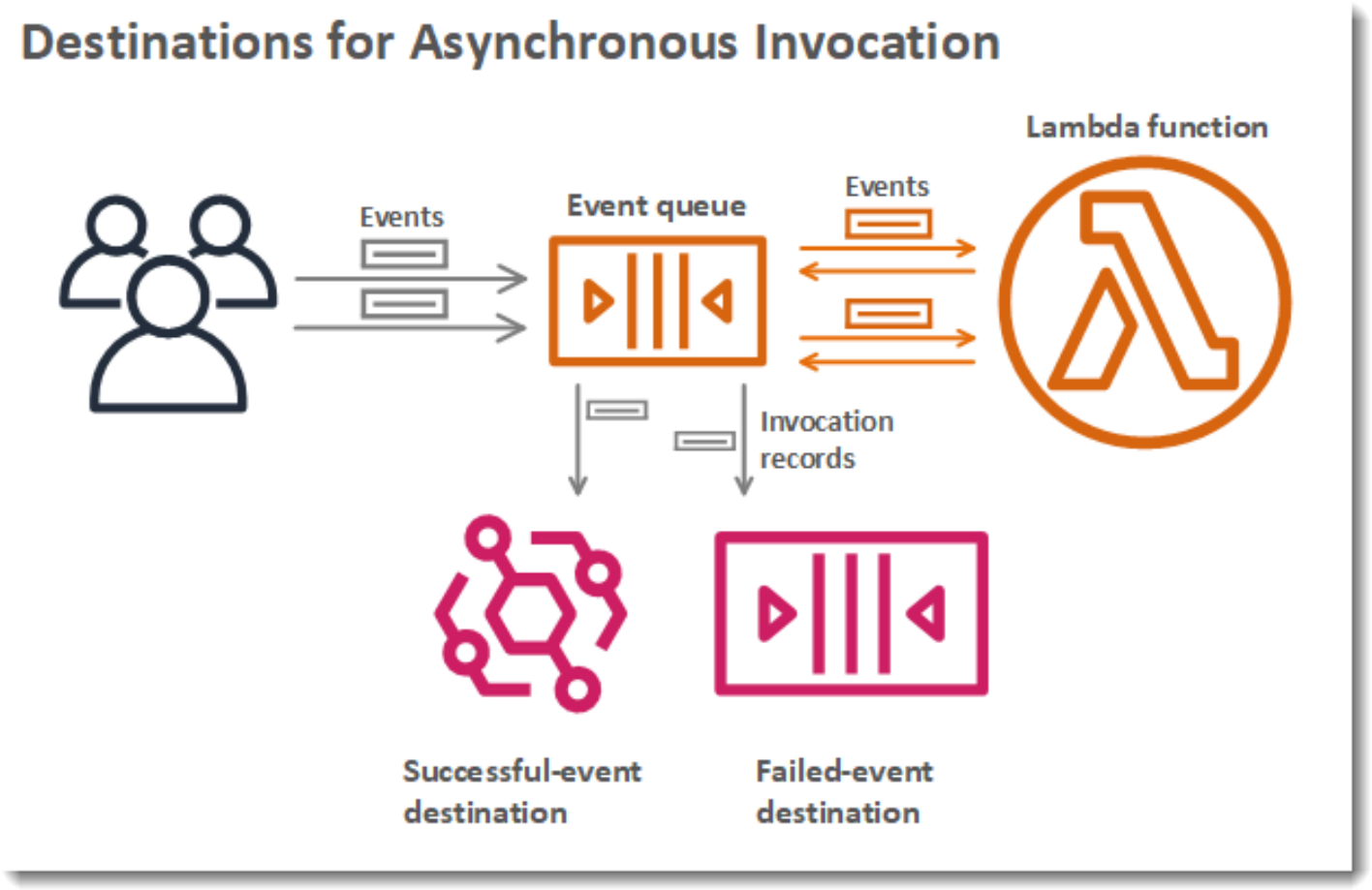


イベントソースマッピングでは、未処理項目がローカルでキューに保存されており、関数がエラーを返すかスロットリングされた場合には処理が再試行されます。イベントソースマッピングを設定することで、バッチ処理の動作とエラー処理をカスタマイズできます。あるいは、処理に失敗した項目のレコードを送信先に伝達させることも可能です。

詳細については、「Lambda がストリームおよびキューベースのイベントソースからのレコードを処理する方法」を参照してください。

送信先

送信先は、関数の呼び出しレコードを受け取る AWS リソースです。非同期呼び出しの場合、呼び出しレコードをキュー、トピック、関数、またはイベントバスに送信するように Lambda を設定できます。成功した呼び出しと処理に失敗したイベントに別々の送信先を設定できます。呼び出しレコードには、イベント、関数のレスポンス、レコードが送信された理由に関する詳細が含まれます。



ストリームから読み取るイベントソースマッピングの場合、処理に失敗したバッチのレコードをキューまたはトピックに送信するように Lambda を設定できます。イベントソースマッピングの失敗レコードには、バッチに関するメタデータが含まれており、ストリームの項目を指しています。

詳細については、「非同期呼び出しの送信先の設定」、および「Amazon DynamoDB で AWS Lambda を使用する」と「Lambda が Amazon Kinesis Data Streams からのレコードを処理する方法」でエラー処理のセクションを参照してください。

関数ブループリント

Lambda コンソールで関数を作成するときは、一から作成、設計図の使用、またはコンテナイメージの使用を選択できます。ブループリントからは、AWS のサービスや一般的なサードパーティーのアプリケーションでの Lambda の使用方法を示すサンプルコードが提供されます。ブループリントには、Node.js および Python ランタイム用のサンプルコードおよび関数設定プリセットが含まれています。

ブループリントは、Amazon Software License の下で使用するために提供されています。これらは、Lambda コンソールでのみ使用できます。

テストおよびデプロイのツール

Lambda では、コードをそのまま、またはコンテナイメージとしてデプロイできます。Lambda 関数の作成、構築、およびデプロイには、AWS のサービスと、Docker コマンドラインインターフェイス (CLI) などの一般的なコミュニティツールを使用できます。Docker CLI の設定については、Docker ドキュメントウェブサイトの「Docker の入手」を参照してください。AWS での Docker の使用方法の概要については、Amazon Elastic Container Reg ユーザーガイドユーザーガイドの「AWS CLI を使用した Amazon ECR の開始方法」を参照してください。

AWS CLI および AWS SAM CLI は、Lambda アプリケーションスタックを管理するためのコマンドラインツールです。AWS CloudFormation API でアプリケーションスタックを管理するコマンドに加え、AWS CLI はデプロイパッケージのアップロードやテンプレートの更新などのタスクを簡素化する高レベルのコマンドをサポートしています。AWS SAM CLI は、テンプレートの検証、ローカルテスト、CI/CD システムとの統合を含む追加の機能性を提供します。

- AWS SAM CLI のインストール
- AWS SAM でサーバーレスアプリケーションのテストとデバッグ
- AWS SAM で CI/CD システムを使用したサーバーレスアプリケーションのデプロイ

アプリケーションテンプレート

Lambda コンソールでは、継続的デリバリーパイプラインを使用してアプリケーションを作成できます。Lambda コンソールのアプリケーションテンプレートには、1 つ以上の関数のコード、関数とそれをサポートする AWS リソースを定義するためのアプリケーションテンプレート、AWS CodePipeline パイプラインを定義するためのインフラストラクチャテンプレートが含まれています。パイプラインには、含まれている Git リポジトリに変更をプッシュするたびに実行されるビルドとデプロイのステージがあります。

アプリケーションテンプレートは、MIT No Attribution ライセンスの下で使用するために提供されています。これらは、Lambda コンソールでのみ使用できます。

詳細については、「AWS Lambda コンソールでのアプリケーションの管理」を参照してください。

サーバーレスソリューションの構築方法について説明します

> **Tip**
>
> サーバーレスソリューションを構築する方法については、「サーバーレスデベロッパーガイド」を参照してください。

Lambda ランタイム

Lambda は、ランタイムの使用により複数の言語をサポートします。ランタイムは、Lambda と関数の間の呼び出しイベント、コンテキスト情報、レスポンスを中継する言語固有の環境を提供します。Lambda が提供するランタイムを使用することも、独自に構築することもできます。

プログラミング言語のメジャーリリースにはそれぞれ別個のランタイムがあり、固有のランタイム識別子 (`nodejs20.x` または `python3.12` など) を持っています。新しいメジャー言語バージョンを使用するように関数を設定するには、ランタイム識別子を変更する必要があります。AWS Lambda はメジャーバージョン間の下位互換性を保証できないため、これは顧客が主導権を持つ操作です。

コンテナーイメージとして定義された関数の場合は、そのコンテナーイメージを作成する際に、ランタイムと Linux ディストリビューションを選択します。ランタイムを変更するには、新しいコンテナーイメージを作成します。

デプロイパッケージに .zip ファイルアーカイブを使用する場合は、関数の作成時にランタイムを選択します。このランタイムは、関数の設定を更新することで変更できます。ランタイムは、Amazon Linux ディストリビューションの 1 つとペアを構成しています。基盤となる実行環境は、関数コードからアクセスできる追加のライブラリと環境変数を提供します。

Lambda は、実行環境で関数を呼び出します。実行環境では、関数の実行に必要なリソースを管理するセキュアで分離されたランタイム環境が提供されます。Lambda は以前の呼び出し (使用可能な場合) から実行環境を再利用しますが、新しい実行環境を作成することもできます。

Lambda で Go や Rust などの他の言語を使用するには、OS 専用ランタイムを使用してください。Lambda 実行環境には、呼び出しイベントの取得とレスポンスの送信を行うためのランタイムインターフェイスが搭載されています。カスタムランタイムを関数コードと共に実装するか、レイヤーに実装することで、他の言語をデプロイできます。

ランタイムのサポート

サポートされている Lambda ランタイムと廃止予定日リストを以下の表に示します。ランタイムが廃止された後も、一定期間中は関数の作成と更新が可能です。詳細については、「the section called “非推奨化後のランタイムの使用について”」を参照してください。この表は、現在予定されているランタイム廃止予定日を示しています。これらの日付は計画上の目的で提供されており、変更されることがあります。

ランタイムのサポート

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日
Python 3.12	`python3.12`	Amazon Linux 2023			
Python 3.11	`python3.11`	Amazon Linux 2			
「Python 3.10」	`python3.10`	Amazon Linux 2			
Python 3.9	`python3.9`	Amazon Linux 2			
Python 3.8	`python3.8`	Amazon Linux 2	2024 年 10 月 14 日	2025 年 2 月 28 日	2025 年 3 月 31 日
Java 21	`java21`	Amazon Linux 2023			
Java 17	`java17`	Amazon Linux 2			
Java 11	`java11`	Amazon Linux 2			
Java 8	`java8.al2`	Amazon Linux 2			

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
.NET 8	dotnet8	Amazon Linux 2023			
.NET 6	dotnet6	Amazon Linux 2	2024 年 11 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日
Ruby 3.3	ruby3.3	Amazon Linux 2023			
Ruby 3.2	ruby3.2	Amazon Linux 2			
OS 専用ランタイム	provided.al2023	Amazon Linux 2023			
OS 専用ランタイム	provided.al2	Amazon Linux 2			

> **Note**
>
> 新しいリージョンでは、Lambda は今後 6 か月以内に廃止される予定のランタイムをサポートしません。

Lambda は、パッチの適用とマイナーバージョンリリースのサポートにより、マネージドランタイムと対応するコンテナベースイメージを最新の状態に保ちます。詳細については、「Lambda ランタイムの更新」を参照してください。

Lambda は Go 1.x ランタイムが廃止された後も Go プログラミング言語を引き続きサポートします。詳細については、「AWS コンピュートブログ」の「Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2」を参照してください。

サポートされているすべての Lambda ランタイムは、x86_64 アーキテクチャと arm64 アーキテクチャの両方をサポートします。

新しいランタイムリリース

リリースが、言語のリリースサイクルの長期サポート (LTS) フェーズに達したときにのみ Lambda でその言語の新バージョン用マネージドランタイムが提供されます。例えば、Node.js のリリースサイクルでは、リリースが Active LTS フェーズに達したときです。

リリースは長期サポート段階に達するまでは開発段階にあり、重大な変更が加えられる可能性があります。Lambda はデフォルトでランタイム更新を自動的に適用するため、ランタイムバージョンに重大な変更があった場合、関数が期待どおりに動作しなくなる可能性があります。

Lambda は、LTS リリースが予定されていない言語バージョン用のマネージドランタイムを提供していません。

次のリストは、今後の Lambda ランタイムのリリース開始予定月を示しています。リストの日程は参考であり、変更される可能性があります。

- Python 3.13 - 2024 年 11 月
- Node.js 22 - 2024 年 11 月

ランタイムの非推奨化に関するポリシー

Lambda ランタイム .zip ファイルアーカイブは、メンテナンスとセキュリティの更新の対象となるオペレーティングシステム、プログラミング言語、およびソフトウェアライブラリの組み合わせを中心に構築されています。Lambda の標準的な非推奨化に関するポリシーは、ランタイムの主要コンポーネントのいずれかに対するコミュニティ長期サポート (LTS) 期間が満了し、セキュリティアップデートが利用できなくなった時点でランタイムを非推奨にするというものです。たいていの場合、これは言語ランタイムが対象ですが、オペレーティングシステム (OS) で LTS の満了を迎えることでランタイムが非推奨化される場合もあります。

ランタイムが非推奨になると、AWS はご使用のランタイムにセキュリティパッチを適用しなくなります。これにより、そのランタイムを使用する関数はテクニカルサポートの対象ではなくなります。このような非推奨のランタイムは、「現状のまま」の保証なしで提供されるため、バグ、エラー、欠陥、またはその他の脆弱性が含まれている可能性があります。

ランタイムのアップグレードと廃止の管理について詳しくは、以下のセクションおよび、「AWS コンピュートブログ」の「Managing AWS Lambda runtime upgrades」を参照してください。

> **Important**
>
> Lambda では、ランタイムがサポートする言語バージョンのサポート期間満了後も Lambda ランタイムの非推奨化を一定期間延長することがあります。この期間中、Lambda ではランタイム OS にのみセキュリティパッチを適用します。Lambda では、サポート期間を満了したプログラミング言語ランタイムにセキュリティパッチを適用しません。

Node.js 16 ランタイムの非推奨化

お客様からのフィードバックにお応えし、AWS は Node.js 16 ランタイムの非推奨化をコミュニティ LTS の終了から 9 か月後まで延期しています。Node.js 16 ランタイムは、「サポート対象ランタイム」の表に記載されている日付に非推奨化される予定です。前の注記で述べたように、2023 年 9 月 11 日の LTS 終了から非推奨化実施までの期間については、Lambda はランタイム OS にのみパッチを適用します。この期間中は、言語ランタイムに対してセキュリティパッチが適用されることはありません。

Node.js 16 の非推奨化を遅らせることで、このランタイムを使用するお客様は Node.js 18 をスキップし、関数を Node.js 20 に直接移行できるようになります。

責任共有モデル

Lambda は、サポートされているすべてのマネージドランタイムとコンテナベースイメージに関するセキュリティ更新のキュレーションと発行に対する責任を担います。デフォルトでは、Lambda はマネージドランタイムを使用してこれらの更新を関数に自動的に適用します。デフォルトの自動ランタイム更新設定が変更されている場合は、「責任共有モデル」を参照してください。コンテナー イメージを使用してデプロイされた関数の場合、最新のベースイメージから関数のコンテナイメージを再構築し、コンテナイメージを再デプロイする責任はお客様が担います。

ランタイムが非推奨になると、 Lambda のマネージドランタイムとコンテナベースイメージの更新はサポートされなくなります。サポートされているランタイムまたはベースイメージの使用における関数のアップグレードは、お客様の責任で行ってください。

いずれの場合も、依存関係を含め、関数コードへの更新の適用について、お客様が責任を持ちます。責任共有モデルに基づくお客様の責任は、次の表にまとめられています。

ランタイムライフサイクルフェーズ	Lambda の責任	お客様の責任
サポートされているマネージドランタイム	セキュリティパッチやその他の更新プログラムを使用して、ランタイムを定期的に更新します。 ランタイム更新をデフォルトで自動的に適用します (デフォルト以外の動作については、「the section called “ランタイム管理コントロール”」を参照してください)。	依存関係を含む関数コードを更新して、セキュリティの脆弱性に対処します。
サポートされているコンテナイメージ	セキュリティパッチやその他の更新プログラムを使用して、コンテナベースイメージを定期的に更新します。	依存関係を含む関数コードを更新して、セキュリティの脆弱性に対処します。 最新のベースイメージを使用して、コンテナイメージを定期的に再構築して再デプロイします。
マネージドランタイムが非推奨になる予定日に近づいています	ドキュメント、AWS Health Dashboard、E メール、および Trusted Advisor を使用して、ランタイムが非推奨になる前にお客様に通知します。 非推奨になると、弊社はランタイムを更新する責任を負いません。	ランタイムの非推奨情報に関する Lambda ドキュメント、AWS Health Dashboard、E メール、または Trusted Advisor をモニタリングします。 以前のランタイムが非奨励になる前に、サポートされているランタイムバージョンに関数をアップグレードします。

ランタイムライフサイクルフェーズ	Lambda の責任	お客様の責任
コンテナイメージが非推奨になる予定日に近づいています	コンテナイメージを使用する関数では、非推奨に関する通知はありません。 コンテナベースイメージの更新は、非推奨時に終了します。	イメージの非推奨に伴い、非推奨に関するスケジュールの確認と、ベースイメージのサポートされるバージョンへのアップグレードを行ってください。

非推奨化後のランタイムの使用について

ランタイムが非推奨になると、AWS はご使用のランタイムにセキュリティパッチを適用しなくなります。これにより、そのランタイムを使用する関数はテクニカルサポートの対象ではなくなります。このような非推奨のランタイムは、「現状のまま」の保証なしで提供されるため、バグ、エラー、欠陥、またはその他の脆弱性が含まれている可能性があります。非推奨のランタイムを使用する関数は、パフォーマンスが低下したり、証明書の有効期限切れなどの問題が発生したりして、正しく動作しなくなる可能性があります。

ランタイムが廃止されてから少なくとも 30 日間は、そのランタイムを使用して新しい Lambda 関数を作成できます。廃止の 30 日後から、Lambda によって新しい関数の作成がブロックされます。

ランタイムが非奨励になってから少なくとも 60 日間は、既存の関数の関数コードを更新できます。非推奨になってから 60 日経過後は、Lambda によって既存の関数の関数コードと設定の更新がブロックされます。

> **Note**
>
> 一部のランタイムでは、AWS は block-function-create および block-function-update の日付が非推奨になってから通常の 30 日および 60 日を超えて延期しています。AWS は顧客のフィードバックに応えてこの変更を行い、関数をアップグレードする時間を増やしました。the section called “ランタイムのサポート” および the section called “非推奨のランタイム” の表を参照し、ランタイムの日付を確認してください。

ランタイムが非奨励になった後も、関数を更新すると、サポートされている新しいランタイムバージョンを使用できます。60 日が経過すると非推奨のランタイムに戻すことができないため、本番環

境にランタイムの変更を適用する前に、関数が新しいランタイムバージョンで動作することをテストする必要があります。ロールバックによる安全なデプロイを有効にするには、関数のバージョンとエイリアスを使用することをお勧めします。

関数の作成と更新を継続できる正確な期間は決まっていないことに注意してください。この期間は、廃止日および AWS リージョン ごとに異なる場合があります。関数の作成と更新がブロックされる具体的な日付は、このページの最初のセクションにあるサポートされるランタイムの表に記載されています。Lambda は、この表に記載されている日付より前に関数の作成や更新のブロックを行うことはありません。

ランタイムが廃止された後も、関数を無期限に呼び出すことができます。ただし、AWS は、引き続き関数がセキュリティパッチを受け取り、テクニカルサポートの利用資格を維持するためにも、サポートされているランタイムへの関数の移行を強くお勧めします。

ランタイムの廃止通知を受け取る

ランタイムが廃止日に近づくと、AWS アカウント 内にそのランタイムを使用している関数があると、Lambda からメールアラートが送信されます。通知は AWS Health Dashboard と AWS Trusted Advisor にも表示されます。

- メール通知の受信:

  Lambda は、ランタイムが廃止される少なくとも 180 日前にメールアラートを送信します。メールには、そのランタイムを使用するすべての関数の $LATEST バージョンが記載されています。影響を受ける関数バージョンの全リストを確認するには、Trusted Advisor を使用するか、the section called “非推奨のランタイムを使用する関数の一覧” をご覧ください。

  Lambda は、AWS アカウント の主要アカウント連絡先にメール通知を送信します。アカウントのメールアドレスの表示または更新については、AWS 全般のリファレンスの「連絡先情報の更新」を参照してください。

- AWS Health Dashboard で通知を受信する:

  AWS Health Dashboard は、ランタイムが廃止される少なくとも 180 日前に通知を表示します。通知は [アカウントの状態] ページの [その他の通知] に表示されます。通知の [影響を受けるリソース] タブには、ランタイムを使用するすべての関数の $LATEST バージョンが一覧表示されます。

> **Note**
>
> 影響を受ける関数バージョンの、最新の全リストを確認するには、Trusted Advisor を使用するか、the section called “非推奨のランタイムを使用する関数の一覧” をご覧ください。

AWS Health Dashboard 通知は、影響を受けるランタイムが廃止されてから 90 日後に表示されなくなります。

- AWS Trusted Advisor を使用する

  Trusted Advisor は、ランタイムが廃止される 180 日前に通知を表示します。通知は [セキュリティ] ページに表示されます。影響を受ける関数のリストは、[非推奨のランタイムを使用する AWS Lambda 関数] に表示されます。この関数リストには $LATEST バージョンと公開済みバージョンの両方が表示され、関数の現在の状態を反映して自動的に更新されます。

  Trusted Advisor コンソールの [設定] ページから、Trusted Advisor の毎週のメール通知を有効にできます。

非推奨のランタイムを使用する関数の一覧

Trusted Advisor を使用することで、予定されているランタイム廃止の影響を受ける関数のライブリストを確認できます。さらに、AWS Command Line Interface (AWS CLI) またはいずれかの AWS SDK を使用することで、特定のランタイムを使用するすべての関数バージョンをリストすることもできます。

AWS CLI を使用してこのリストを生成するには、次のコマンドを実行します。`RUNTIME_IDENTIFIER` を廃止予定のランタイムの名前に置き換えて、お使いの AWS リージョン を選択してください。関数の $LATEST バージョンだけを一覧表示するには、`--function-version ALL` をコマンドから省略します。

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> コマンド例では、特定の AWS アカウント の us-east-1 リージョン内の関数を一覧表示します。このコマンドを、アカウントが機能している各リージョンと各 AWS アカウント で繰り返す必要があります。

AWS SDK で ListFunctions アクションを使用して関数をリストする方法の詳細については、目的のプログラミング言語の SDK ドキュメントを参照してください。また、いずれかの AWS SDK を使用して、DescribeLogStreams および GetMetricStatistics API アクションを使用して、最も多く呼び出された関数と最近呼び出された関数に関する統計を収集することもできます。

AWS Config の高度なクエリ機能を使用して、影響を受けるランタイムを使用するすべての関数を一覧表示することもできます。このクエリは関数の $LATEST バージョンのみを返しますが、1 つのコマンドでクエリを集約して、すべてのリージョンの複数の AWS アカウント にある関数を一覧表示できます。詳細については、「AWS Config デベロッパーガイド」の「Querying the Current Configuration State of AWS Auto Scaling Resources」を参照してください。

非推奨のランタイム

次のランタイムは、サポート終了に達しています。

非推奨のランタイム

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
.NET 7 (コンテナのみ)	dotnet7	Amazon Linux 2	2024 年 5 月 14 日		
Java 8	java8	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日
Go 1.x	go1.x	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日
OS 専用ランタイム	provided	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2025 年 2 月 28 日

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Ruby 2.7	ruby2.7	Amazon Linux 2	2023 年 12 月 7 日	2024 年 1 月 9 日	2025 年 2 月 28 日
Node.js 14	nodejs14.x	Amazon Linux 2	2023 年 12 月 4 日	2024 年 1 月 9 日	2025 年 2 月 28 日
「Python 3.7」	python3.7	Amazon Linux	2023 年 12 月 4 日	2024 年 1 月 9 日	2025 年 2 月 28 日
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	2023 年 4 月 3 日	2023 年 4 月 3 日	2023 年 5 月 3 日
Node.js 12	nodejs12.x	Amazon Linux 2	2023 年 3 月 31 日	2023 年 3 月 31 日	2023 年 4 月 30 日
Python 3.6	python3.6	Amazon Linux	2022 年 7 月 18 日	2022 年 7 月 18 日	2022 年 8 月 29 日
.NET 5 (コンテナのみ)	dotnet5.0	Amazon Linux 2	2022 年 5 月 10 日		
.NET Core 2.1	dotnetcore2.1	Amazon Linux	2022 年 1 月 5 日	2022 年 1 月 5 日	2022 年 4 月 13 日
Node.js 10	nodejs10.x	Amazon Linux 2	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 2 月 14 日
Ruby 2.5	ruby2.5	Amazon Linux	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 3 月 31 日
Python 2.7	python2.7	Amazon Linux	2021 年 7 月 15 日	2021 年 7 月 15 日	2022 年 5 月 30 日
Node.js 8.10	nodejs8.10	Amazon Linux	2020 年 3 月 6 日		2020 年 3 月 6 日

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Node.js 4.3	nodejs4.3	Amazon Linux	2020 年 3 月 5 日		2020 年 3 月 5 日
Node.js 4.3 Edge	nodejs4.3-edge	Amazon Linux	2020 年 3 月 5 日		2019 年 4 月 30 日
Node.js 6.10	nodejs6.10	Amazon Linux	2019 年 8 月 12 日	2019 年 8 月 12 日	
.NET Core 1.0	dotnetcore1.0	Amazon Linux	2019 年 6 月 27 日		2019 年 7 月 30 日
.NET Core 2.0	dotnetcore2.0	Amazon Linux	2019 年 5 月 30 日		2019 年 5 月 30 日
Node.js 0.10	nodejs	Amazon Linux			2016 年 10 月 31 日

ほとんどの場合、言語バージョンまたはオペレーティングシステムのサポート終了日は事前に通知されます。以下のリンク先には、Lambda がマネージドランタイムとしてサポートする各言語のサポート終了スケジュールが記載されています。

言語およびフレームワークのサポートポリシー

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com と Corretto のよくある質問
- Go – golang.org
- .NET — dotnet.microsoft.com

Lambda ランタイム更新

Lambda は、セキュリティ更新、バグ修正、新機能、パフォーマンス強化、およびマイナーバージョンリリースのサポートを使用して、各マネージドランタイムを最新の状態に保ちます。これらのランタイム更新は、ランタイムバージョンとして発行されます Lambda は、関数を以前のランタイムバージョンから新しいランタイムバージョンに移行することによって、ランタイム更新を関数に適用します。

マネージドランタイムを使用する関数の場合、Lambda はデフォルトでランタイム更新を自動的に適用します。自動ランタイム更新により、Lambda はランタイムバージョンにパッチを適用する運用上の負担を取り除きます。大抵のお客様には、自動更新が最適な選択肢です。詳細については、「ランタイム管理コントロール」を参照してください。

Lambda は、新しい各ランタイムバージョンのコンテナイメージとしての発行も行います。コンテナベースの関数のランタイムバージョンを更新するには、更新されたベースイメージから新しいコンテナイメージを作成して、関数を再デプロイする必要があります。

各ランタイムバージョンには、バージョン番号と ARN (Amazon リソースネーム) が関連付けられています。ランタイムのバージョン番号は Lambda が定義する番号付けスキームを使用しており、これはプログラミング言語が使用するバージョン番号とは無関係の番号です。ランタイムバージョンの ARN は、各ランタイムバージョンの一意の識別子です。

関数の現在のランタイムバージョンの ARN は、関数ログの `INIT_START` 行と、Lambda コンソールで確認できます。

ランタイムバージョンとランタイム識別子は、それぞれ別個に考える必要があります。各ランタイムには、`python3.9` や `nodejs18.x` などの一意のランタイム識別子があります。これらは主要プログラミング言語の各リリースに対応しています。ランタイムバージョンは、個々のランタイムのパッチバージョンを表しています。

> **Note**
>
> 同じランタイムバージョン番号の ARN は、AWS リージョンと CPU アーキテクチャによって異なる場合があります。

トピック

- ランタイム管理コントロール

- 2 フェーズのランタイムバージョンロールアウト
- ランタイムバージョンのロールバック
- ランタイムバージョン変更の特定
- ランタイム管理を設定する
- 責任共有モデル
- 高コンプライアンスアプリケーション

ランタイム管理コントロール

Lambda は、既存の関数との後方互換性を備えたランタイム更新を提供するように努めていますが、ソフトウェアパッチと同様に、ランタイム更新が既存の関数に悪影響を及ぼす状況がまれに発生します。例えば、セキュリティパッチは、以前のセキュアではない動作に依存する既存の関数に関する内在的な問題を明らかにする可能性があります。Lambda ランタイム管理コントロールは、ランタイムバージョンの非互換性というまれな状況で、ワークロードに影響が及ぶリスクを軽減するために役立ちます。関数バージョン ($LATEST または発行済みバージョン) ごとに、以下のランタイム更新モードのいずれかを選択できます。

- 自動 (デフォルト) – 2 フェーズのランタイムバージョンロールアウト を使用して、最新のセキュアなランタイムバージョンに自動的に更新します。ランタイム更新のメリットを常に得るためにも、これは大半のお客様に推奨されるモードです。
- 関数の更新 – 関数を更新するときに、最新の安全なランタイムバージョンに更新します。関数が更新されると、Lambda が関数のランタイムを最新のセキュアなランタイムバージョンに更新します。このアプローチは、ランタイム更新を関数デプロイと同期させることから、Lambda がランタイム更新を適用するタイミングを制御できます。このモードを使用することで、まれに発生するランタイム更新の非互換性を早期に検出して緩和することができます。このモードを使用するときは、関数を定期的に更新して、それらのランタイムを最新の状態に保つ必要があります。
- 手動 – ランタイムバージョンを手動で更新します。関数設定でランタイムバージョンを指定します。関数は、このランタイムバージョンを恒久的に使用します。新しいランタイムバージョンに既存の関数との互換性がないというまれな状況でも、このモードを使用して、関数を以前のランタイムバージョンにロールバックすることができます。デプロイ間でのランタイムの整合性を実現するためにも、[Manual] (手動) モードは使用しないことをお勧めします。詳細については、「ランタイムバージョンのロールバック」を参照してください。

ランタイム更新を関数に適用する責任は、選択するランタイム更新モードに応じて異なります。詳細については、「責任共有モデル」を参照してください。

2 フェーズのランタイムバージョンロールアウト

Lambda は、次の順序で新しいランタイムバージョンを導入します。

1. 第 1 フェーズでは、関数が作成または更新されるたびに Lambda が新しいランタイムバージョンを適用します。関数は、 UpdateFunctionCodeまたは UpdateFunctionConfiguration API オペレーションを呼び出すと更新されます。
2. 第 2 フェーズでは、[Auto] (自動) ランタイム更新モードを使用する関数で、まだ新しいランタイムバージョンに更新されていない関数を Lambda が更新します。

このロールアウトプロセスの全体的な所要時間は、ランタイム更新に含まれるセキュリティパッチの重大度などの複数の要因に応じて異なります。

関数の開発とデプロイを積極的に行っている場合は、第 1 フェーズ中に新しいランタイムバージョンを取得する可能性が高くなります。これは、ランタイム更新を関数の更新に同期します。最新のランタイムバージョンがアプリケーションに悪影響を及ぼすというまれな状況でも、このアプローチによって迅速に是正措置を講じることができます。積極的な開発が行われていない関数でも、第 2 フェーズ中の自動ランタイム更新による運用上のメリットが得られます。

このアプローチは、[Function update] (関数の更新) または [Manual] (手動) モードに設定された関数には影響しません。[Function update] (関数の更新) モードを使用する関数が最新のランタイム更新を受け取るのは、それらの作成時または更新時のみです。[Manual (手動) モードを使用する関数は、ランタイム更新を受け取りません。

Lambda は、新しいランタイムバージョンを段階的なローリング形式で AWS リージョン全体に発行します。関数が [Auto] (自動) または [Function update] (関数の更新) モードに設定されている場合、異なるリージョンに同時にデプロイされた関数、または同じリージョン内で異なる時間にデプロイされた関数が、異なるランタイムバージョンを取得する可能性があります。環境全体でランタイムバージョンの整合性を確保する必要があるお客様は、コンテナイメージを使用して Lambda 関数をデプロイする必要があります。[手動] モードは、ランタイムバージョンが関数と互換性がないというまれなイベントで、ランタイムバージョンのロールバックを可能にするための一時的な緩和策として設計されています。

ランタイムバージョンのロールバック

新しいランタイムバージョンに既存の関数との互換性がないというまれな状況が発生した場合は、関数のランタイムバージョンを以前のバージョンにロールバックすることができます。そうするこ

とで、アプリケーションが引き続き機能し、中断が最小限に抑えられるので、最新のランタイムバージョンに戻る前に非互換性を修正する時間を確保できます。

Lambda は、特定のランタイムバージョンを使用できる期間を制限していませんが、最新のセキュリティパッチ、パフォーマンスの向上、および機能面でのメリットを得るためにも、可能な限り早急に最新のランタイムバージョンに更新することを強くお勧めします。Lambda は以前のランタイムバージョンにロールバックするオプションを提供してますが、これはあくまでも、まれに発生するランタイム更新の互換性問題の一時的な緩和策として使用するものです。以前のランタイムバージョンを長期間使用している関数では、最終的にパフォーマンスの低下、または証明書の有効期限切れなどの問題が発生するので、適切に動作しなくなる可能性があります。

ランタイムバージョンは、以下の方法でロールバックできます。

- [Manual] (手動) ランタイム更新モードの使用
- 発行済みの関数バージョンの使用

詳細については、AWS コンピューティングブログの「AWS Lambda ランタイム管理コントロールの紹介」を参照してください。

[Manual] (手動) ランタイム更新モードを使用したランタイムバージョンのロールバック

[Auto] (自動) ランタイムバージョン更新モードを使用している場合、または $LATEST ランタイムバージョンを使用している場合は、[Manual] (手動)モードを使用してランタイムバージョンをロールバックできます。ロールバックする関数バージョンで、ランタイムバージョン更新モードを [Manual] (手動) に変更し、以前のランタイムバージョンの ARN を指定します。以前のランタイムバージョンの ARN を確認する方法の詳細については、「ランタイムバージョン変更の特定」を参照してください。

> **Note**
>
> 関数の $LATEST バージョンが [Manual] (手動) モードを使用するように設定されている場合は、関数が使用する CPU アーキテクチャまたはランタイムバージョンを変更できません。これらの変更を行うには、[Auto] (自動) モードまたは [Function update] (関数の更新) モードに変更する必要があります。

発行済みの関数バージョンを使用したランタイムバージョンのロールバック

発行済みの関数バージョンは、関数が作成されたときの $LATEST 関数コードと設定のイミュータブルなスナップショットです。[Auto] (自動) モードでは、ランタイムバージョンロールアウトの第 2 フェーズ中に、Lambda が発行済みの関数バージョンのランタイムバージョンを自動的に更新します。[Function update] (関数の更新) モードでは、Lambda は発行済みの関数バージョンのランタイムバージョンを更新しません。

そのため、[Function update] (関数の更新) モードを使用する発行済みの関数バージョンは、関数コード、設定、およびランタイムバージョンの静的スナップショットを作成します。関数バージョンで [Function update] (関数の更新) モードを使用することによって、ランタイム更新をデプロイと同期できます。また、トラフィックを以前の発行済み関数バージョンにリダイレクトすることで、コード、設定、およびランタイムバージョンのロールバックを調整することもできます。このアプローチは、ランタイム更新に互換性がないというまれな状況での完全に自動化されたロールバックのために、継続的インテグレーションおよび継続的デリバリー (CI/CD) に統合することができます。このアプローチを使用するときは、関数を定期的に更新し、新しい関数バージョンを発行して、最新のランタイム更新を取得する必要があります。詳細については、「責任共有モデル」を参照してください。

ランタイムバージョン変更の特定

ランタイムバージョン番号と ARN はINIT_START ログ行に記録され、Lambda は新しい実行環境 を作成するたびに CloudWatch ログに出力します。実行環境はすべての関数呼び出しに同じランタイムバージョンを使用するため、Lambda は、Lambda が init フェーズを実行するときにのみ INIT_START ログ行を出力します。Lambda は、関数の呼び出しごとにこのログ行を出力しません。Lambda はログ行を CloudWatch Logs に出力しますが、コンソールには表示されません。

Example INIT_START ログ行の例

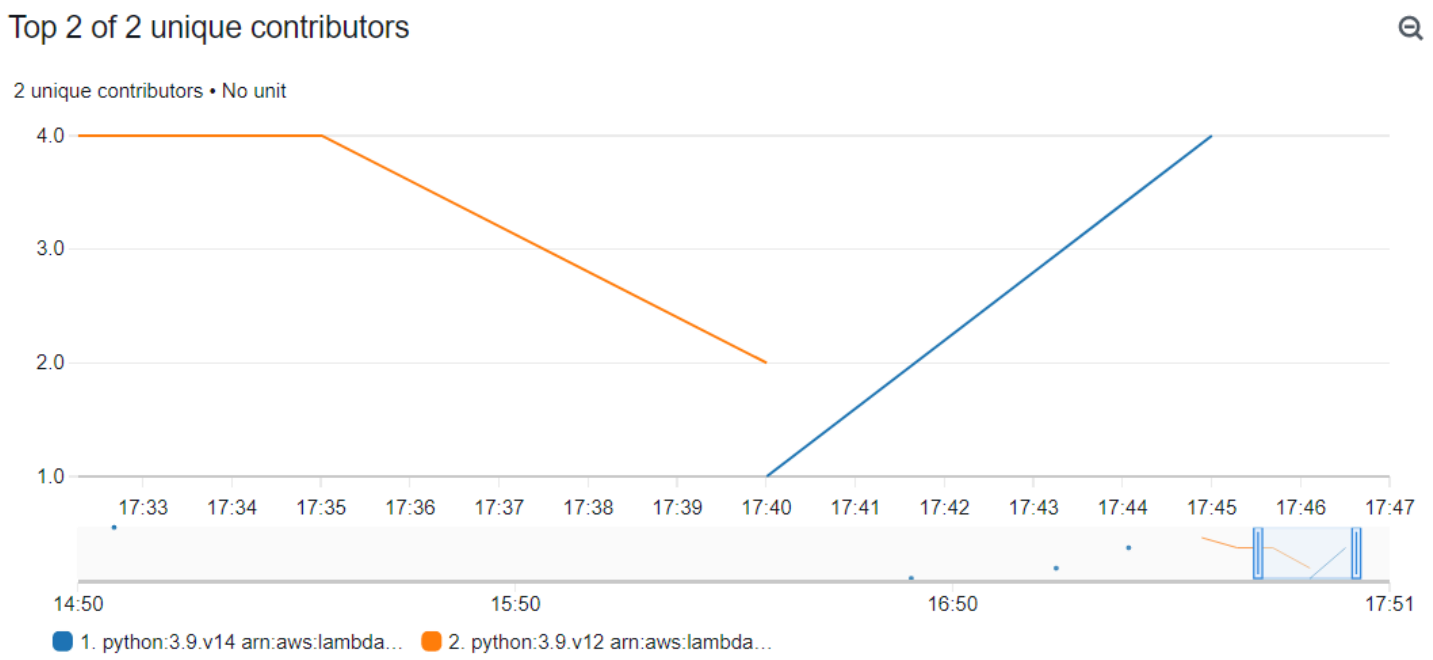
```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

ログを直接操作するのではなく、Amazon CloudWatch Contributor Insights を使用してランタイムバージョン間の移行を特定できます。以下のルールは、各 INIT_START ログ行からの独特のランタイムバージョンの個数を数えます。このルールを使用するには、サンプルロググループ名 /aws/lambda/* を、関数または一連の関数の適切なプレフィックスに置き換えてください。

```
{
  "Schema": {
```

```
    "Name": "CloudWatchLogRule",
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

次の CloudWatch Contributor Insights レポートは、前のルールでキャプチャされたランタイムバージョンの移行の例を示しています。オレンジ色の線は以前のランタイムバージョン (python:3.9.v12) の実行環境の初期化を示し、青い線は新しいランタイムバージョン (python:3.9.v14) の実行環境の初期化を示しています。



ランタイム管理を設定する

ランタイム管理は、Lambda コンソール、または AWS Command Line Interface (AWS CLI) を使用して設定できます。

> **Note**
>
> ランタイム管理は、関数バージョンごとに個別に設定できます。

Lambda がランタイムバージョンを更新する方法を設定する (コンソール)

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Code] (コード) タブの [Runtime settings] (ランタイム設定) で [Edit runtime management configuration] (ランタイム管理設定を編集) を選択します。
4. [Runtime management configuration] (ランタイム管理設定) で、以下のいずれかを選択します。
   - 関数を最新のランタイムバージョンに自動的に更新するには、[Auto] (自動) を選択します。
   - 関数を変更するときに関数を最新のランタイムバージョンに更新するには、[Function update] (関数の更新) を選択します。

- ランタイムバージョンの ARN を変更するときにのみ関数を最新のランタイムバージョンに更新するには、[Manual (手動) を選択します。

> **Note**
>
> ランタイムバージョンの ARN は、[Runtime management configuration] (ランタイム管理設定) で確認できます。ARN は、関数ログの `INIT_START` 行でも確認できます。

5. [保存] を選択します。

Lambda がランタイムバージョンを更新する方法を設定する (AWS CLI)

関数のランタイム管理を設定するには、`put-runtime-management-config` AWS CLI コマンドをランタイム更新モードとともに使用できます。`Manual` モードを使用するときは、ランタイムバージョンの ARN も指定する必要があります。

```
aws lambda put-runtime-management-config --function-name arn:aws:lambda:eu-west-1:069549076217:function:myfunction --update-runtime-on Manual --runtime-version-arn arn:aws:lambda:eu-west-1::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

次のような出力が表示されます:

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:eu-west-1:069549076217:function:myfunction",
  "RuntimeVersionArn": "arn:aws:lambda:eu-west-1::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

責任共有モデル

Lambda は、サポートされているすべてのマネージドランタイムとコンテナイメージに関するセキュリティ更新のキュレーションと発行に対する責任を担います。最新のランタイムバージョンを使用するように既存の関数を更新する責任は、お客様が使用するランタイム更新モードに応じて異なります。

Lambda は、[Auto] (自動) ランタイム更新モードを使用するように設定されたすべての関数にランタイム更新を適用する責任を担います。

[Function update] (関数の更新) ランタイム更新モードで設定された関数については、お客様が関数を定期的に更新する責任を担います。Lambda は、お客様がこれらの更新を行うときにランタイム更新を適用する責任を担います。お客様が関数を更新しない場合、Lambda はランタイムを更新しません。関数を定期的に更新しない場合は、関数を自動的なランタイム更新に設定して、継続的にセキュリティ更新を受け取るようにすることを強くお勧めします。

[Manual] (手動) ランタイム更新モードを使用するように設定された関数については、お客様が関数を更新して、最新のランタイムバージョンが使用されるようにする責任を担います。このモードは、ランタイム更新に互換性がないというまれな状況での一時的な緩和策として、ランタイムバージョンをロールバックするためのみに使用することを強くお勧めします。また、関数にパッチが適用されていない時間を最小限に抑えるため、可能な限り早急に [Auto] (自動) モードに変更することもお勧めします。

お客様が関数のデプロイにコンテナイメージを使用している場合は、Lambda が更新されたベースイメージを発行する責任を担います。この場合、最新のベースイメージから関数のコンテナイメージを再構築し、コンテナイメージを再デプロイする責任はお客様が担います。

これらは、以下の表に要約されています。

デプロイモード	Lambda の責任	お客様の責任
マネージドランタイム、[Auto] (自動) モード	最新のパッチが含まれる新しいランタイムバージョンを発行します。 既存の関数にランタイムパッチを適用します。	ランタイム更新の互換性問題が発生したまれな状況において、以前のランタイムバージョンにロールバックします。
マネージドランタイム、[Function update] (関数の更新) モード	最新のパッチが含まれる新しいランタイムバージョンを発行します。	関数を定期的に更新して、最新のランタイムバージョンを取得します。 関数を定期的に更新していないときは、関数を [Auto] (自動) モードに切り替えます。

デプロイモード	Lambda の責任	お客様の責任
		ランタイム更新の互換性問題が発生したまれな状況において、以前のランタイムバージョンにロールバックします。
マネージドランタイム、[Manual] (手動) モード	最新のパッチが含まれる新しいランタイムバージョンを発行します。	このモードは、ランタイム更新の互換性問題が発生したまれな状況における、一時的なランタイムロールバックのみに使用します。 可能な限り早急に、関数を [Auto] (自動) または [Function update] (関数の更新) モード、および最新のランタイムバージョンに切り替えます。
コンテナイメージ	最新のパッチが含まれる新しいコンテナイメージを発行します。	最新のコンテナベースイメージを使用して定期的に関数を再デプロイし、最新のパッチを取得します。

AWS との責任共有の詳細については、AWS クラウド セキュリティサイトの「責任共有モデル」を参照してください。

高コンプライアンスアプリケーション

Lambda のお客様は通常、パッチ適用要件を満たすために自動ランタイム更新を利用しています。アプリケーションが厳しいパッチ最新性要件の対象になっている場合は、以前のランタイムバージョンの使用を制限することが推奨されます。AWS Identity and Access Management (IAM) を使用してAWSアカウントのユーザーによる PutRuntimeManagementConfig API オペレーションへのアクセスを拒否することで、Lambda のランタイム管理コントロールを制限できます。この操作は、関数のランタイム更新モードを選択するために使用されます。この操作へのアクセスを拒否すると、すべての関数がデフォルトで [Auto] (自動) モードになります。この制限は、サービスコントロールポリシー (SCP) を使用することで組織全体に適用できます。関数を以前のランタイムバージョンにロールバックする必要がある場合は、ポリシー例外を case-by-case ベースで付与できます。

ランタイム環境の変更

内部拡張機能を使用して、ランタイムプロセスを変更できます。内部拡張機能は、個別のプロセスではありません。ランタイムプロセスの一部として実行されます。

Lambda は、ランタイムにオプションとツールを追加するように設定することが可能な、言語固有の環境変数を提供します。Lambda さらにラッパースクリプトを呼び出します。これにより、Lambda はランタイム起動をスクリプトに委任できます。ラッパースクリプトを作成して、ランタイムの起動動作をカスタマイズできます。

言語固有の環境変数

Lambda は、次の言語固有の環境変数を通じて、関数の初期化中にコードをプリロードできるように設定専用の方法をサポートしています。

- `JAVA_TOOL_OPTIONS` – Java を使用する場合、Lambda は追加のコマンドライン変数を設定するために、この環境変数をサポートします。この環境変数では、ツールの初期化を指定できます。具体的には、`agentlib` または `javaagent` オプションを使用して、ネイティブまたは Java プログラミング言語エージェントの起動を指定できます。詳細については、「`JAVA_TOOL_OPTIONS` 環境変数」を参照してください。
- `NODE_OPTIONS` — Node.js ランタイムで使用できます。
- `DOTNET_STARTUP_HOOKS` -.NET Core 3.1 以降では、この環境変数が、Lambda が使用できるアセンブリ (dll) へのパスを指定します。

起動プロパティを設定するには、言語固有の環境変数を使用することを推奨します。

ラッパースクリプト

ラッパースクリプトを作成することで、Lambda 関数のランタイム起動動作をカスタマイズできます。ラッパースクリプトを使用すると、言語固有の環境変数を通じて、設定できない設定パラメーターを設定できます。

> **Note**
>
> ラッパースクリプトがランタイムプロセスを正常に開始しない場合、呼び出しが失敗することがあります。

ラッパースクリプトは、すべてのネイティブ Lambda ランタイムでサポートされています。ラッパースクリプトは OS 専用ランタイム (provided ランタイムファミリー) ではサポートされていません。

関数にラッパースクリプトを使用すると、Lambda はユーザーのスクリプトを使ってランタイムを開始します。Lambda は、インタープリターへのパスと、標準ランタイムを起動するための元の引数のすべてをスクリプトに送信します。スクリプトは、プログラムの起動動作を拡張または変換できます。例えば、スクリプトは、引数の挿入と変更、環境変数の設定、またはメトリクス/エラー/その他の診断情報の取得を行うことができます。

スクリプトを指定するには、実行可能バイナリまたはスクリプトのファイルシステムパスとして AWS_LAMBDA_EXEC_WRAPPER 環境変数の値を設定します。

例: Python 3.8 でのラッパースクリプトの作成と使用

次の例では、-X importtime オプションを指定して Python インタプリタを起動するラッパースクリプトを作成します。関数を実行すると、Lambda は、各インポートのインポートに要した時間を示すログエントリを生成します。

Python 3.8 でラッパースクリプトを作成して使用するには

1. ラッパースクリプトを作成するには、次のコードを importtime_wrapper という名前のファイルに貼り付けます。

```
#!/bin/bash

# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

2. スクリプトに実行可能なアクセス許可を与えるには、コマンドラインから「chmod +x importtime_wrapper」と入力します。

3. スクリプトを Lambda レイヤーとしてデプロイします。
4. Lambda コンソールを使用して関数を作成します。
   a. Lambda コンソールを開きます。
   b. [Create function] を選択します。
   c. [基本的な情報] の [関数名] に「**wrapper-test-function**」と入力します。
   d. [Runtime (ランタイム)] で [Python 3.8] を選択します。
   e. [Create function] を選択します。
5. レイヤーを関数に追加します。
   a. 関数を選択し、次に [コード] を選択します。
   b. [Add a layer] を選択します。
   c. [Choose a layer (レイヤーの選択)] で、前に作成した互換性のあるレイヤーの [名前] と [バージョン] を選択します。
   d. [Add] (追加) をクリックします。
6. コードと環境変数を関数に追加します。
   a. 関数コードエディタで、次の関数コードを貼り付けます。

```
import json

  def lambda_handler(event, context):
      # TODO implement
      return {
          'statusCode': 200,
          'body': json.dumps('Hello from Lambda!')
      }
```

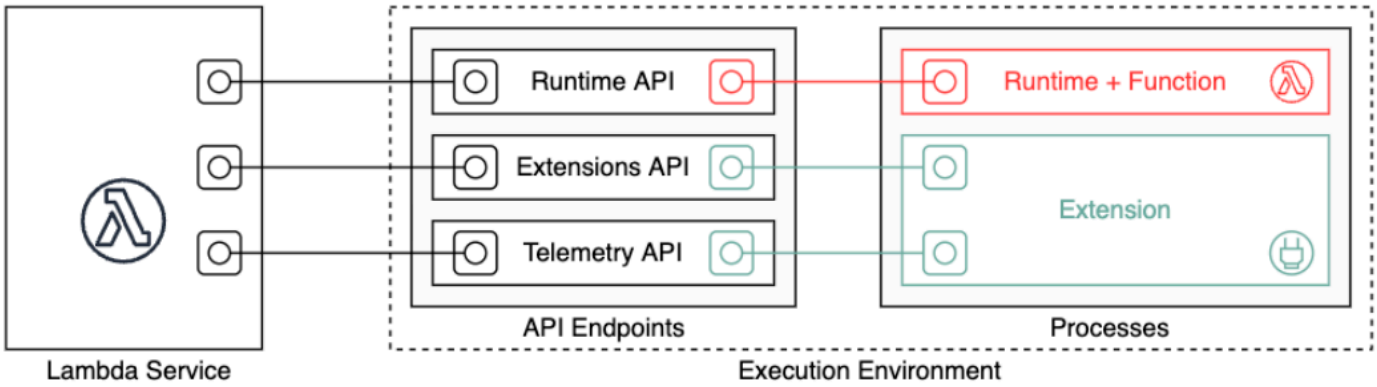
   b. [Save] を選択します。
   c. [環境変数] で、[編集] を選択します。
   d. [環境変数の追加] を選択します
   e. [キー] に「AWS_LAMBDA_EXEC_WRAPPER」と入力します。
   f. [Value (値)] に「/opt/importtime_wrapper」と入力します。
   g. [Save] を選択します。
7. 関数を実行するには、[テスト] を選択します。

ラッパースクリプトが `-X importtime` オプションで Python インタプリタを起動したため、ログには各インポートにかかった時間が示されます。以下に例を示します。

```
...
2020-06-30T18:48:46.780+01:00 import time: 213 | 213 | simplejson
2020-06-30T18:48:46.780+01:00 import time: 50 | 263 | simplejson.raw_json
...
```

Lambda Runtime API

AWS Lambda では、カスタムランタイムの HTTP API を使用して Lambda の呼び出しイベントを受け取り、レスポンスデータを Lambda の実行環境に送り返します。



ランタイム API バージョン 2018-06-01 の OpenAPI 仕様は、runtime-api.zip から入手できます。

API リクエスト URL を作成するには、ランタイムは `AWS_LAMBDA_RUNTIME_API` 環境変数から API エンドポイントを取得し、API バージョンを追加し、目的のリソースパスを追加します。

Example リクエスト

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API メソッド

- 次の呼び出し
- 呼び出しレスポンス
- 初期化エラー
- 呼び出しエラー

次の呼び出し

パス – `/runtime/invocation/next`

メソッド - GET

ランタイムは、このメッセージを Lambda に送信して、呼び出しイベントをリクエストします。レスポンス本文には、呼び出しのペイロードが含まれます。これは、関数トリガーのイベントデータを

含む JSON ドキュメントです。レスポンスヘッダーには、呼び出しに関する追加データが含まれます。

レスポンスヘッダー

- `Lambda-Runtime-Aws-Request-Id` - リクエスト ID。関数の呼び出しをトリガーしたリクエストを識別します。

  たとえば、`8476a536-e9f4-11e8-9739-2dfe598c3fcd` と指定します。

- `Lambda-Runtime-Deadline-Ms` - 関数がタイムアウトした日付 (Unix 時間のミリ秒)。

  たとえば、`1542409706888` と指定します。

- `Lambda-Runtime-Invoked-Function-Arn` - 呼び出しで指定されている Lambda 関数、バージョン、またはエイリアスの ARN。

  たとえば、`arn:aws:lambda:us-east-2:123456789012:function:custom-runtime` と指定します。

- `Lambda-Runtime-Trace-Id` - AWS X-Ray トレースヘッダー。

  たとえば、`Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1` と指定します。

- `Lambda-Runtime-Client-Context` - AWS Mobile SDK の呼び出しにおいて、クライアントアプリケーションおよびデバイスに関するデータ。

- `Lambda-Runtime-Cognito-Identity` - AWS Mobile SDK からの呼び出しの場合は、Amazon Cognito ID プロバイダーに関するデータ。

応答が遅れる可能性があるため、`GET` リクエストにタイムアウトを設定しないでください。Lambda がランタイムをブートストラップするときと、返すイベントがランタイムにあるときとの間に、ランタイムプロセスが数秒間停止する可能性があります。

リクエスト ID は、Lambda 内の呼び出しを追跡します。レスポンス送信時に呼び出しを指定する場合に使用します。

トレースヘッダーには、トレース ID、親 ID、サンプリングデシジョンが含まれます。リクエストがサンプリングされている場合、リクエストが Lambda、またはアップストリームサービスによってサンプリングされた場合。ランタイムは、`_X_AMZN_TRACE_ID` をヘッダーの値に設定します。X-Ray SDK はこの値を読み込んで ID を取得し、リクエストを追跡するかどうかを判断します。

呼び出しレスポンス

パス – `/runtime/invocation/AwsRequestId/response`

メソッド - POST

関数が実行されて完了すると、ランタイムは呼び出し応答を Lambda に送信します。同期呼び出しの場合、Lambda はそのレスポンスをクライアントに送ります。

Example 成功リクエスト

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

初期化エラー

関数がエラーを返すか、初期化中にランタイムでエラーが発生した場合、ランタイムはこのメソッドを使用してエラーを Lambda に報告します。

パス – `/runtime/init/error`

メソッド - POST

ヘッダー

`Lambda-Runtime-Function-Error-Type` – ランタイムで発生したエラータイプ。必須: いいえ。

このヘッダーは、文字列値で構成されています。Lambda はどのような文字列でも受け入れますが、形式は <category.reason> にすることが推奨されます。例:

- ランタイム。NoSuchHandler
- Runtime.APIKeyNotFound
- ランタイム。ConfigInvalid
- ランタイム。UnknownReason

Body パラメータ

ErrorRequest - エラーに関する情報。必須: いいえ。

このフィールドは、次の構造を持つ JSON オブジェクトです。

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Lambda は、errorType として任意の値を受け入れることに注意してください。

次の例は、呼び出しで指定されたイベントデータを関数で解析できなかった Lambda 関数のエラーメッセージを示しています。

Example 関数エラー

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

レスポンス本文のパラメータ

- StatusResponse – 文字列。202 応答コードとともに送信されるステータス情報。
- ErrorResponse - エラー応答コードとともに送信される追加のエラー情報。 ErrorResponse には、エラータイプとエラーメッセージが含まれています。

レスポンスコード

- 202 - Accepted
- 403 – Forbidden
- 500 – Container error 回復不能な状態。ランタイムはすぐに終了することが望ましいです。

Example 初期化エラーリクエスト

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
```

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

呼び出しエラー

関数がエラーを返すか、ランタイムでエラーが発生した場合、ランタイムはこのメソッドを使用してエラーを Lambda に報告します。

パス – `/runtime/invocation/AwsRequestId/error`

メソッド - POST

ヘッダー

`Lambda-Runtime-Function-Error-Type` – ランタイムで発生したエラータイプ。必須: いいえ。

このヘッダーは、文字列値で構成されています。Lambda はどのような文字列でも受け入れますが、形式は <category.reason> にすることが推奨されます。例:

- ランタイム。NoSuchHandler
- Runtime.APIKeyNotFound
- ランタイム。ConfigInvalid
- ランタイム。UnknownReason

Body パラメータ

`ErrorRequest` - エラーに関する情報。必須: いいえ。

このフィールドは、次の構造を持つ JSON オブジェクトです。

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Lambda は、`errorType` として任意の値を受け入れることに注意してください。

次の例は、呼び出しで指定されたイベントデータを関数で解析できなかった Lambda 関数のエラーメッセージを示しています。

Example 関数エラー

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

レスポンス本文のパラメータ

- StatusResponse – 文字列。202 応答コードとともに送信されるステータス情報。
- ErrorResponse - エラー応答コードとともに送信される追加のエラー情報。 ErrorResponse には、エラータイプとエラーメッセージが含まれています。

レスポンスコード

- 202 - Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error 回復不能な状態。ランタイムはすぐに終了することが望ましいです。

Example エラーリクエスト

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Lambda の OS 専用ランタイムを使用する状況

Lambda は Java、Python、Node.js、.NET、Ruby のマネージドランタイムを提供しています。マネージドランタイムとして使用できないプログラミング言語で Lambda 関数を作成するには、OS 専用ランタイム (`provided` ランタイムファミリー) を使用します。OS 専用ランタイムの主な使用例は次の 3 つです。

- ネイティブアヘッドオブタイム (AOT) コンパイル: Go、Rust、C++ などの言語は、実行可能なバイナリにネイティブにコンパイルされるため、専用の言語ランタイムは必要ありません。これらの言語に必要なのは、コンパイルされたバイナリを実行できる OS 環境だけです。Lambda OS 専用ランタイムを使用して、.NET ネイティブ AOT と Java GraalVM ネイティブでコンパイルされたバイナリをデプロイすることもできます。

  バイナリには、ランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは Lambda Runtime API を呼び出して関数呼び出しを取得し、次に関数ハンドラーを呼び出します。Lambda は Go、.NET Native AOT、C++、Rust (実験的) 用のランタイムインターフェイスクライアントを提供しています。

  バイナリは Linux 環境用に、関数に使用する予定のものと同じ命令セットアーキテクチャ (x86_64 または arm64) でコンパイルする必要があります。
- サードパーティランタイム: PHP 用の Bref や Swift 用の Swift AWS Lambda ランタイムなどの既製のランタイムを使用して Lambda 関数を実行できます。
- カスタムランタイム: Lambda がマネージドランタイムを提供していない言語または言語バージョン (Node.js 19 など) 用に独自のランタイムを構築できます。詳細については、「AWS Lambda 用カスタムランタイムの構築」を参照してください。これは OS 専用ランタイムでは最も稀なユースケースです。

Lambda は以下の OS 専用 ランタイムをサポートします。

OS 専用

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
OS 専用ランタイム	`provided.al2023`	Amazon Linux 2023			

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
OS 専用ランタイム	provided.al2	Amazon Linux 2			

Amazon Linux 2023 (provided.al2023) ランタイムには、デプロイのフットプリントが小さいことや、glibc などのライブラリのバージョンが更新されていることなど、Amazon Linux 2 に比べていくつかの利点があります。

provided.al2023 ランタイムは、Amazon Linux 2 のデフォルトのパッケージマネージャーである yum ではなく、dnf をパッケージマネージャーとして使用します。provided.al2023 と provided.al2 の違いの詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。

AWS Lambda 用カスタムランタイムの構築

AWS Lambda ランタイムは、どのプログラミング言語でも実装できます。ランタイムは、関数が呼び出されたときに Lambda 関数のハンドラメソッドを実行するプログラムです。ランタイムは、関数のデプロイパッケージに含めるか、またはレイヤーで配信することができます。Lambda 関数を作成するときは、OS 専用ランタイム (provided ランタイムファミリー) を選択します。

> **Note**
>
> カスタムランタイムの作成は高度なユースケースです。ネイティブバイナリへのコンパイルやサードパーティの既製のランタイムの使用に関する情報をお探しの場合は、「Lambda の OS 専用ランタイムを使用する状況」を参照してください。

カスタムランタイムデプロイの手順については、「チュートリアル: カスタムランタイムの構築」を参照してください。また、GitHub の awslabs/aws-lambda-cpp に C++ で実装されているカスタムランタイムを検証することもできます。

トピック

- 要件
- カスタムランタイムでのレスポンスストリーミングの実装

要件

カスタムランタイムでは、特定の初期化タスクと処理タスクを完了する必要があります。ランタイムは、関数のセットアップコードの実行、環境変数からのハンドラ名の読み取り、Lambda ランタイム API からの呼び出しイベントの読み取りを行います。ランタイムは、イベントデータを関数ハンドラに渡し、ハンドラからのレスポンスを Lambda に戻します。

初期化タスク

初期化タスクは、呼び出しを処理する環境を準備するために、関数のインスタンスごとに 1 回実行されます。

- 設定の取得 – 関数と環境に関する詳細を取得するには、環境変数を参照してください。
  - _HANDLER – 関数の設定からハンドラへの場所。標準形式は、*file*.*method* です。ここで、file は、拡張子のないファイル名、method は、ファイルで定義されているメソッドまたは関数の名前を表します。
  - LAMBDA_TASK_ROOT – 関数コードを含むディレクトリ。
  - AWS_LAMBDA_RUNTIME_API – ランタイム API のホストおよびポート。

  使用可能な変数の完全なリストについては、「定義されたランタイム環境変数」を参照してください。
- 関数の初期化 – ハンドラファイルをロードし、ハンドラファイルに含まれているグローバルコードまたは静的コードを実行します。関数は、SDK クライアントやデータベース接続などの静的リソースを一度作成し、複数の呼び出しに再利用します。
- ハンドラエラー – エラーが発生した場合は、初期化エラー API を呼び出し、すぐに終了します。

初期化は請求された実行時間とタイムアウトの対象です。実行によって関数の新しいインスタンスの初期化がトリガーされると、ログと AWS X-Ray トレースに初期化時間が表示されます。

Example ログ

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

処理タスク

実行されている間、ランタイムは Lambda ランタイムインターフェイスを使用して、受信イベントの管理とエラーのレポートを行います。初期化タスクが完了したら、ランタイムは、受信イベントをループで処理します。ランタイムコードで、次のステップを順に実行します。

- イベントの取得 – 次の呼び出し API を呼び出して、次のイベントを取得します。レスポンス本文には、イベントデータが含まれます。レスポンスヘッダーには、リクエスト ID などの情報が含まれます。
- トレースヘッダーの伝播 – X-Ray トレースヘッダーを API レスポンスの`Lambda-Runtime-Trace-Id`ヘッダーから取得します。`_X_AMZN_TRACE_ID` 環境変数をローカルで同じ値に設定します。X-Ray SDK はこの値を使用して、サービス間でトレースデータを接続します。
- コンテキストオブジェクトの作成 – 環境変数のコンテキスト情報および API レスポンスのヘッダーでオブジェクトを作成します。
- 関数ハンドラの呼び出し – イベントおよびコンテキストオブジェクトをハンドラに渡します。
- レスポンスの処理 – 呼び出しレスポンス API を呼び出し、レスポンスをハンドラから投稿します。
- ハンドラエラー – エラーが発生した場合は、呼び出しエラー API を呼び出します。
- クリーンアップ – 不要なリソースの解放、他のサービスへのデータ送信、次のイベント取得前の追加タスクの実行を行います。

エントリポイント

カスタムランタイムのエンドポイントは、`bootstrap` という名前の実行可能ファイルです。ブートストラップファイルをランタイムにするか、ランタイムを作成する別のファイルを呼び出す場合があります。デプロイパッケージのルートに `bootstrap` という名前のファイルが含まれていない場合、Lambda は関数のレイヤーでファイルを検索します。`bootstrap` ファイルが存在しないか、実行可能でない場合、関数は呼び出し時に `Runtime.InvalidEntrypoint` エラーを返します。

`bootstrap` ファイルによる例では、バンドルされた Node.js バージョンを使用して、`runtime.js` という別のファイルで JavaScript ランタイムを実行します。

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

カスタムランタイムでのレスポンスストリーミングの実装

レスポンスストリーミング関数については、response および error エンドポイントは、ランタイムが部分的なレスポンスをクライアントにストリーミングし、ペイロードをチャンクで返すことができるように動作が若干変更されています。特定の動作の詳細については、以下を参照してください。

- /runtime/invocation/AwsRequestId/response – ランタイムからの Content-Type ヘッダーを伝播して、クライアントに送信します。Lambda は、HTTP/1.1 チャンク転送エンコーディングを介して、レスポンスペイロードをチャンクで返します。レスポンスストリーミングは最大サイズは 20 MiB まで許可されます。Lambda にレスポンスをストリーミングするには、ランタイムが以下を実行する必要があります。
  - Lambda-Runtime-Function-Response-Mode HTTP ヘッダーを streaming に設定する。
  - Transfer-Encoding ヘッダーを chunked に設定します。
  - HTTP/1.1 チャンク転送エンコーディング仕様に従ったレスポンスを書き込む
  - レスポンスを正常に書き込んだ後、基盤となる接続を閉じます。
- /runtime/invocation/AwsRequestId/error – ランタイムはこのエンドポイントを使用して、関数またはランタイムのエラーを Lambda に報告できます。Lambda は Transfer-Encoding ヘッダーも受け入れます。このエンドポイントを呼び出せるのは、ランタイムが呼び出しレスポンスの送信を開始する前だけです。
- /runtime/invocation/AwsRequestId/response でエラー トレーラーを使用して中間エラーを報告 - ランタイムが呼び出しレスポンスの書き込みを開始した後に発生したエラーを報告するために、ランタイムはオプションで Lambda-Runtime-Function-Error-Type および Lambda-Runtime-Function-Error-Body という名前の HTTP 末尾ヘッダーをアタッチすることができます。Lambda はこれを正常なレスポンスとして扱い、ランタイムが提供するエラーメタデータをクライアントに転送します。

> **Note**
>
> 末尾ヘッダーをアタッチするには、ランタイムが HTTP リクエストの先頭に Trailer ヘッダー値を設定する必要があります。これは、HTTP/1.1 チャンク転送エンコーディング仕様によって義務付けられています。

  - Lambda-Runtime-Function-Error-Type – ランタイムで発生したエラータイプ。このヘッダーは、文字列値で構成されています。Lambda はどのような文字列でも受け入れますが、形式

は *<category.reason>* にすることが推奨されます。例えば、Runtime.APIKeyNotFound と指定します。

- Lambda-Runtime-Function-Error-Body – Base64-encoded でのエラーに関する情報。

チュートリアル: カスタムランタイムの構築

このチュートリアルでは、カスタムランタイムで Lambda 関数を使用します。まず、ランタイムを関数のデプロイパッケージに含めます。次に、それを関数とは別に管理するレイヤーに移行します。最後に、リソースベースのアクセス許可ポリシーを更新して、ランタイムレイヤーを世界と共有します。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

Lambda 関数を作成するには IAM ロールが必要です。ロールには、ログを CloudWatch Logs に送信し、関数が使用する AWSサービスにアクセスするためのアクセス許可が必要です。関数開発用の実行ロールをお持ちでない場合は、ここで作成します。

実行ロールを作成するには

1. IAM コンソールの [ロールページ] を開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。

   - 信頼されたエンティティ – Lambda。
   - アクセス許可 – AWSLambdaBasicExecutionRole。
   - [ロール名] - **lambda-role**

   このAWSLambdaBasicExecutionRoleポリシーには、関数が CloudWatch ログにログを書き込むために必要なアクセス許可があります。

関数の作成

カスタムランタイムで Lambda 関数を作成します。この例には、ランタイム `bootstrap` ファイルと関数ハンドラーの 2 つのファイルが含まれています。いずれのファイルも Bash で実装されています。

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

   ```
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. `bootstrap` という名前の新しいファイルを作成します。これはカスタムランタイムです。

   Example ブートストラップ

   ```
   #!/bin/sh

   set -euo pipefail

   # Initialization - load function handler
   source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"
   ```

```
# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

ランタイムは、デプロイパッケージから関数スクリプトを読み込みます。2 つの変数を使用して、スクリプトを見つけます。LAMBDA_TASK_ROOT は、パッケージが抽出された場所を変数に伝え、_HANDLER には、そのスクリプトの名前が含まれます。

ランタイムは関数スクリプトをロードした後、ランタイム API を使用して Lambda から呼び出しイベントを取得し、そのイベントをハンドラーに渡して、レスポンスを Lambda に戻します。リクエスト ID を取得するには、API レスポンスのヘッダーを一時ファイルに保存し、ファイルから `Lambda-Runtime-Aws-Request-Id` ヘッダーを読み込みます。

> **Note**
>
> ランタイムは、他にもエラーの処理などに使用され、コンテキスト情報をハンドラに提供します。詳細については、「要件」を参照してください。

3. 関数のためのスクリプトを作成します。以下のスクリプト例は、イベントデータを取得するハンドラー関数を定義し、それを `stderr` にログ記録して返します。

   Example function.sh

```
function handler () {
  EVENT_DATA=$1
```

```
  echo "$EVENT_DATA" 1>&2;
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

runtime-tutorial ディレクトリは以下のようになります。

```
runtime-tutorial
# bootstrap
# function.sh
```

4. ファイルを実行可能にして .zip アーカイブに追加します。これがデプロイパッケージです。

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. bash-runtimeという名前の関数を作成します。--role には、Lambda 実行ロールの ARN を入力します。

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. 関数を呼び出します。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

次のような結果が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
```

```
}
```

7. レスポンスを確認してください。

```
cat response.txt
```

次のような結果が表示されます。

```
Echoing request: '{"text":"Hello"}'
```

レイヤーの作成

ランタイムコードと関数コードを区別するには、ランタイムのみを含むレイヤーを作成します。レイヤーを使用すると、関数の依存関係を個別に開発することができ、複数の関数で同じレイヤーを使用する場合には、ストレージの使用量を抑えることができます。詳細については、「レイヤーによる Lambda 依存関係の管理」を参照してください。

1. bootstrap ファイルを含む .zip ファイルを作成します。

```
zip runtime.zip bootstrap
```

2. publish-layer-version コマンドでレイヤーを作成します。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

これにより、最初のバージョンのレイヤーが作成されます。

関数の更新

関数でランタイムレイヤーを使用するには、レイヤーを使用するように関数を設定し、関数からランタイムコードを削除します。

1. 関数設定を更新して、レイヤーに取り込みます。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
```

これにより、ランタイムが /opt ディレクトリの関数に追加されます。Lambda がレイヤーのランタイムを使用するようにするには、次の 2 つのステップに示すように、関数のデプロイパッケージから boostrap を削除する必要があります。

2. 関数コードを含む .zip ファイルを作成します。

```
zip function-only.zip function.sh
```

3. ハンドラスクリプトのみ含まれるように関数コードを更新します。

```
aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
function-only.zip
```

4. 関数を呼び出し、ランタイムレイヤーで正常に動作することを確認します。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

次のような結果が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. レスポンスを確認してください。

```
cat response.txt
```

次のような結果が表示されます。

```
Echoing request: '{"text":"Hello"}'
```

ランタイムの更新

1. 実行環境に関する情報を口グ記録するには、環境変数が出力されるようにランタイムスクリプトを更新します。

   Example ブートストラップ

   ```
   #!/bin/sh

   set -euo pipefail

   # Configure runtime to output environment variables
   echo "##  Environment variables:"
   env

   # Load function handler
   source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

   # Processing
   while true
   do
     HEADERS="$(mktemp)"
     # Get an event. The HTTP request will block until one is received
     EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
   ${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

     # Extract request ID by scraping response headers received above
     REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
    '[:space:]' | cut -d: -f2)

     # Run the handler function from the script
     RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

     # Send the response
     curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
   response"  -d "$RESPONSE"
   done
   ```

2. bootstrap ファイルの新しいバージョンを含む .zip ファイルを作成します。

   ```
   zip runtime.zip bootstrap
   ```

3. bash-runtime レイヤーの新しいバージョンを作成します。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. 新しいバージョンのレイヤーを使用するように関数を設定します。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

レイヤーを共有する

レイヤーの使用に関するアクセス許可を別のアカウントに付与するには、add-layer-version-permission コマンドを使用して、ステートメントをレイヤーバージョンのアクセス許可ポリシーに追加します。アクセス許可は、各ステートメントで、1 つのアカウント、すべてのアカウント、または組織に付与することができます。

以下の例では、アカウント 111122223333 に bash-runtime レイヤーのバージョン 2 へのアクセス許可を付与します。

```
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id
 xaccount \
--action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output
 text
```

次のような出力が表示されます。

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16    {"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

アクセス許可は、単一レイヤーバージョンにのみ適用されます。新しいレイヤーバージョンを作成するたびに、このプロセスを繰り返します。

クリーンアップ

各バージョンのレイヤーを削除します。

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

バージョン 2 のレイヤーへの参照が関数で保持されているため、現在も Lambda に存在します。関数は引き続き動作しますが、削除したバージョンが使用されるように、参照を設定することはできません。関数のレイヤーのリストを変更する場合は、新しいバージョンを指定するか、削除したレイヤーを除外する必要があります。

delete-function コマンドを使用して関数を削除します。

```
aws lambda delete-function --function-name bash-runtime
```

Lambda で AVX2 ベクトル化を使用する

アドバンスドベクトルエクステンション 2 (AVX2) はインテル x86 命令セットのベクトル化拡張で、256ビット以上のベクトルをシングルインストラクションマルチプルデータ (SIMD) で実行できるものです。高度に並列化可能な操作ができるベクトル化可能なアルゴリズムの場合、AVX2 を使用すると CPU パフォーマンスが向上され、その結果レイテンシーが低くなり、スループットが向上します。AVX2 命令セットは、機械学習推定、マルチメディア処理、科学シミュレーション、財務モデリングアプリケーションなどの計算負荷の高いワークロードに使用します。

> **Note**
>
> Lambda arm64 は NEON SIMD アーキテクチャを使用します。また、x86 AVX2 拡張機能をサポートしません。

Lambda 関数で AVX2 を使用するには、関数コードが AVX2 最適化コードにアクセスしていることを確認してください。言語によっては、AVX2 がサポートするバージョンのライブラリとパッケージをインストールできます。他の言語では、適切なコンパイラーフラグセット (コンパイラーが自動ベクトル化をサポートしている場合) を使用して、コードと依存関係を再コンパイルできます。また、AVX2 を使用して演算機能を最適化するサードパーティライブラリを使用してコードをコンパイルすることもできます。たとえば、インテルマスカーネルライブラリー (インテル MKL)、OpenBLAS (基本線形代数サブプログラム)、AMD BLASライクライブラリインスタンシエーションソフトウェア （ BLIS) などです。Java などの自動ベクトル化された言語では、計算に AVX2 が自動的に使用されます。

追加コストなしで、新しい Lambda ワークロードを作成したり、既存の AVX2 対応のワークロードを Lambda に移動させたりすることができます。

AVX2 の詳細については、Wikipedia の「 Advanced Vector Extensions 2 」を参照してください。

ソースからのコンパイル

Lambda 関数が C または C++ ライブラリを使用して計算負荷の高いベクトル化可能な操作を実行する場合は、適切なコンパイラフラグを設定し、関数コードを再コンパイルできます。次に、コンパイラーはコードを自動的にベクトル化します。

`gcc` 、 `clang` コンパイラでは、`-march=haswell` をコマンドに追加するか、`-mavx2` をコマンドオプションとして設定します。

```
~ gcc -march=haswell main.c
or
~ gcc -mavx2 main.c

~ clang -march=haswell main.c
or
~ clang -mavx2 main.c
```

特定のライブラリを使用するには、ライブラリのドキュメントの指示に従ってライブラリをコンパイルおよび構築します。例えば、ソース TensorFlow から構築するには、 TensorFlow ウェブサイトのインストール手順に従ってください。コンパイルオプション `-march=haswell` を使用してください。

インテル MKL での AVX2 の有効化

インテル MKL は、コンピューティングプラットフォームが AVX2 命令をサポートしている場合、暗に AVX2 命令を使用する演算機能が最適化されたライブラリです。などのフレームワークは PyTorch 、デフォルトでインテル MKL を使用して構築されるため、AVX2 を有効にする必要はありません。

などの一部のライブラリでは TensorFlow、ビルドプロセスでインテル MKL 最適化を指定するオプションが提供されます。例えば、 では TensorFlow、 `--config=mkl`オプションを使用します。

インテル MKL NumPyを使用して、 SciPy や などの一般的な科学 Python ライブラリを構築することもできます。インテル MKL でこれらのライブラリを構築する方法については、インテル ウェブサイトの Numpy/Scipy インテル MKL およびインテル コンパイラー を参照してください。

Intel MKL および同様のライブラリの詳細については、Wikipedia の Math Kernel Library、OpenBLAS ウェブサイト 、および の AMD BLIS リポジトリを参照してください GitHub。

他の言語での AVX2 のサポート

C または C++ ライブラリを使用せず、インテル MKL で構築しない場合も、アプリケーションの AVX2 のパフォーマンス向上ができます。実際の改善はコンパイラーまたはインタープリタのコードで AVX2 機能を活用する能力次第です。

Python

Python ユーザーは通常、計算負荷の高いワークロードに SciPy および NumPy ライブラリを使用します。これらのライブラリをコンパイルして AVX2 を有効にすることも、インテル MKL 対応バージョンのライブラリを使用することもできます。

ノード

処理負荷の高いワークロードの場合は、AVX2 対応、またはインテル MKL 対応バージョンのライブラリを使用してください。

Java

Java の JIT コンパイラーは、AVX2 命令で実行するようにコードを自動ベクトル化できます。ベクトル化されたコードの検出については、OpenJDK ウェブサイト、 JVM プレゼンテーションの「コードのベクトル化」を参照してください。

Go

標準の Go コンパイラーは現在、自動ベクトル化をサポートしていませんが、Go の GCC コンパイラー gccgo を使用できます。`-mavx2` オプションを設定します。

```
gcc -o avx2 -mavx2 -Wall main.c
```

組込み関数

多くの言語では 組み込み関数 を使用して、コードを手動でベクトル化して AVX2 を使用することが可能です。ただし、このアプローチはお勧めしません。ベクトル化されたコードを手動で記述すると、多大な労力がかかります。また、このようなコードのデバッグと管理は、自動ベクトル化に依存するコードを使用するよりも困難です。

AWS Lambda 関数の設定

Lambda API またはコンソールを使用して、Lambda 関数の中核的な機能とオプションを設定する方法を学びます。

「メモリ」

関数のメモリを増やす方法とタイミングについて説明します。

エフェメラルストレージ

関数の一時ストレージ容量を増やす方法とタイミングについて説明します。

タイムアウト

関数のタイムアウト値を増やす方法とタイミングについて説明します。

環境変数

環境変数を使用することによって、関数コードを移植可能にするとともに、シークレットを関数の設定内に保存することで、それらがコードに含まれないようにします。

アウトバウンドネットワーク

Lambda 関数は、Amazon VPC 内の AWS リソースと使用できます。関数を VPC に接続すると、リレーショナルデータベースやキャッシュなどのプライベートサブネットのリソースにアクセスできます。

インバウンドネットワーク

インターフェイス VPC エンドポイントを使用して、パブリックインターネットを経由せずに Lambda 関数を呼び出すことができます。

ファイルシステム

Lambda 関数を使用して、Amazon EFS をローカルディレクトリにマウントすることができます。ファイルシステムは、関数コードが安全かつ高い同時実行性で共有リソースにアクセスし、変更することを可能にします。

エイリアス

エイリアスを使用することによって、クライアントを更新する代わりに、特定の Lambda 関数バージョンを呼び出すようにクライアントを設定できます。

バージョン

関数のバージョンを発行することによって、変更できない別個のリソースとしてコードと設定を保存できます。

レスポンスストリーミング

Lambda 関数 URL を設定して、レスポンスペイロードをクライアントにストリーミングで返すようにできます。レスポンスストリーミングは、最初のバイトまでの時間 (TTFB) のパフォーマンスを向上させることで、レイテンシーの影響を受けやすいアプリケーションに役立ちます。これは、レスポンスの一部が利用可能になったときにクライアントに返送できるためです。さらに、レスポンスストリーミングを使用して、より大きなペイロードを返す関数を構築できます。

Lambda 関数のメモリを設定

Lambda は、設定されたメモリの量に比例して CPU パワーを割り当てます。メモリは、実行時に Lambda 関数で使用できるメモリの量です。[メモリ] 設定を使用して、関数に割り当てられたメモリと CPU パワーを増減できます。メモリは、128 MB ~ 10,240 MB の値を 1 MB 単位で設定できます。1,769 MB の場合、1 つの vCPU (1 秒あたりのクレジットの 1 vCPU 秒分) に相当します。

このページでは、Lambda 関数のメモリ設定を更新の方法とタイミングについて説明します。

セクション

Lambda 関数の適切なメモリ設定を確認する

メモリは、関数のパフォーマンスを制御するための主要な手段です。デフォルト設定の 128 MB は、設定可能な最小値です。イベントを変換して他のサービスにルーティングする関数など、シンプルな Lambda 関数には 128 MB で使用することをお勧めしますAWS。メモリ割り当てを増やすと、インポートされたライブラリ Lambda レイヤー、Amazon Simple Storage Service (Amazon S3)、または Amazon Elastic File System (Amazon EFS) を使用する関数のパフォーマンスを向上させることができます。メモリを追加すると、CPU の処理量が比例的に増加して、計算能力全体が向上します。関数が CPU、ネットワーク、またはメモリにバインドされている場合、メモリ設定を増やすとパフォーマンスが大幅に向上する可能性があります。

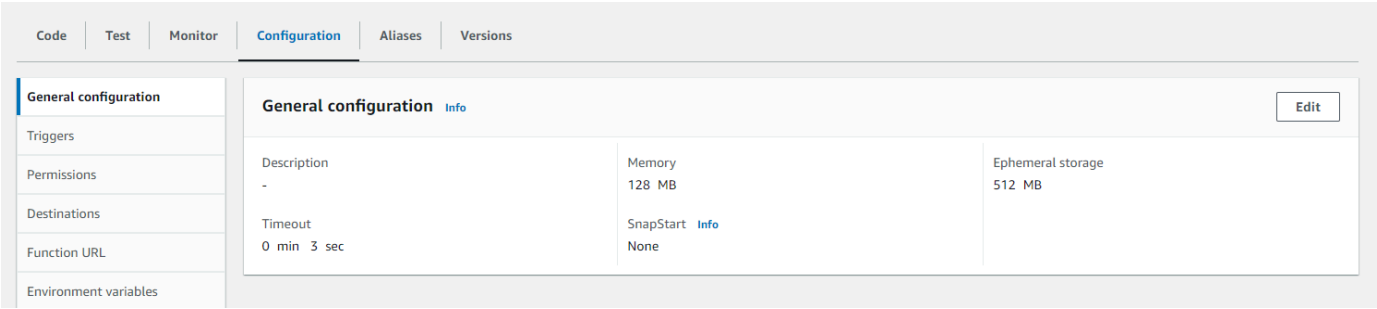
関数に適切なメモリの設定を見つけるには、オープンソースの AWS Lambda Power Tuning ツールを使用することを推奨しています。このツールは AWS Step Functions を使用して、異なるメモリ割り当てで複数のバージョンの Lambda 関数を同時に実行し、パフォーマンスを測定します。入力関数は AWS アカウントで実行され、ライブ HTTP 呼び出しと SDK インタラクションを実行して、ライブ本番環境で期待されるパフォーマンスを測定します。このツールを使用して、デプロイする新しい関数のパフォーマンスを自動的に測定する CI/CD プロセスを実装することもできます。

関数メモリの設定 (コンソール)

関数のメモリは Lambda コンソールで設定できます。

関数のメモリ割り当てを変更するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[一般設定] を選択します。



4. [全般設定] で、[編集] を選択します。
5. [メモリ] に、128 MB から 10,240 MB の値を設定します。
6. [Save] を選択します。

関数のメモリの設定 (AWS CLI)

update-function-configuration コマンドを使用して、関数のメモリを設定できます。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

関数のメモリの設定 (AWS SAM)

AWS Serverless Application Model を使用して、関数のメモリを設定できます。`template.yaml` ファイル内の MemorySize プロパティを更新し、sam deploy を実行します。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
      # Other function properties...
```

関数のメモリーの推奨事項を受け入れる (コンソール)

AWS Identity and Access Management (IAM) での管理者権限がある場合は、Lambda 関数のメモリ設定に関する推奨事項を、AWS Compute Optimizer から受け取るようにオプトインできます。アカウントまたは組織でメモリの推奨事項にオプトインする手順については、AWS Compute Optimizer ユーザーガイドの「アカウントにオプトインする」を参照してください。

> **Note**
>
> Compute Optimizer は x86_64 アーキテクチャを使用する関数のみをサポートします。

オプトインが完了しており、Lambda 関数が Compute Optimizer の要件を満たしている場合は、Compute Optimizer による関数のメモリに関する推奨事項を、Lambda コンソールの [一般設定] で表示したり、受け入れたりすることができます。

Lambda 関数のエフェメラルストレージを設定する

Lambda は、 /tmp ディレクトリ内の関数にエフェメラルストレージを提供します。このストレージは一時的なものであり、各実行環境に固有のものです。エフェメラルストレージ設定で、関数に割り当てられるエフェメラルストレージの容量を変更できます。エフェメラルストレージの容量は、512 MB から 10,240 MB まで、1-MB 単位で設定できます。/tmp に保存されているすべてのデータは、AWS によって管理されるキーを使用して保管時に暗号化されます。

このページでは、一般的なユースケースと、Lambda 関数のエフェメラルストレージを更新する方法について説明します。

セクション

エフェメラルストレージを増やす一般的なユースケース

エフェメラルストレージを増やすことでメリットを得られる一般的なユースケースを以下に示します。

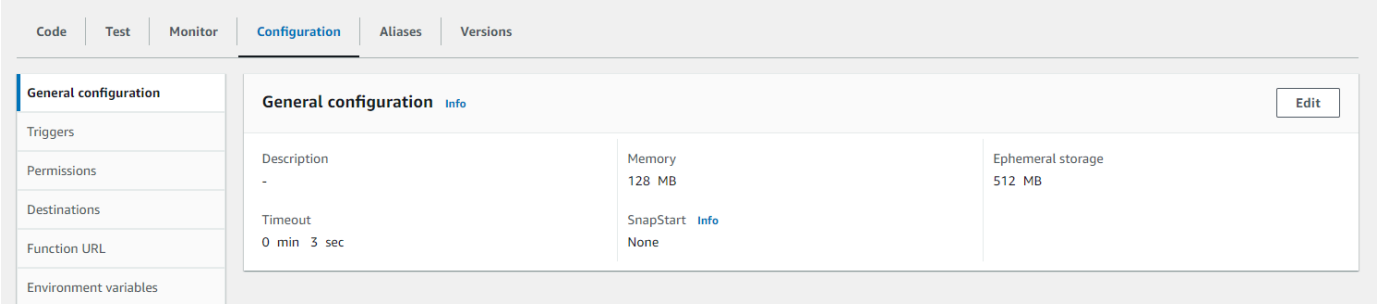
- 抽出/変換ロード (ETL) ジョブ: コードが中間計算を実行するか、他のリソースをダウンロードして処理を完了すると、エフェメラルストレージが増加します。一時スペースが多いほど、Lambda 関数でより複雑な ETL ジョブを実行できます。
- 機械学習 (ML) 推論： 多くの推論タスクは、ライブラリやモデルを含む大規模なリファレンスデータファイルに依存しています。エフェメラルストレージを使用すると、Amazon Simple Storage Service (Amazon S3) から /tmp により大きなモデルをダウンロードして、処理に使用できます。
- データ処理： Amazon S3S3 からオブジェクトをダウンロードするワークロードの場合、/tmp領域を増やすと、インメモリ処理を考慮せずに大きなオブジェクトを処理できます。PDF を作成したり、メディアを処理したりするワークロードも、より一時的なストレージの恩恵を受けます。
- グラフィック処理: 画像処理は Lambda ベースのアプリケーションの一般的なユースケースです。大きな TIFF ファイルまたは衛星画像を処理するワークロードの場合、エフェメラルストレージを使用すると、ライブラリの使用と Lambda での計算の実行が容易になります。

エフェメラルストレージの設定 (コンソール)

Lambda コンソールでエフェメラルストレージを設定できます。

関数のエフェメラルストレージを変更するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[一般設定] を選択します。



4. [全般設定] で、[編集] を選択します。
5. エフェメラルストレージ の場合、512 MB から 10,240 MB までの値を 1-MB 単位で設定します。
6. [Save] を選択します。

エフェメラルストレージの設定 (AWS CLI)

update-function-configuration コマンドを使用して、エフェメラルストレージを設定できます。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

エフェメラルストレージの設定 (AWS SAM)

AWS Serverless Application Model を使用して、関数のエフェメラルストレージを設定できます。`template.yaml` ファイル内の EphemeralStorage プロパティを更新し、sam deploy を実行します。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs20.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

Lambda 関数のタイムアウトを設定する

Lambda は、コードを一定時間実行してからタイムアウトします。タイムアウトとは、Lambda 関数がタイムアウトするまでの最大実行時間です。この設定のデフォルト値は 3 秒ですが、最大値の 900 秒 (15 分) まで 1 秒単位で調整できます。

このページでは、Lambda 関数のタイムアウト設定を更新する方法とタイミングについて説明します。

セクション

Lambda 関数の適切なタイムアウト値を決定する

タイムアウト値が関数の平均期間に近い場合、関数が予期せずタイムアウトするリスクが高くなります。関数の期間は、データ転送と処理の量、および関数が相互作用するサービスのレイテンシーによって異なる場合があります。タイムアウトの一般的な原因には、次のようなものがあります。

- Amazon Simple Storage Service (Amazon S3) からダウンロードを行う場合、通常よりもサイズが大きくなり、時間がかかります。
- 関数が別のサービスにリクエストを行う場合、応答に時間がかかります。
- 関数に使用されるパラメータでは、関数の計算が複雑になり、呼び出しに時間がかかります。

アプリケーションをテストする際は、テストがデータのサイズと量、および実際のパラメータ値を正確に反映していることを確認してください。テストでは、便宜上小さいサンプルがよく使用されますが、ワークロードで適度に予想される上限のデータセットを使用する必要があります。

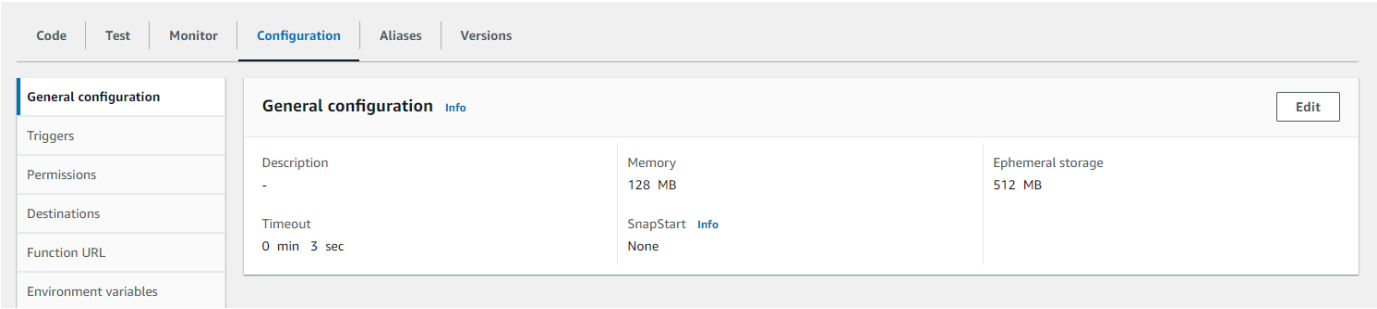
タイムアウトの設定 (コンソール)

Lambda コンソールで関数のタイムアウトを設定できます。

関数のタイムアウトを変更する方法

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数を選択します。
3. [設定] を選択してから、[一般設定] を選択します。



4. [全般設定] で、[編集] を選択します。
5. [タイムアウト] に、1 ~ 900 秒 (15 分) の値を入力します。
6. [Save] を選択します。

タイムアウトの設定 (AWS CLI)

update-function-configuration コマンドを使用して、タイムアウト値を秒単位で設定できます。次のコマンドの例では、関数のタイムアウトを 120 秒 (2 分) に増やします。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

タイムアウトの設定 (AWS SAM)

AWS Serverless Application Model を使用して、関数のタイムアウト値を設定できます。`template.yaml` ファイルの Timeout プロパティを更新し、sam deploy を実行します。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
```

```
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      # Other function properties...
```

Lambda 環境変数を使用したコードの値の設定

環境変数を使用すると、コードを更新せずに関数の動作を調整できます。環境変数は、関数のバージョン固有の設定に保存される文字列のペアです。Lambda ランタイムは、環境変数をコードで使用できるようにし、関数と呼び出しリクエストに関する情報を含む追加の環境変数を設定します。

> **Note**
>
> セキュリティを強化するには、環境変数の代わりに AWS Secrets Manager を使用してデータベースの認証情報や (API キーや認証トークンなど) その他の機密情報を保存することをお勧めします。詳細については、「AWS Secrets Manager を使用したシークレットの作成および管理」を参照してください。

環境変数は、関数を呼び出す前には評価されません。定義した値はリテラル文字列とみなされ、展開されません。関数コードで変数の評価を実行します。

Lambda コンソール、AWS Command Line Interface (AWS CLI)、AWS Serverless Application Model (AWS SAM)、または AWS SDK を使用して、Lambda で環境変数を設定できます。

Console

環境変数は、関数の未公開バージョンで定義します。バージョンを公開するとき、他のバージョン固有の構成設定とともに、そのバージョンの環境変数がロックされます。

関数の環境変数を作成するには、キーと値を定義します。関数は、キーの名前を使用して、環境変数の値を取得します。

Lambda コンソールで環境変数を設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [Configuration] (設定) を選択してから、[Environment variables] (環境変数) を選択します。
4. [環境変数] で、[編集] を選択します。
5. [環境変数の追加] を選択します。
6. キーと値を入力します。

要件

- キーは文字で始まり、少なくとも 2 文字です。
- キーには、文字、数字、およびアンダースコア (_) のみを含める。
- キーは Lambda によって予約されていない。
- すべての環境変数の合計サイズは 4 KB を超えない。

7. [Save] を選択します。

コンソールコードエディタで環境変数のリストを生成するには

Lambda コードエディタで環境変数のリストを生成することができます。これを使用することで、コーディング中に環境変数を簡単に参照することができます。

1. [コード] タブを選択します。
2. [環境変数] タブを選択します。
3. [ツール]、[環境変数を表示] の順に選択します。

環境変数は、コンソールのコードエディタに一覧表示されても暗号化されたままです。転送中の暗号化の暗号化ヘルパーを有効にした場合、それらの設定は変更されません。詳細については、「Lambda 環境変数の保護」を参照してください。

環境変数リストは読み取り専用で、Lambda コンソールでのみ使用できます。このファイルは、関数の .zip ファイルアーカイブをダウンロードしたときには含まれていません。また、このファイルをアップロードしても環境変数を追加することはできません。

AWS CLI

次の例では、`my-function` という名前の関数に 2 つの環境変数を設定します。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=DOC-EXAMPLE-BUCKET,KEY=file.txt}"
```

`update-function-configuration` コマンドを使用して環境変数を適用すると、`Variables` 構造体の内容全体が置き換えられます。新しい環境変数を追加するときに既存の環境変数を保持するには、リクエストに既存の値をすべて含めます。

現在の設定を取得するには、`get-function-configuration` コマンドを使用します。

```
aws lambda get-function-configuration \
  --function-name my-function
```

以下の出力が表示されます。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "DOC-EXAMPLE-BUCKET",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
    ...
}
```

get-function-configuration の出力にあるリビジョン ID をパラメータとして update-function-configuration に渡すことができます。これにより、構成を読み込んだときから更新したときまでの間に、値が変更されることはありません。

関数の暗号化キーを設定するには、KMSKeyARN オプションを設定します。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

AWS SAM

AWS Serverless Application Model を使用して関数の環境変数を設定できます。template.yaml ファイル内の Environment プロパティと Variables プロパティを更新し、sam deploy を実行します。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
```

```
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs18.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      Environment:
        Variables:
          BUCKET: DOC-EXAMPLE-BUCKET
          KEY: file.txt
      # Other function properties...
```

AWS SDKs

AWS SDK を使用して環境変数を管理するには、以下の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

詳細については、目的のプログラミング言語の AWS SDK ドキュメントを参照してください。

定義されたランタイム環境変数

Lambda ランタイムは、初期化中にいくつかの環境変数を設定します。ほとんどの環境変数は、関数またはランタイムに関する情報を提供します。これらの環境変数のキーは予約済みであるため、関数設定では設定できません。

予約済み環境変数

- _HANDLER - 関数に設定されているハンドラの場所。
- _X_AMZN_TRACE_ID - X-Ray トレースヘッダー。この環境変数は呼び出しごとに変化します。

  - この環境変数は OS 専用ランタイム (provided ランタイムファミリー) には定義されていません。カスタムランタイムには 次の呼び出し からの Lambda-Runtime-Trace-Id レスポンスのヘッダーに _X_AMZN_TRACE_ID を設定できます。
  - Java ランタイムバージョン 17 以降では、この環境変数は使用されません。代わりに、Lambda はトレース情報を com.amazonaws.xray.traceHeader システムプロパティに保存します。
- AWS_DEFAULT_REGION - Lambda 関数が実行されるデフォルトの AWS リージョン。
- AWS_REGION - Lambda 関数が実行される AWS リージョン。定義されている場合、この値は AWS_DEFAULT_REGION を上書きします。
  - AWS SDK での AWS リージョン 環境変数を使用する方法の詳細については、「AWS SDK とツールリファレンスガイド」の「AWSリージョン」を参照してください。
- AWS_EXECUTION_ENV – AWS_Lambda_ (例: AWS_Lambda_java8) のプレフィックスが付いたランタイム識別子。この環境変数は OS 専用ランタイム (provided ランタイムファミリー) には定義されていません。
- AWS_LAMBDA_FUNCTION_NAME - 関数の名前。
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE - 関数で使用できるメモリの量 (MB 単位)。
- AWS_LAMBDA_FUNCTION_VERSION - 実行される関数のバージョン。
- AWS_LAMBDA_INITIALIZATION_TYPE - 関数の初期化タイプ。これは、on-demand、provisioned-concurrency、または snap-start になります。詳細については、「プロビジョニングされた同時実行の設定」または「Lambda SnapStart による起動パフォーマンスの向上」を参照してください。
- AWS_LAMBDA_LOG_GROUP_NAME、AWS_LAMBDA_LOG_STREAM_NAME - Amazon CloudWatch Logs グループの名前と関数のストリーム。AWS_LAMBDA_LOG_GROUP_NAME および AWS_LAMBDA_LOG_STREAM_NAME の環境変数は Lambda SnapStart 関数では使用できません。
- AWS_ACCESS_KEY、AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN – 関数の実行ロールから取得したアクセスキー。
- AWS_LAMBDA_RUNTIME_API - (カスタムランタイム) ランタイム API のホストおよびポート。
- LAMBDA_TASK_ROOT - Lambda 関数コードへのパス。
- LAMBDA_RUNTIME_DIR - ランタイムライブラリへのパス。

以下の追加の環境変数は予約されていないため、関数設定で拡張できます。

予約されていない環境変数

- LANG - ランタイムのロケール (en_US.UTF-8)。

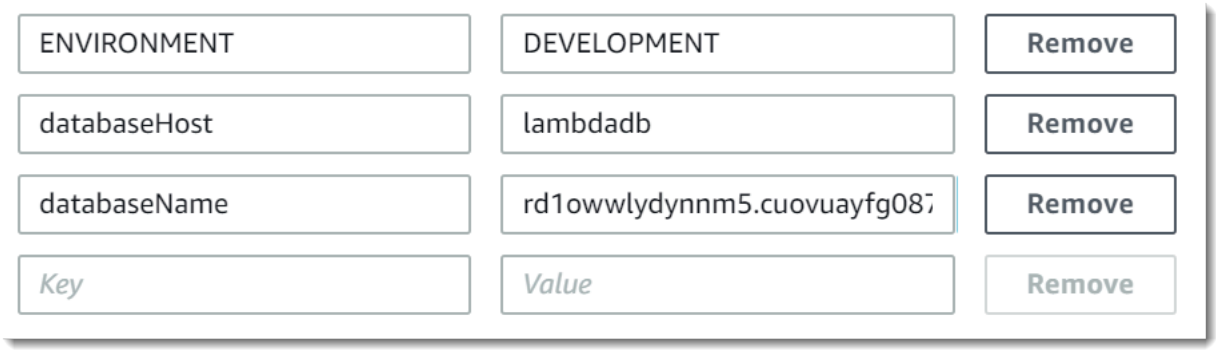
- PATH - 実行パス (/usr/local/bin:/usr/bin/:/bin:/opt/bin)。
- LD_LIBRARY_PATH - システムライブラリのパス (/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib)。
- NODE_PATH - (Node.js) Node.js ライブラリのパス (/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules)。
- PYTHONPATH - (Python 2.7、3.6、3.8) Python ライブラリのパス ($LAMBDA_RUNTIME_DIR)。
- GEM_PATH - (Ruby) Ruby ライブラリのパス ($LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0)。
- AWS_XRAY_CONTEXT_MISSING - X-Rayトレースの場合、Lambda は X-Ray SDK からランタイムエラーがスローされないように、これを LOG_ERROR に設定します。
- AWS_XRAY_DAEMON_ADDRESS - X-Ray トレーシングの場合、X-Ray デーモンの IP アドレスとポート。
- AWS_LAMBDA_DOTNET_PREJIT - .NET 6 と .NET 7 ランタイムの場合、この変数を設定して、.NET 固有のランタイムの最適化を有効または無効にします。値には always、never、および provisioned-concurrency があります。詳細については、「関数に対するプロビジョニングされた同時実行数の設定」を参照してください。
- TZ - 環境のタイムゾーン (UTC)。実行環境は、システムクロックを同期するために NTP を使用します。

表示されるサンプル値は、最新のランタイムを反映しています。特定の変数やその値の有無は、以前のランタイムでは異なる場合があります。

環境変数のシナリオ例

環境変数を使用して、テスト環境および本番環境における関数の動作をカスタマイズできます。例えば、同じコードでも設定が異なる 2 つの関数を作成できます。1 つの関数はテストデータベースに接続し、もう 1 つはプロダクションデータベースに接続します。この状況では、環境変数を使用して、データベースのホスト名とその他の接続に関する詳細を関数に渡します。

次の例は、データベースホストとデータベース名を環境変数として定義する方法を示しています。



テスト環境で本番環境よりも多くのデバッグ情報を生成する場合は、環境変数を設定して、より詳細なログ記録またはトレースを使用するようにテスト環境を設定できます。

Lambda 環境変数の保護

環境変数の保護する場合、サーバー側の暗号化を使用して保管中のデータを保護し、クライアント側の暗号化を使用して転送中のデータを保護することができます。

> **Note**
>
> データベースのセキュリティを強化するには、環境変数の代わりに AWS Secrets Manager を使用してデータベースの認証情報を保存することをお勧めします。詳細については、「Amazon RDS で AWS Lambda 使用する」を参照してください。

保管時のセキュリティ

Lambda は、AWS KMS key で常にサーバー側の暗号化を提供します。デフォルトでは、Lambda は AWS マネージドキーを使用します。このデフォルトの動作がワークフローに適している場合は、他の設定をする必要はありません。Lambda はアカウントに AWS マネージドキーを作成し、それに対するアクセス許可を管理します。このキーの使用に対する AWS の請求は発生しません。

必要に応じて、AWS KMS カスタマー管理のキーを使用することもできます。その場合、KMS キーのローテーションの制御や、KMS キーの管理に関する組織の要件への準拠を行うことができます。カスタマー管理のキーを使用すると、KMS キーへのアクセス許可があるアカウントのユーザーのみが、関数の環境変数を表示または管理できます。

カスタマーマネージドキーには、標準の AWS KMS 料金が発生します。詳細については、「AWS Key Management Service 料金表」を参照してください。

転送中のセキュリティ

セキュリティを強化するために、転送中の暗号化のヘルパーを有効にして、転送中の保護のために環境変数をクライアント側で暗号化することができます。

環境変数の暗号化を設定するには

1. AWS Key Management Service (AWS KMS) を使用して、Lambda でサーバー側およびクライアント側の暗号化に使用するカスタマー管理のキーを作成します。詳細については、 AWS Key Management Service デベロッパーガイドの「キーの作成」を参照してください。
2. Lambda コンソールを使用して、[環境変数の編集] ページに移動します。
   a. Lambda コンソールの [関数ページ] を開きます。
   b. 関数を選択します。
   c. [設定] を選択し、左側のナビゲーションバーで [環境変数] を選択します。
   d. [環境変数] セクションで、[編集] を選択します。
   e. [暗号化設定] を展開します。
3. コンソール暗号化ヘルパーを有効にして、クライアント側の暗号化を使用し、転送中のデータを保護します (オプション)。
   a. [転送時の暗号化] で、[転送時の暗号化に使用するヘルパーの有効化] を選択します。
   b. コンソール暗号化ヘルパーを有効にする各環境変数に対して、環境変数の横にある [Encrypt] (暗号化) を選択します。
   c. 転送時に暗号化する AWS KMS key で、この手順の最初で作成したカスタマー管理キーを選択します。
   d. [実行ロールポリシー] をクリックしてポリシーをコピーします。このポリシーは、環境変数を復号化するアクセス許可を関数の実行ロールに付与します。

      このポリシーは、この手順の最後のステップで使用するために保存します。
   e. 環境変数を暗号化する関数にコードを追加します。例を表示するには、[Decrypt secrets snippet] を選択します。
4. 保管中の暗号化に使用するカスタマーマネージドキーを指定します (オプション)。
   a. [カスタマーマスターキーの使用] を選択します。
   b. この手順の最初に作成したカスタマー管理のキーを選択します。
5. [Save] を選択します。

6. 許可を設定します。

   サーバー側の暗号化でカスタマーマネージドキーを使用している場合は、関数の環境変数を表示または管理できるようにしたいユーザーまたはロールに許可を付与します。詳細については、「サーバー側の暗号化 KMS キーに対するアクセス許可の管理」を参照してください。

   転送時のセキュリティの目的でクライアント側の暗号化を有効にする場合、関数に `kms:Decrypt` API オペレーションを呼び出すためのアクセス許可が必要です。この手順で以前に保存したポリシーを関数の実行ロールに追加します。

サーバー側の暗号化 KMS キーに対するアクセス許可の管理

ユーザーや関数の実行ロールには、デフォルトの暗号化キーを使用するための AWS KMS アクセス許可が不要です。カスタマー管理のキーを使用するには、キーを使用するためのアクセス許可が必要です。Lambda はユーザーのアクセス許可を使用して、キーを付与します。これにより、Lambda はこのキーを暗号化に使用できます。

- `kms:ListAliases` - Lambda コンソールでキーを表示します。
- `kms:CreateGrant`、`kms:Encrypt` - 関数でカスタマー管理のキーを設定します。
- `kms:Decrypt` - カスタマー管理のキーで暗号化された環境変数を表示および管理します。

これらの許可は、AWS アカウントから、またはキーのリソースベースの許可ポリシーから取得できます。`ListAliases` は、Lambda のマネージドポリシーから提供されます。キーポリシーは、キーユーザーグループのユーザーに対して残りのアクセス許可を付与します。

`Decrypt` アクセス許可を持たないユーザーは、引き続き関数を管理できますが、Lambda コンソールで環境変数を表示または管理することはできません。ユーザーが環境変数を表示できないようにするには、デフォルトキー、カスタマー管理キー、またはすべてのキーへのアクセスを拒否するステートメントをユーザーのアクセス許可に追加します。

Example IAM ポリシー - キー ARN によるアクセスの拒否

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
```

```
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-xmpl-4be4-
bc9d-0405a71945cc"
        }
    ]
}
```

キーのアクセス許可の管理についての詳細は、AWS Key Management Service デベロッパーガイドの「AWS KMS でのキーポリシー」を参照してください。

Lambda 環境変数の取得

関数コード内の環境変数を取得するには、プログラミング言語の標準メソッドを使用します。

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> 場合によっては、次の形式の使用が必要になる場合があります。
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda は、環境変数を保管時に暗号化して安全に保存します。別の暗号化キーを使用したり、クライアント側で環境変数値を暗号化したり、AWS CloudFormation テンプレートで AWS Secrets Manager を使用して環境変数を設定したりするように Lambda を設定できます。

Lambda 関数に Amazon VPC 内のリソースへのアクセスを許可する

Amazon Virtual Private Cloud (Amazon VPC) を使用すると、AWS アカウント にプライベートネットワークを作成して、Amazon Elastic Compute Cloud (Amazon EC2) インスタンス、Amazon Relational Database Service (Amazon RDS) インスタンス、Amazon ElastiCache インスタンスなどのリソースをホストできます。リソースを含むプライベートサブネットを介して関数を VPC にアタッチすることで、Lambda 関数に Amazon VPC でホストされているリソースへのアクセスを許可できます。Lambda コンソール、AWS Command Line Interface (AWS CLI) 、または AWS SAM を使用して Lambda 関数を Amazon VPC にアタッチするには、次のセクションの手順に従います。

> **Note**
>
> すべての Lambda 関数は、Lambda サービスによって所有および管理されている VPC 内で実行されます。これらの VPC は Lambda によって自動的に管理され、顧客には表示されません。Amazon VPC 内の他の AWS リソースにアクセスするように関数を設定しても、関数が内部で実行される Lambda が管理する VPC には影響しません。

セクション

必要な IAM 許可

Lambda 関数をAWS アカウント の Amazon VPC にアタッチするには、Lambda が VPC 内のリソースへのアクセスを関数に許可するために使用するネットワークインターフェイスを作成および管理するためのアクセス許可が必要です。

Lambda が作成するネットワークインターフェイスは、Hyperplane Elastic Network Interface または Hyperplane ENI と呼ばれます。これらのネットワークインターフェイスの詳細については、「the section called “Elastic Network Interface (ENI) について”」を参照してください。

AWS 管理ポリシー AWSLambdaVPCAccessExecutionRole を関数の実行ロールにアタッチすることで、関数に必要なアクセス許可を付与できます。Lambda コンソールで新しい関数を作成して VPC にアタッチすると、Lambda は自動的にこのアクセス許可ポリシーを追加します。

独自の IAM アクセス許可ポリシーを作成する場合は、次のアクセス許可をすべて追加してください。

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces - このアクションは、すべてのリソースで許可されている場合にのみ機能します (`"Resource": "*"`)。
- ec2:DescribeSubnets
- ec2:DeleteNetworkInterface - 実行ロール内の [DeleteNetworkInterface] でリソース ID を指定しない場合、関数から VPC へのアクセスができなくなる場合があります。ここでは、一意のリソース ID を指定するか、すべてのリソース ID を含めます (例: `"Resource": "arn:aws:ec2:us-west-2:123456789012:*/*"`)。
- ec2:AssignPrivateIpAddresses
- ec2:UnassignPrivateIpAddresses

関数のロールにこれらのアクセス許可が必要なのは、関数を呼び出すためではなく、ネットワークインターフェイスを作成するためだけであることに注意してください。関数の実行ロールからこれらのアクセス許可を削除しても、関数が Amazon VPC にアタッチされたときにその関数を正常に呼び出すことができます。

関数を VPC にアタッチするには、Lambda も IAM ユーザーロールを使用してネットワークリソースを検証する必要があります。ユーザー ロールに次の IAM アクセス許可があることを確認してください。

- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs

> **Note**
>
> 関数の実行ロールに付与する Amazon EC2 許可は、関数を VPC にアタッチするために Lambda サービスが使用します。ただし、これらのアクセス許可を関数のコードに暗黙的に付与することになります。これは、関数コードがこれらの Amazon EC2 API 呼び出しを実行できることを意味します。セキュリティのベストプラクティスに従うためのアドバイスについては、「the section called “セキュリティに関するベストプラクティス”」を参照してください。

AWS アカウント の Amazon VPC への Lambda 関数のアタッチ

Lambda コンソール、AWS アカウント、または AWS CLI を使用して、AWS SAM の Amazon VPC に関数をアタッチします。AWS CLI または AWS SAM を使用している場合、または Lambda コンソールを使用して既存の関数を VPC にアタッチする場合は、関数の実行ロールに前のセクションに記載されている必要なアクセス許可があることを確認してください。

Lambda 関数は、ハードウェア専有インスタンスのテナンシーを使用して VPC に直接接続することはできません。専有 VPC のリソースに接続するには、デフォルトのテナンシーで 2 番目の VPC にピア接続します。

Lambda console

作成時に Amazon VPC に関数をアタッチするには

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [基本的な情報] の [関数名] に、関数の名前を入力します。
3. 次の手順を実行して、関数の VPC 設定を行います。

    a. [Advanced settings (詳細設定)] を展開します。

    b. [VPC を有効化] を選択し、次に関数をアタッチする VPC を選択します。

    c. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。

    d. ネットワークインターフェイスを作成するサブネットとセキュリティグループを選択します。[デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

> **Note**
>
> プライベートリソースにアクセスするには、関数をプライベートサブネットに接続します。関数にインターネットアクセスが必要な場合、「the section called “VPC 関数のインターネットアクセス”」を参照してください。関数をパブリックサブネットに接続しても、インターネットアクセスやパブリック IP アドレスは提供されません。

4. [Create function (関数の作成)] を選択します。

既存の関数を Amazon VPC にアタッチするには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [設定] タブを選択し、次に [VPC] を選択します。
3. [編集] を選択します。
4. VPC で、関数をアタッチする Amazon VPC を選択します。
5. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。
6. ネットワークインターフェイスを作成するサブネットとセキュリティグループを選択します。[デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

> **Note**
>
> プライベートリソースにアクセスするには、関数をプライベートサブネットに接続します。関数にインターネットアクセスが必要な場合、「the section called “VPC 関数のインターネットアクセス”」を参照してください。関数をパブリックサブネットに接続しても、インターネットアクセスやパブリック IP アドレスは提供されません。

7. [Save] を選択します。

AWS CLI

作成時に Amazon VPC に関数をアタッチするには

- Lambda 関数を作成して VPC にアタッチするには、次の CLI `create-function` コマンドを実行します。

```
aws lambda create-function --function-name my-function \
--runtime nodejs20.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

独自のサブネットとセキュリティグループを指定し、ユースケースに応じて `Ipv6AllowedForDualStack` を `true`または `false` に設定します。

既存の関数を Amazon VPC にアタッチするには

- 既存の関数を VPC にアタッチするには、次の CLI `update-function-configuration` コマンドを実行します。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

VPC から関数のアタッチを解除するには

- VPC から関数をアタッチ解除するには、VPC サブネットとセキュリティグループの空のリストを使用して次の `update-function-configuration` CLI コマンドを実行します。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

関数を VPC にアタッチするには

- Lambda 関数を Amazon VPC にアタッチするには、次のテンプレート例に示すように、関数定義に `VpcConfig` プロパティを追加します。このプロパティの詳細については、「AWS CloudFormationユーザーガイド」の「AWS::Lambda::Function VpcConfig」を参照してください (AWS SAM`VpcConfig`プロパティは AWS CloudFormation`AWS::Lambda::Function`リソースの `VpcConfig` プロパティに直接渡されます) 。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC

  MySubnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.1.0/24
```

```
  MySubnet2:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: 10.0.2.0/24

  MyVPC:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
```

AWS SAM で VPC を設定する方法の詳細については、「AWS CloudFormation User Guide」の「AWS::EC2::VPC」を参照してください。

VPC にアタッチされたときのインターネットアクセス

デフォルトでは、Lambda 関数はインターネットにアクセスできます。関数を VPC にアタッチすると、関数でアクセスできるのは、その VPC 内で利用可能なリソースのみになります。関数にインターネットへのアクセスを許可するには、インターネットにアクセスできるように VPC を設定する必要があります。詳細については、「the section called “VPC 関数のインターネットアクセス”」を参照してください。

Amazon VPC で Lambda を使用するためのベストプラクティス

Lambda VPC 設定がベストプラクティスガイドラインに準拠するには、次のセクションのアドバイスに従ってください。

セキュリティに関するベストプラクティス

Lambda 関数を VPC にアタッチするには、関数の実行ロールに複数の Amazon EC2 アクセス許可を付与する必要があります。これらのアクセス許可は、関数が VPC 内のリソースにアクセスするために使用するネットワークインターフェイスを作成するために必要です。ただし、これらのアクセス許可は関数のコードにも暗黙的に付与されます。つまり、関数コードにはこれらの Amazon EC2 API コールを行うアクセス許可が付与されます。

最小特権アクセスの原則に従うには、次の例のような拒否ポリシーを関数の実行ロールに追加します。このポリシーにより、関数が VPC に関数をアタッチするために Lambda サービスが使用する Amazon EC2 API を呼び出すことができなくなります。

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses",
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS は、VPC のセキュリティを強化するために、セキュリティグループとアクセスコントロールリスト (ACL) を提供します。セキュリティグループは、リソースのインバウンドトラフィックとアウトバウンドトラフィックをコントロールします。ネットワーク ACL は、サブネットのインバウンドトラフィックとアウトバウンドトラフィックをコントロールします。セキュリティグループは、ほとんどのサブネットに対して十分なアクセス制御を提供します。VPC に追加のセキュリティレイヤーが必要な場合は、ネットワーク ACL を使用できます。Amazon VPC を使用する際のセキュリティのベスト プラクティスに関する一般的なガイドラインについては、「Amazon Virtual Private Cloud ユーザーガイド」の「VPC のセキュリティのベスト プラクティス」を参照してください。

パフォーマンスに関するベストプラクティス

関数を VPC にアタッチすると、Lambda は接続に使用できるネットワークリソース (Hyperplane ENI) があるかどうかを確認します。Hyperplane ENI は、セキュリティグループと VPC サブネットの特定の組み合わせに関連付けられています。VPC にすでに 1 つの関数をアタッチしている場合、別の関数をアタッチするときに同じサブネットとセキュリティグループを指定すると、Lambda はネットワークリソースを共有でき、新しい Hyperplane ENI を作成する必要がなく

なります。Hyperplane ENI とそのライフサイクルの詳細については、「the section called “Elastic Network Interface (ENI) について”」を参照してください。

Elastic Network Interface (ENI) について

Hyperplane ENI は、Lambda 関数と関数に接続するリソースとの間のネットワークインターフェイスとして機能するマネージドリソースです。Lambda サービスは、関数を VPC にアタッチするときに、これらの ENI を自動的に作成および管理します。

Hyperplane ENI は直接表示されないため、設定や管理を行う必要はありません。ただし、関数がどのように機能するかを知っておくと、VPC にアタッチする際の関数の動作を理解するのに役立ちます。

特定のサブネットとセキュリティグループの組み合わせを使用して VPC に関数を初めてアタッチすると、Lambda は Hyperplane ENI を作成します。同じサブネットとセキュリティグループの組み合わせを使用するアカウント内の他の関数も、この ENI を使用できます。Lambda は可能な限り既存の ENI を再利用して、リソースの使用率を最適化し、新しい ENI の作成を最小限に抑えます。各 Hyperplane ENI は最大 65,000 個の接続/ポートをサポートします。接続数がこの制限を超えると、Lambda はネットワークトラフィックと同時実行要件に基づいて ENI の数を自動的にスケーリングします。

新しい関数の場合、Lambda が Hyperplane ENI を作成している間は、関数は保留状態のままになり、呼び出されることはありません。関数は、Hyperplane ENI の準備ができた場合にのみアクティブ状態に移行します。これには数分かかる場合があります。既存の関数の場合、バージョンの作成や関数のコードの更新など、関数をターゲットとする追加のオペレーションを実行することはできませんが、関数の以前のバージョンを引き続き呼び出すことはできます。

> **Note**
>
> Lambda 関数が 30 日間アイドル状態のままである場合、Lambda は未使用の Hyperplane ENI を回収し、関数の状態をアイドルに設定します 次の呼び出しの試行は失敗し、Lambda が Hyperplane ENI の作成または割り当てを完了するまで、関数は再び保留状態になります。Lambda 関数の状態の詳細については、「the section called “関数の状態”」を参照してください。

Hyperplane ENI のライフサイクルの詳細については、「the section called “Lambda Hyperplane ENI”」を参照してください。

VPC 設定で IAM 条件キーを使用する

VPC 設定で Lambda 固有の条件キーを使用して、Lambda 関数に追加のアクセス許可コントロールを提供できます。例えば、組織内のすべての関数を VPC に接続するように要求できます。また、関数のユーザーに対して使用を許可または拒否するサブネットとセキュリティグループを指定することもできます。

Lambda は IAM ポリシーで次の条件キーをサポートしています。

- lambda:VpcIds - 1 つ以上の VPC を許可または拒否します。
- lambda:SubnetIds - 1 つ以上のサブネットを許可または拒否します。
- lambda:SecurityGroupIds - 1 つ以上のセキュリティグループを許可または拒否します。

Lambda API オペレーションの CreateFunction および UpdateFunctionConfiguration は、これらの条件キーをサポートしています。IAMポリシーでの条件キーの使用の詳細については、IAM ユーザーガイドの「IAM JSON ポリシーエレメント: 条件」を参照してください。

> **Tip**
>
> 関数に以前の API リクエストの VPC 設定が既に含まれている場合は、VPC 設定なしで `UpdateFunctionConfiguration` リクエストを送信できます。

VPC 設定の条件キーを使用したポリシーの例

以下の例は、VPC 設定で条件キーを使用する方法を示しています。必要な制限を含むポリシーステートメントを作成したら、このポリシーステートメントをターゲットの ユーザーまたはロールに追加します。

ユーザーに対して VPC に接続された関数のみをデプロイさせる

すべてのユーザーに対して VPC に接続された関数のみをデプロイさせるには、有効な VPC ID を含まない関数の作成および更新オペレーションを拒否できます。

VPC ID は `CreateFunction` リクエストまたは `UpdateFunctionConfiguration` リクエストへの入力パラメータではないことに注意してください。Lambda は、サブネットとセキュリティグループのパラメータに基づいて VPC ID 値を取得します。

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

特定の VPC、サブネット、セキュリティグループに対するユーザーアクセスを拒否する

特定の VPC へのユーザーアクセスを拒否するには、StringEquals を使用して lambda:VpcIds 条件の値を確認します。次の例では、 vpc-1 および vpc-2 へのユーザーアクセスを拒否します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceOutOfVPC",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

特定のサブネットへのユーザーアクセスを拒否するには、StringEquals を使用して lambda:SubnetIds 条件の値を確認します。次の例では、 subnet-1 および subnet-2 へのユーザーアクセスを拒否します。

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

特定のセキュリティグループへのユーザーアクセスを拒否するには、StringEquals を使用して lambda:SecurityGroupIds 条件の値を確認します。次の例では、 sg-1 および sg-2 へのユーザーアクセスを拒否します。

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

特定の VPC 設定を使用して関数を作成および更新することをユーザーに許可する

特定の VPC にアクセスすることをユーザーに許可するには、StringEquals を使用して lambda:VpcIds 条件の値を確認します。次の例では、vpc-1 および vpc-2 にアクセスすることをユーザーに許可します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceStayInSpecificVpc",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

特定のサブネットにアクセスすることをユーザーに許可するには、StringEquals を使用して lambda:SubnetIds 条件の値を確認します。次の例では、subnet-1 および subnet-2 にアクセスすることをユーザーに許可します。

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
```

```
    }
```

特定のセキュリティグループにアクセスすることをユーザーに許可するには、StringEquals を使用して lambda:SecurityGroupIds 条件の値を確認します。次の例では、sg-1 および sg-2 にアクセスすることをユーザーに許可します。

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC チュートリアル

次のチュートリアルでは、VPC 内のリソースに Lambda 関数を接続します。

- チュートリアル: Lambda 関数を使用して Amazon VPC 内の Amazon RDS にアクセスする
- チュートリアル: Amazon VPC の Amazon ElastiCache にアクセスする Lambda 関数の設定

VPC に接続された Lambda 関数にインターネットアクセスを有効にする

デフォルトでは、Lambda 関数は、インターネットにアクセスできる Lambda 管理型 VPC で実行されます。アカウントで VPC のリソースにアクセスするには、VPC 設定を関数に追加できます。これにより、VPC がインターネットにアクセスできる場合を除き、機能は VPC 内のリソースに制限されます。このページでは、VPC に接続された Lambda 関数にインターネットアクセスを提供する方法について説明します。

VPC をまだ持っていません

VPC を作成する

[VPC ワークフローの作成] は、サブネット、NAT ゲートウェイ、インターネットゲートウェイ、ルートテーブルエントリなど、Lambda 関数がプライベートサブネットからパブリックインターネットにアクセスするために必要なすべての VPC リソースを作成します。

VPC を作成するには

1. Amazon VPC コンソール (https://console.aws.amazon.com/vpc/) を開きます。
2. ダッシュボードで、[VPC を作成] を選択します。
3. [Resources to create] (作成するリソース) で、[VPC and more] (VPC など) を選択します。
4. VPC を設定する
   a. [名前タグの自動生成] に、VPC の名前を入力します。
   b. [IPv4 CIDR ブロック] で、デフォルトの候補を維持するか、アプリケーションまたはネットワークが必要とする CIDR ブロックを入力します。
   c. アプリケーションが IPv6 アドレスを使用して通信する場合は、[IPv6 CIDR ブロック]、[Amazon が提供する IPv6 CIDR ブロック] を選択します。
5. サブネットを設定する
   a. [アベイラビリティゾーンの数] で、[2] を選択します。高可用性を実現するには、少なくとも 2 つの AZ をお勧めします。
   b. [Number of public subnets] (パブリックサブネットの数) で 2 を選択します。
   c. [Number of private subnets] (プライベートサブネットの数) は、2 を選択します。

d. パブリックサブネットのデフォルトの CIDR ブロックをそのまま使用することも、[サブネット CIDR ブロックをカスタマイズする] を展開して CIDR ブロックを入力することもできます。詳細については、「サブネット CIDR ブロック」を参照してください。

6. [NAT ゲートウェイ] で [AZ ごとに 1] を選択すると、回復性が高まります。
7. [Egress 専用インターネットゲートウェイ] では、IPv6 CIDR ブロックを含める場合は [はい] を選択します。
8. [VPC エンドポイント] は、デフォルトの [S3 ゲートウェイ] のままにします。このオプションには費用はかかりません。詳細については、「Amazon S3 向け VPC エンドポイントの種類」を参照してください。
9. [DNS オプション]については、デフォルト設定のままにします。
10. [Create VPC (VPC の作成)] を選択します。

Lambda 関数を設定

関数の作成時に VPC を設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. [Create function (関数の作成)] を選択します。
3. [基本的な情報] の [関数名] に、関数の名前を入力します。
4. [Advanced settings (詳細設定)] を展開します。
5. [VPC を有効にする] を選択したら、VPC を選択します。
6. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。
7. [サブネット] では、すべてのプライベートサブネットを選択します。プライベートサブネットは、NAT ゲートウェイ経由でインターネットにアクセスできます。関数をパブリックサブネットに接続しても、インターネットにアクセスできません。

> **Note**
>
> [デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

8. [セキュリティグループ] では、アウトバウンドトラフィックを許可するセキュリティグループを選択します。
9. [Create function (関数の作成)] を選択します。

Lambda は、「AWSLambdaVPCAccessExecutionRole」 AWS 管理ポリシーをで実行ロールを自動的に作成します。このポリシーのアクセス許可は、VPC 設定の Elastic Network Interface を作成するためにのみに必要であり、関数を呼び出すためではありません。最小特権のアクセス許可を適用するには、関数および VPC 設定を作成した後に、実行ロールから [AWSLambdaVpcAccessExecutionRole] ポリシーを削除できます。詳細については、「必要な IAM 許可」を参照してください。

既存の関数に対して VPC を設定するには

既存の関数に VPC 設定を追加するには、関数の実行ロールに Elastic Network Interface を作成して管理する許可 が必要です。「AWSLambdaVPCAccessExecutionRole」 AWS 管理ポリシーには、必要な許可が含まれています。最小特権のアクセス許可を適用するには、VPC 設定を作成した後に、実行ロールから [AWSLambdaVpcAccessExecutionRole] ポリシーを削除できます。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] タブを選択したら、[VPC] を選択します。
4. [VPC] で、[編集] を選択します。
5. VPC を選択します。
6. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。
7. [サブネット] では、すべてのプライベートサブネットを選択します。プライベートサブネットは、NAT ゲートウェイ経由でインターネットにアクセスできます。関数をパブリックサブネットに接続しても、インターネットにアクセスできません。

   > **Note**
   >
   > [デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

8. [セキュリティグループ] では、アウトバウンドトラフィックを許可するセキュリティグループを選択します。
9. [Save] を選択します。

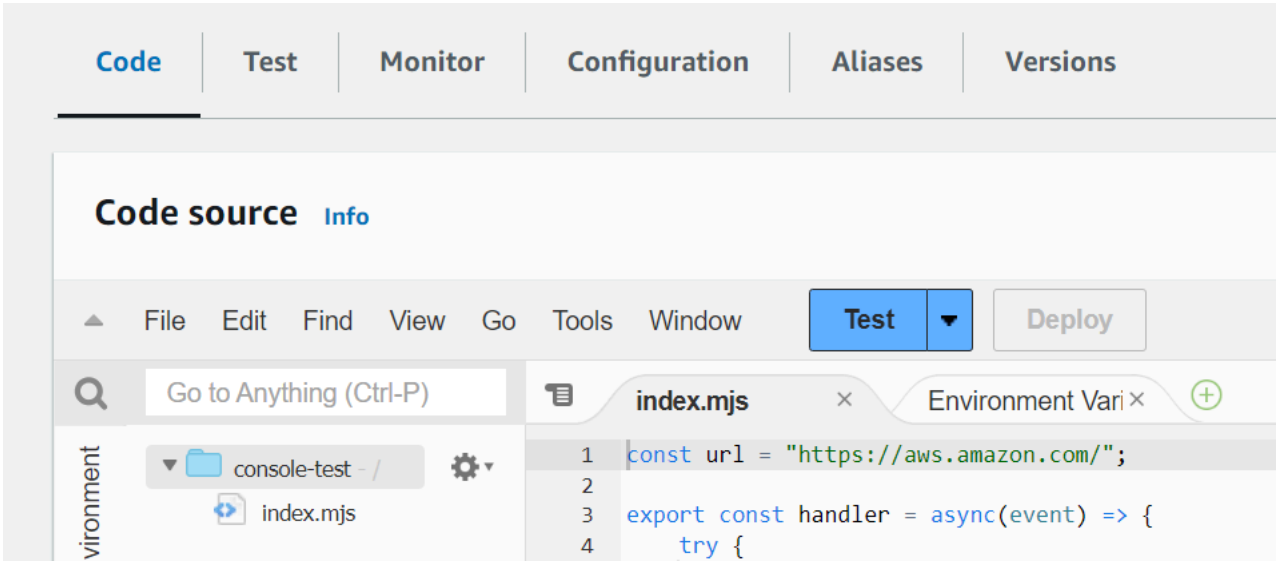
関数をテストする

次のサンプルコードを使用し、VPC に接続された関数がパブリックインターネットにアクセスできることを確認します。正常に処理された場合、コードは 200 ステータスコードを返します。失敗した場合、関数がタイムアウトします。

Node.js

この例では、nodejs18.x 以降のランタイムで利用できる fetch を使用します。

1. Lambda コンソールの [コードソース] ペインで、次のコードを [index.mjs] ファイルに貼り付けます。この関数はパブリックエンドポイントに HTTP GET リクエストを行い、HTTP レスポンスコードを返し、関数がパブリックインターネットにアクセスできるかどうかをテストします。

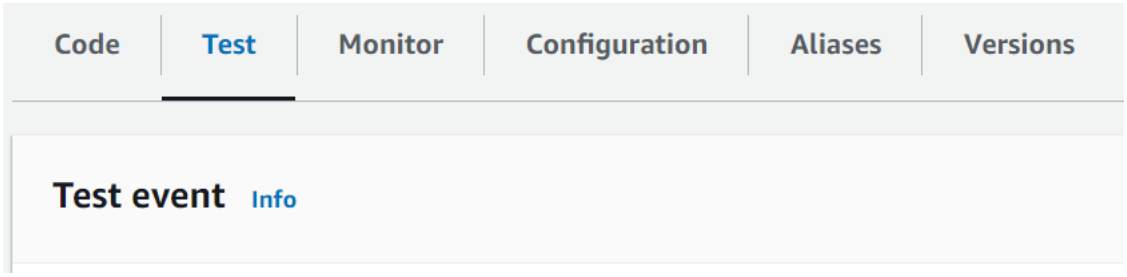


Example — async/await を持つ HTTP リクエスト

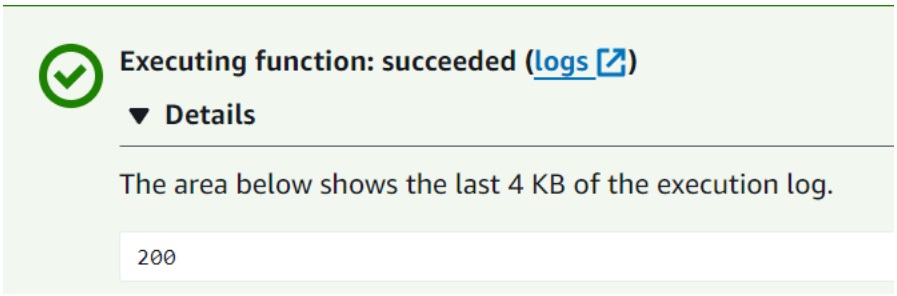
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
```

```
        return 500;
    }
};
```

2. [デプロイ] を選択します。
3. [テスト] タブを選択します。



4. [テスト] を選択します。
5. 関数は 200 ステータスコードを返します。つまり、この関数はアウトバウンドのインターネットアクセスができることを意味します。

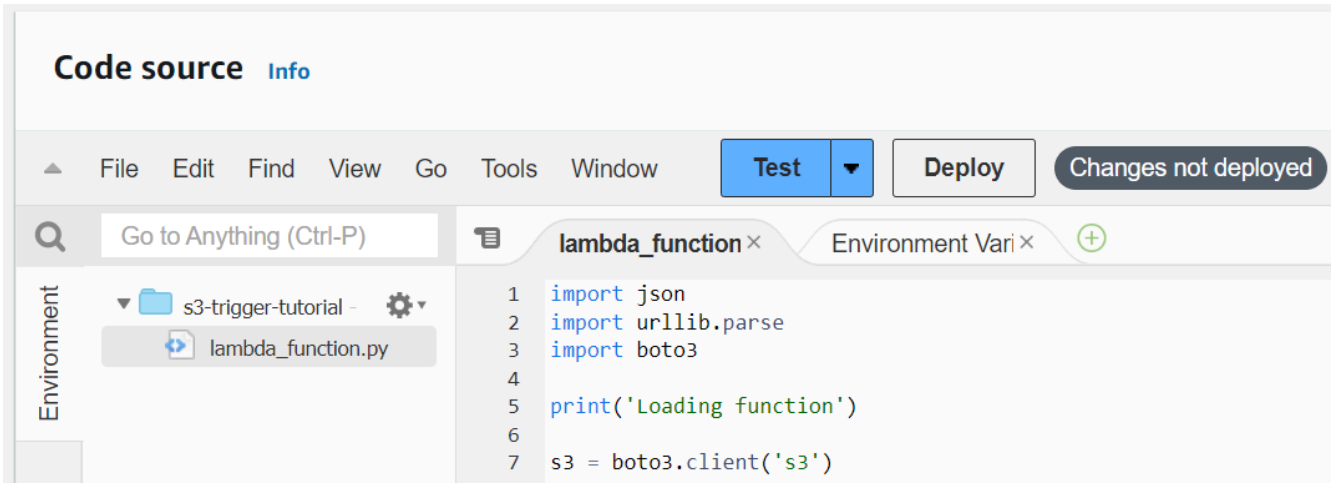


関数がパブリックインターネットにアクセスできない場合、次のようなエラーメッセージが表示されます。

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

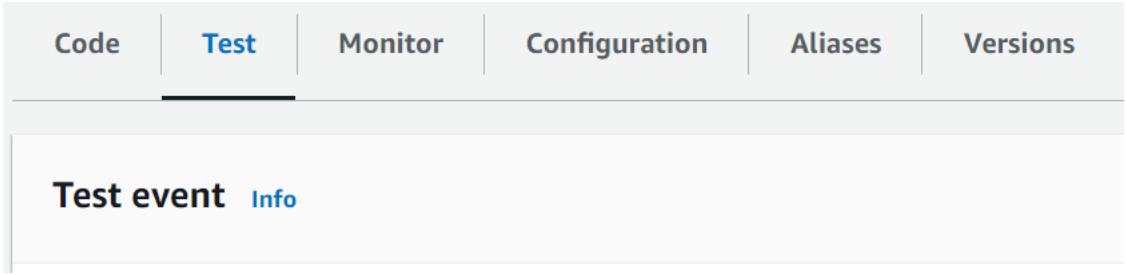
Python

1. Lambda コンソールの [コードソースペイン] ペインで、次のコードを [lambda_function.py] ファイルに貼り付けます。この関数はパブリックエンドポイントに HTTP GET リクエストを行い、HTTP レスポンスコードを返し、関数がパブリックインターネットにアクセスできるかどうかをテストします。

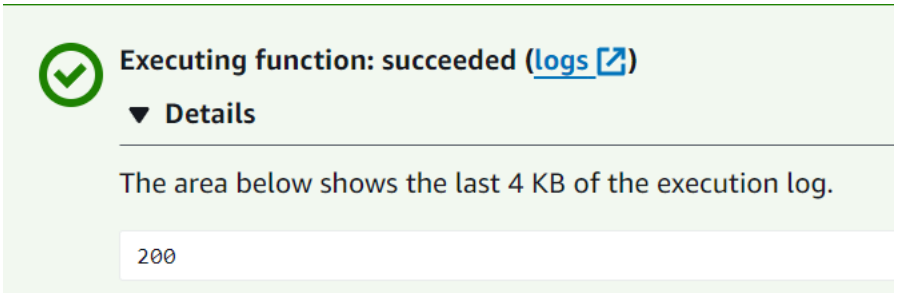



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. [デプロイ] を選択します。
3. [テスト] タブを選択します。



4. [テスト] を選択します。
5. 関数は 200 ステータスコードを返します。つまり、この関数はアウトバウンドのインターネットアクセスができることを意味します。



関数がパブリックインターネットにアクセスできない場合、次のようなエラーメッセージが表示されます。

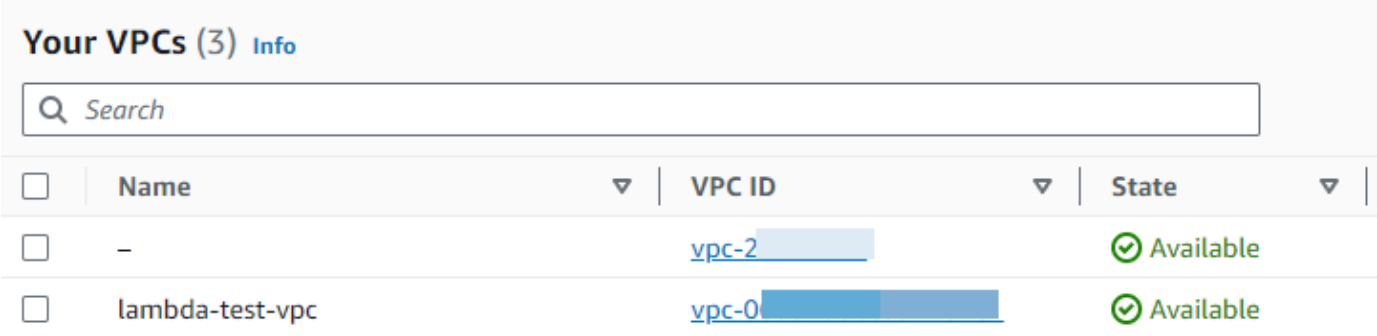
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

VPC を既に持っています

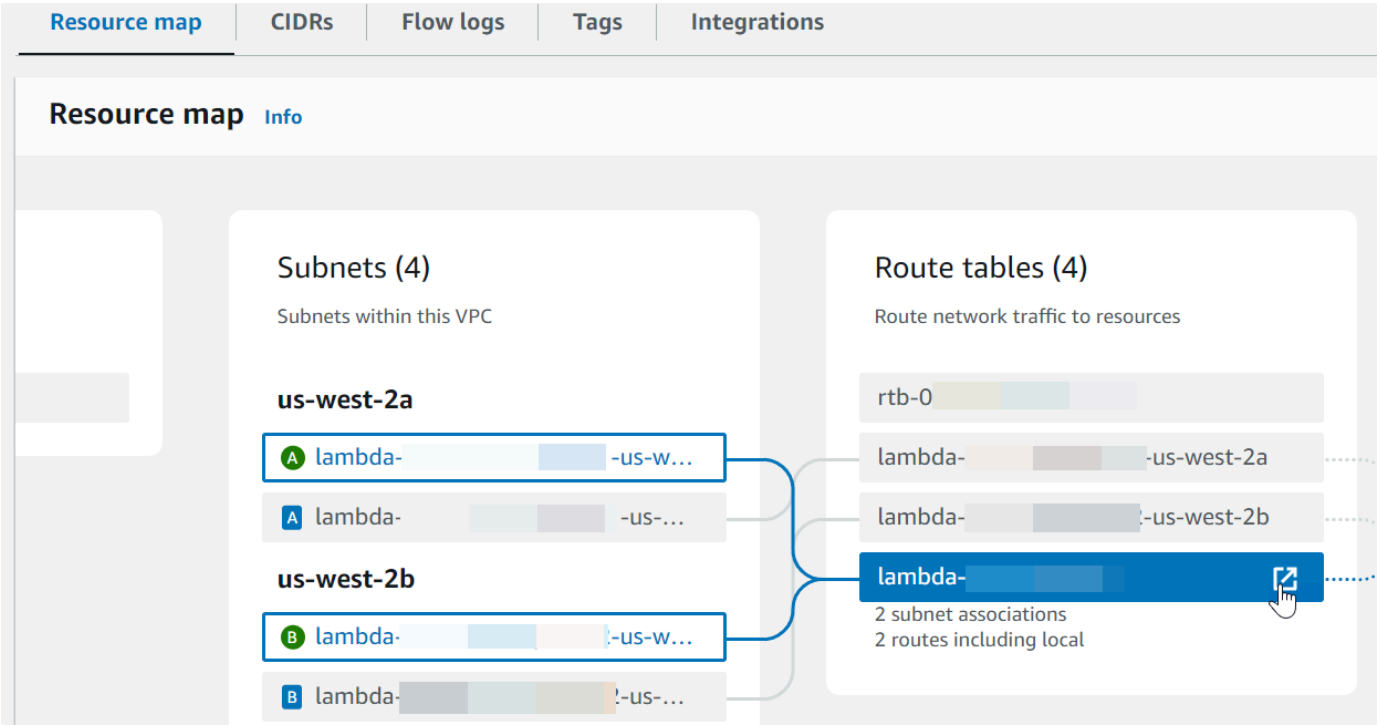
VPC は既にあるが、Lambda 関数用にパブリックインターネットアクセスを設定する必要がある場合は、次の手順に従ってください。この手順では、VPC に少なくとも 2 つのサブネットがあることを前提としています。サブネットが 2 つない場合、Amazon VPC ユーザーガイドの「サブネットを作成する」を参照してください。

ルートテーブルの設定を確認する

1. Amazon VPC コンソール (https://console.aws.amazon.com/vpc/) を開きます。
2. [VPC ID] を選択します。



3. [リソースマップ] セクションまでスクロールダウンします。ルートテーブルのマッピングをメモします。サブネットにマッピングされている各ルートテーブルを開きます。

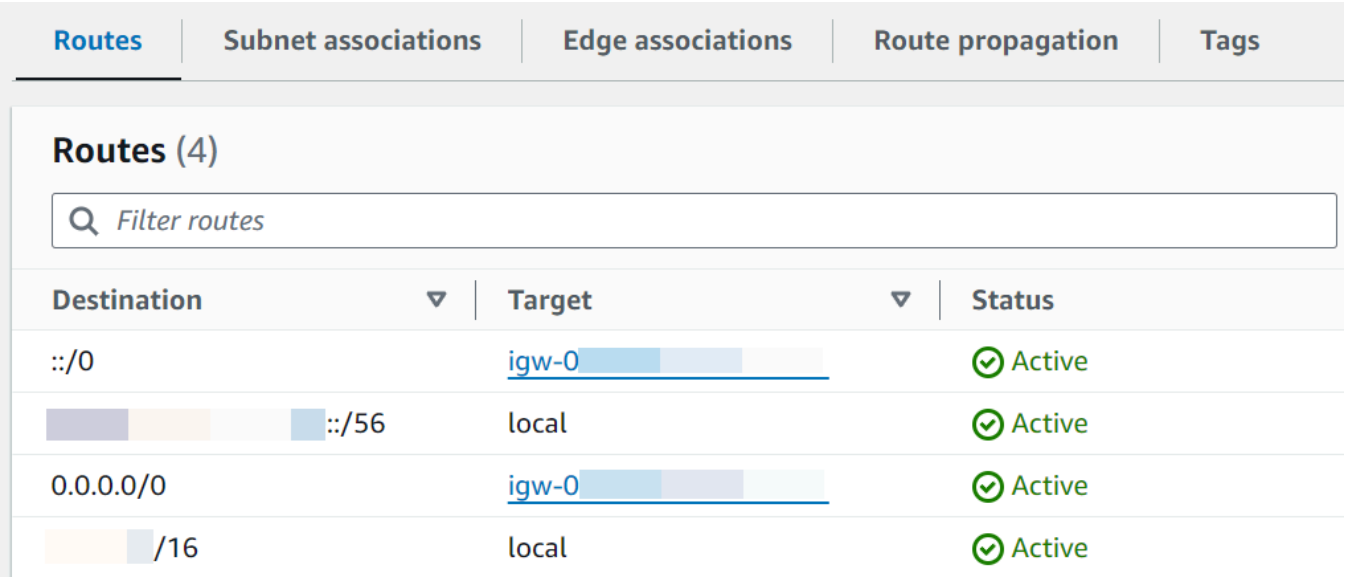


4. [ルート] タブまでスクロールダウンします。ルートを確認し、次のいずれかが正しいかどうかを判断します。この各要件は、個別のルートテーブルで満たされる必要があります。

   - インターネット向けトラフィック (IPv4 用に `0.0.0.0/0`、IPv6 用に `::/0`) は、インターネットゲートウェイ (`igw-xxxxxxxxxx`) にルーティングされます。つまり、ルートテーブルに関連付けられているサブネットはパブリックサブネットです。

     > **Note**
     >
     > サブネットに IPv6 CIDR ブロックがない場合、IPv4 ルート (`0.0.0.0/0`) のみが表示されます。

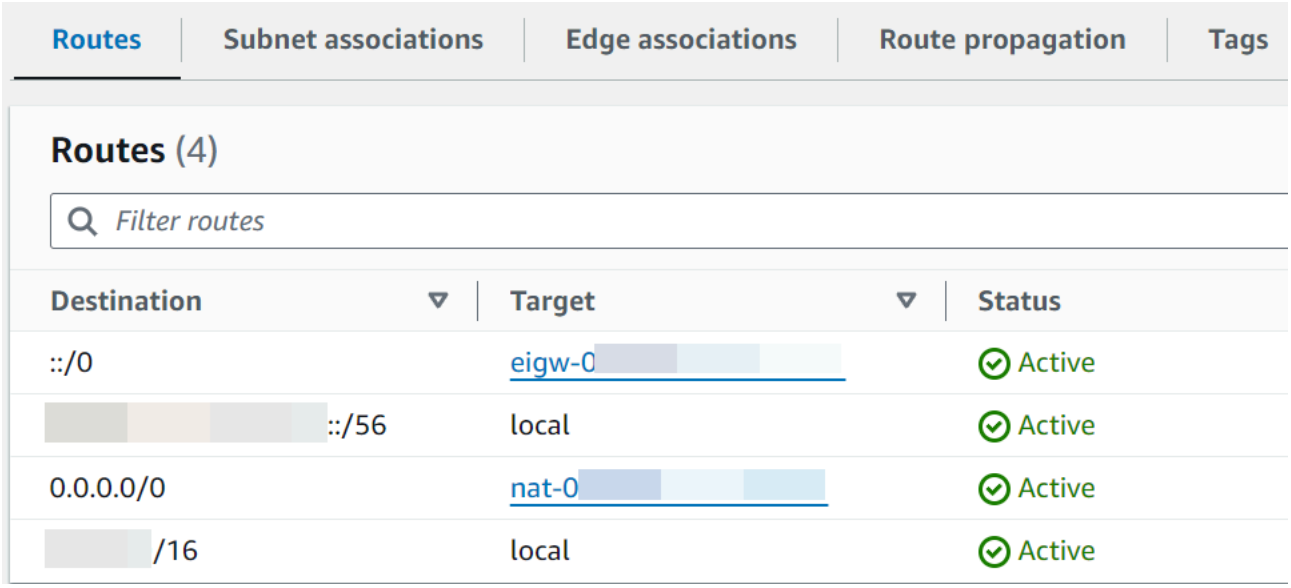
Example パブリックサブネットのルートテーブル



- IPv4 (`0.0.0.0/0`) のインターネット向けトラフィックは、パブリックサブネットに関連付けられた NAT ゲートウェイ (`nat-xxxxxxxxxx`) にルーティングされます。つまり、サブネットは NAT ゲートウェイ経由でインターネットにアクセスできるプライベートサブネットです。

> **Note**
>
> サブネットに IPv6 CIDR ブロックがある場合、ルートテーブルはインターネット向け IPv6 トラフィック (`::/0`) も Egress-Only のインターネットゲートウェイ (`eigw-xxxxxxxxxx`) にルーティングする必要があります。サブネットに IPv6 CIDR ブロックがない場合、IPv4 ルート (`0.0.0.0/0`) のみが表示されます。

Example プライベートサブネットのルートテーブル



5. VPC のサブネットに関連付けられている各ルートテーブルを確認し、インターネットゲートウェイを含むルートテーブルおよび NAT ゲートウェイを含むルートテーブルがあることを確認できるまで、前の手順を繰り返します。

   インターネットゲートウェイへのルートおよび NAT ゲートウェイへのルートで構成される 2 つのルートテーブルがない場合、次の手順に従って不足しているリソースおよびルートテーブルのエントリを作成します。

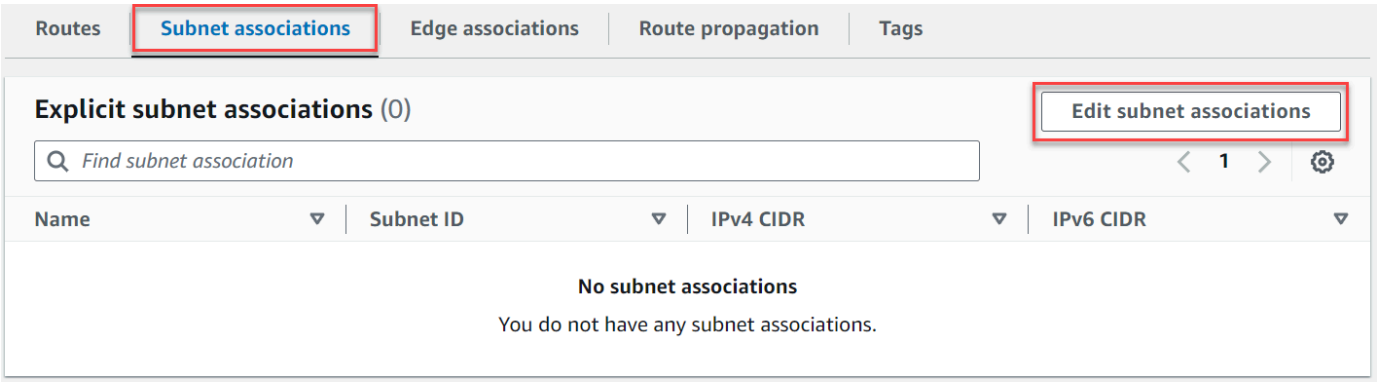
ルートテーブルの作成

次の手順に従って、ルートテーブルを作成し、サブネットに関連付けます。

Amazon VPC コンソールを使用してカスタムルートテーブルを作成する方法

1. Amazon VPC コンソール (https://console.aws.amazon.com/vpc/) を開きます。
2. ナビゲーションペインで、[Route tables] (ルートテーブル) を選択します。
3. [ルートテーブルの作成] を選択します。
4. (オプション) [Name] (名前) には、ルートテーブルの名前を入力します。
5. [VPC] で、ユーザーの VPC を選択します。
6. (オプション) タグを追加するには、[Add new tag] (新しいタグを追加) を選択し、タグキーとタグ値を入力します。

7. [ルートテーブルの作成] を選択します。
8. [Subnet Associations] (サブネットの関連付け) タブで、 [Edit subnet associations] (サブネットの関連付けの編集) を選択します。



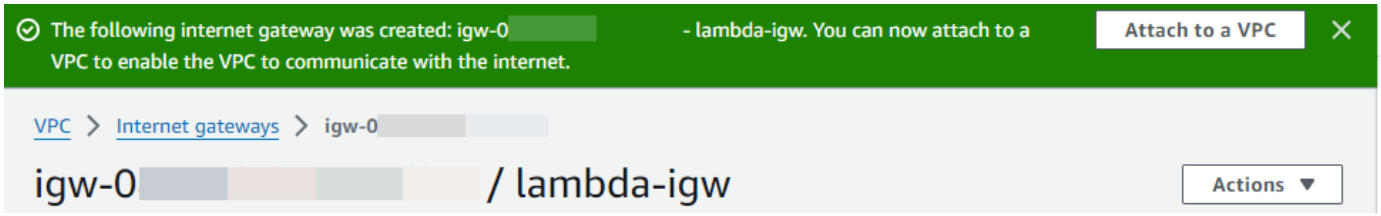
9. ルートテーブルに関連付けるサブネットのチェックボックスをオンにします。
10. [Save associations] (関連付けを保存する) を選択します。

インターネットゲートウェイを作成する

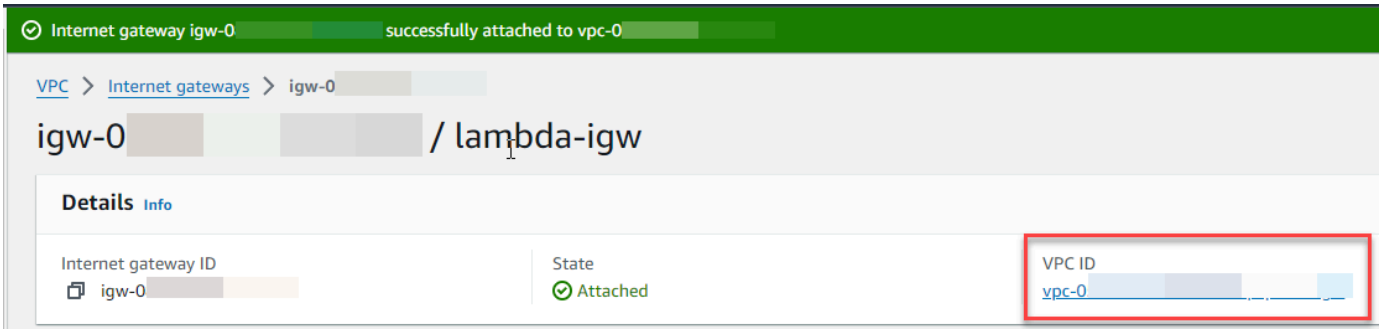
次の手順を実行してインターネットゲートウェイを作成し、VPC にアタッチしてパブリックサブネットのルートテーブルに追加します。

インターネットゲートウェイを作成するには

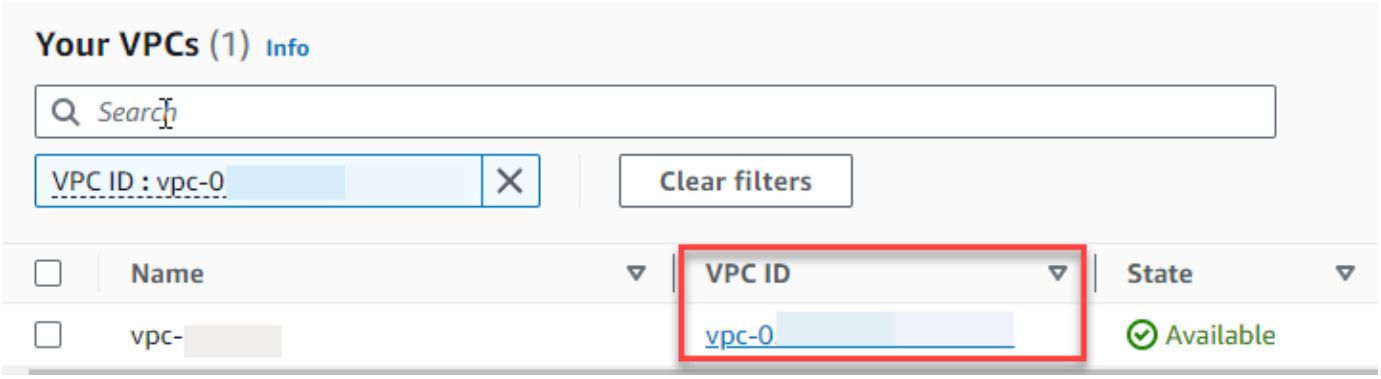
1. Amazon VPC コンソール (https://console.aws.amazon.com/vpc/) を開きます。
2. ナビゲーションペインで、[Internet gateways] (インターネットゲートウェイ) を選択します。
3. [インターネットゲートウェイの作成] を選択します。
4. (オプション) インターネットゲートウェイの名前を入力します。
5. (オプション) タグを追加するには、[Add new tag] (新しいタグを追加) を選択し、そのタグのキーと値を入力します。
6. [インターネットゲートウェイの作成] を選択します。
7. 画面上部のバナーから [VPC にアタッチ] を選択し、利用可能な VPC を選択したら、[インターネットゲートウェイをアタッチする] を選択します。



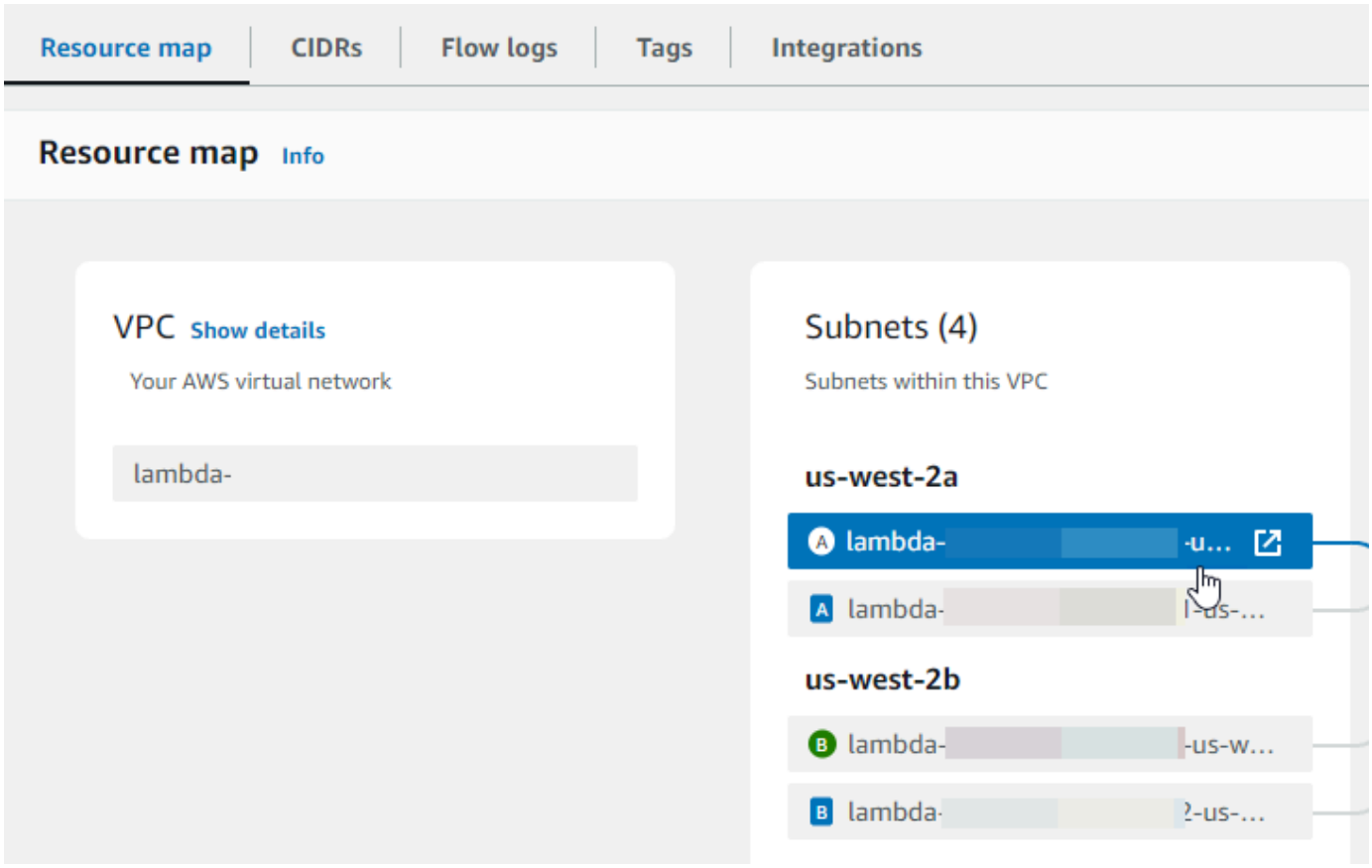
8. [VPC ID] を選択します。



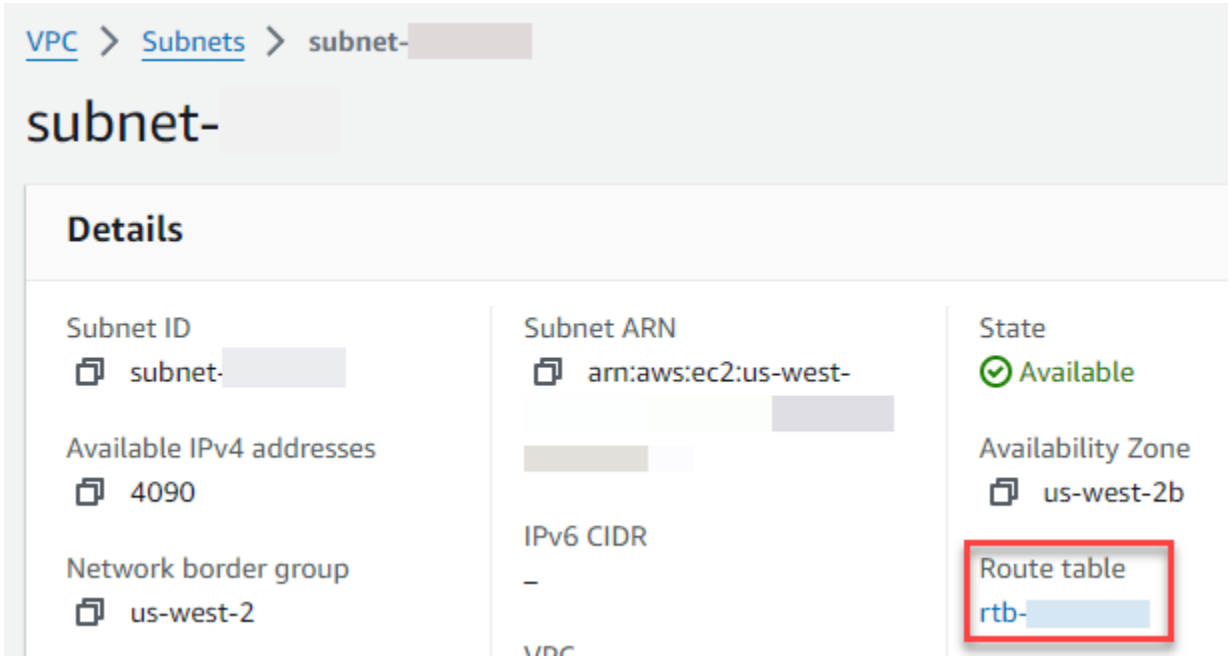
9. [VPC ID] を再び選択し、VPC 詳細ページを開きます。



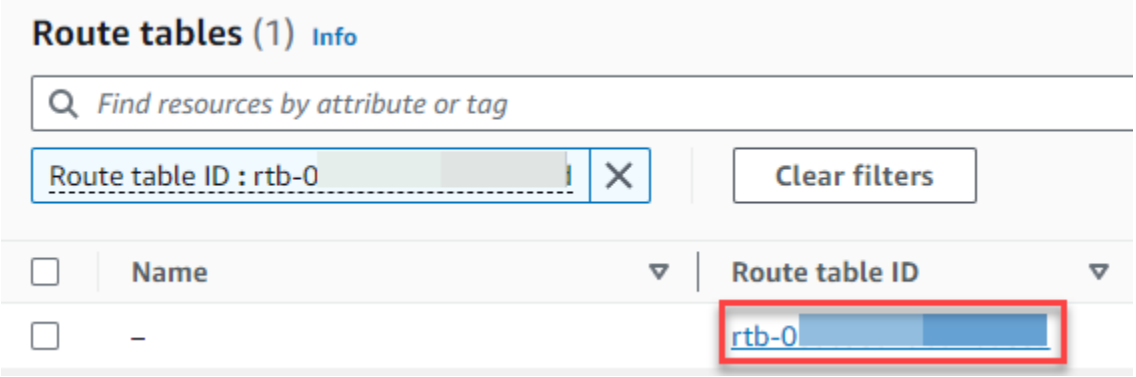
10. [リソースマップ] セクションまでスクロールダウンし、サブネットを選択します。サブネットの詳細は新しいタブに表示されます。



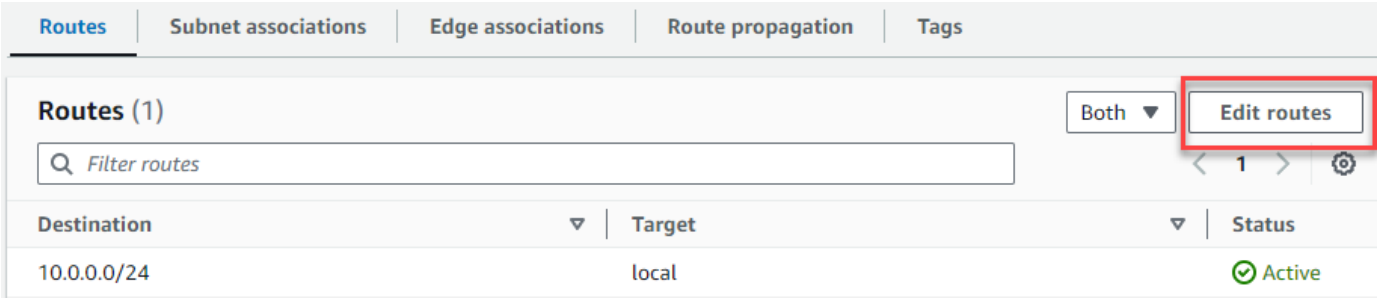
11. [ルートテーブル] のリンクを選択します。



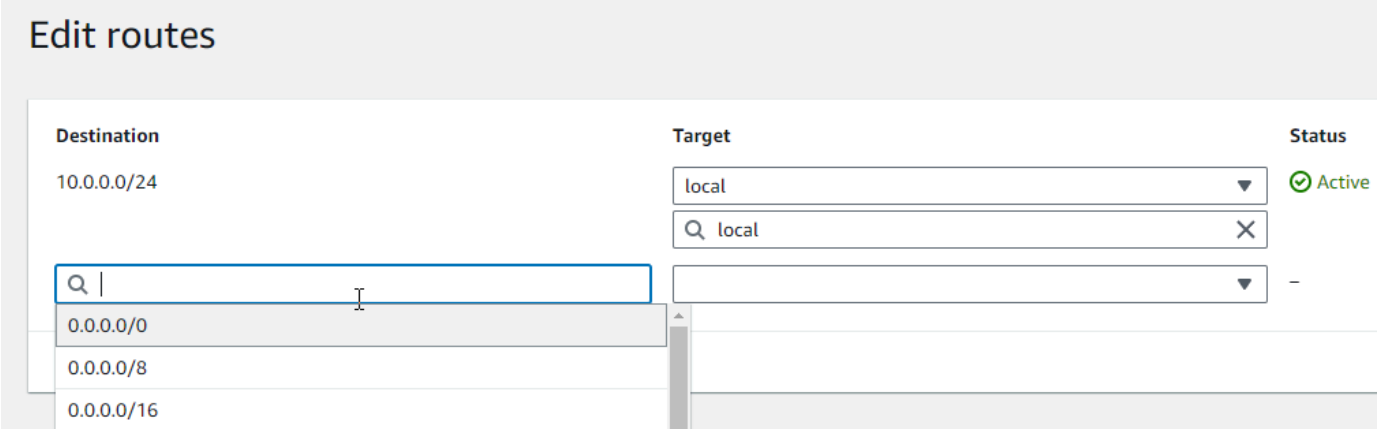
12. [ルートテーブル ID] を選択し、ルートテーブルの詳細ページを開きます。



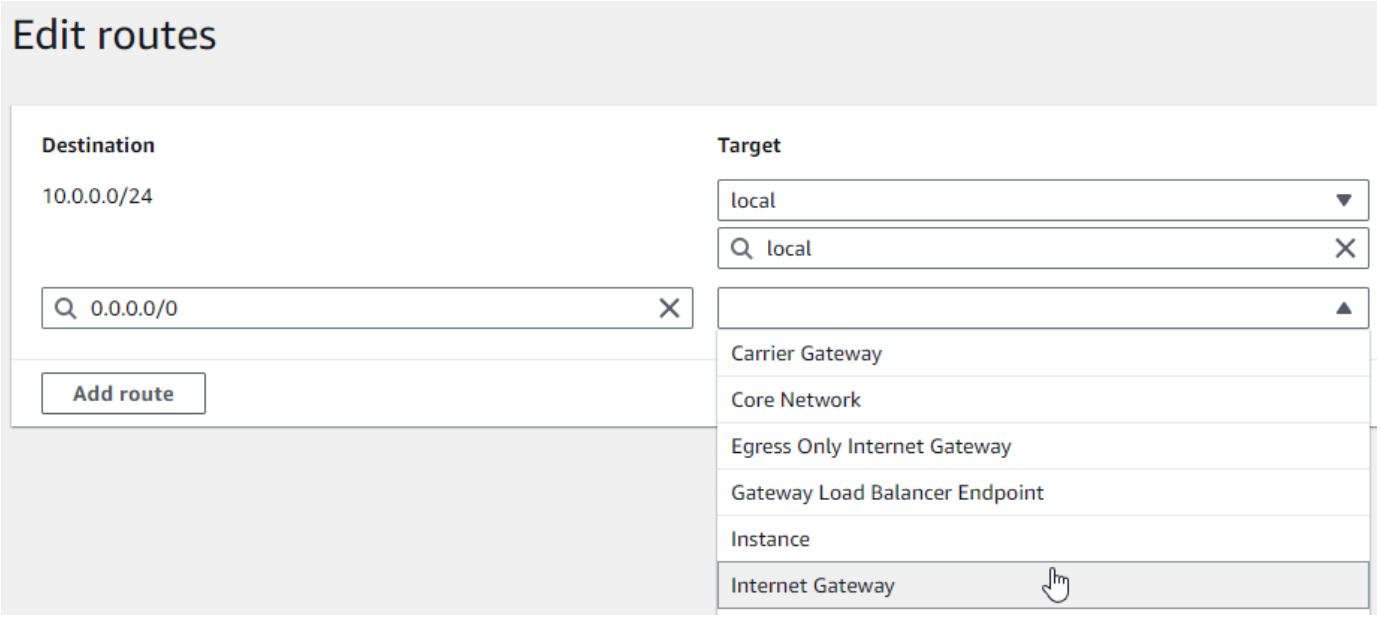
13. [ルート] で、[ルートを編集する] を選択します。



14. [ルートを追加する] を選択し、[宛先] ボックスに `0.0.0.0/0` と入力します。



15. [ターゲット]] には、[インターネットゲートウェイ] を選択し、先ほど作成したインターネットゲートウェイを選択します。サブネットに IPv6 CIDR ブロックがある場合、同じインターネットゲートウェイに `::/0` のルートも追加する必要があります。



16. [Save changes] (変更の保存) をクリックします。

NAT ゲートウェイを作成する

次の手順に従って NAT ゲートウェイを作成し、パブリックサブネットに関連付けて、プライベートサブネットのルートテーブルに追加します。

NAT ゲートウェイを作成してパブリックサブネットに関連付ける方法

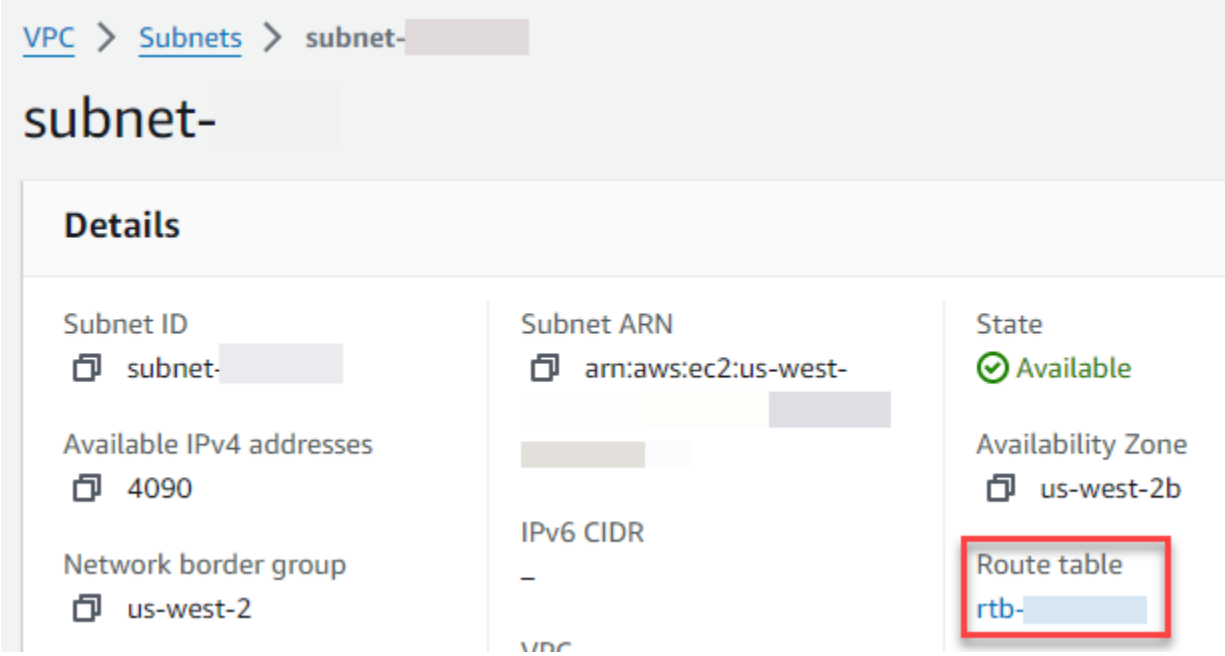
1. ナビゲーションペインで [NAT ゲートウェイ] を選択します。
2. [NAT ゲートウェイを作成] を選択します。
3. (オプション) NAT ゲートウェイの名前を入力します。
4. [サブネット] では、VPC のパブリックサブネットを選択します。(パブリックサブネットは、ルートテーブルにインターネットゲートウェイへの直接ルートを持つサブネットです。)

> **Note**
>
> NAT ゲートウェイはパブリックサブネットに関連付けられていますが、ルートテーブルのエントリはプライベートサブネットにあります。

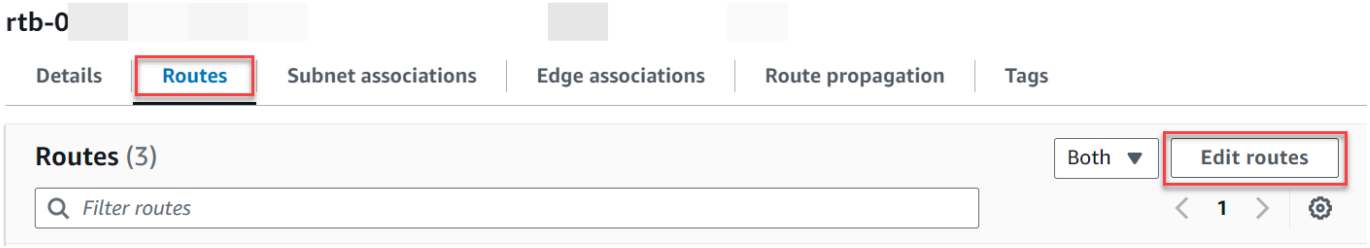
5. [Elastic IP 配分 ID] には、Elastic IP アドレスを選択するか、[Elastic IP を配分する] を選択します。
6. [NAT ゲートウェイを作成] を選択します。

プライベートサブネットのルートテーブルで NAT ゲートウェイにルートを追加する方法

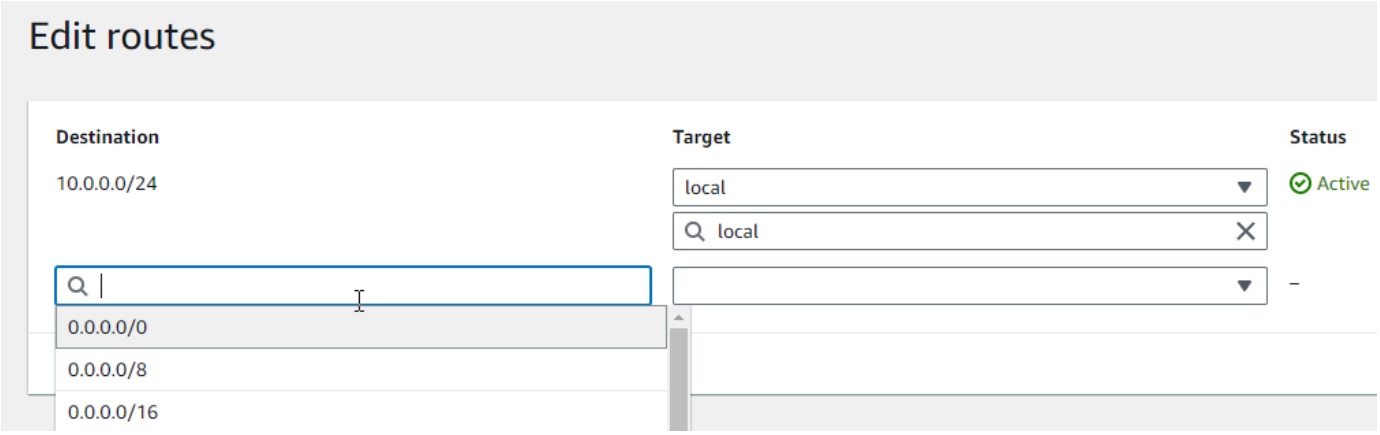
1. ナビゲーションペインで、[Subnets (サブネット)] を選択します。
2. VPC 内のプライベートサブネットを選択します。(プライベートサブネットは、ルートテーブルにインターネットゲートウェイへのルートがないサブネットです。)
3. [ルートテーブル] のリンクを選択します。



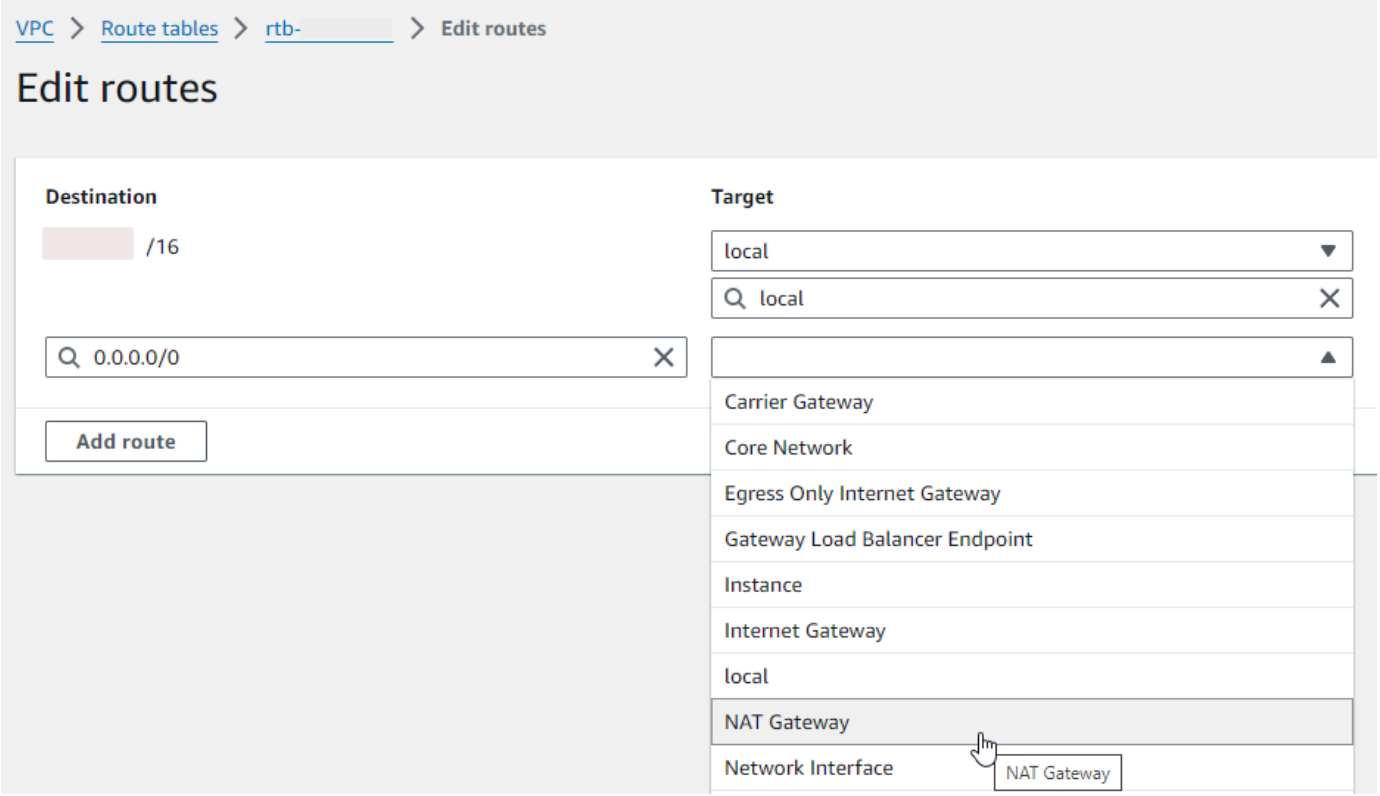
4. スクロールダウンして[ルート] タブを選択し、[ルートを編集する] を選択します。



5. [ルートを追加する] を選択し、[宛先] ボックスに `0.0.0.0/0` と入力します。



6. [ターゲット] には、[NAT ゲートウェイ] を選択し、先ほど作成した NAT ゲートウェイを選択します。



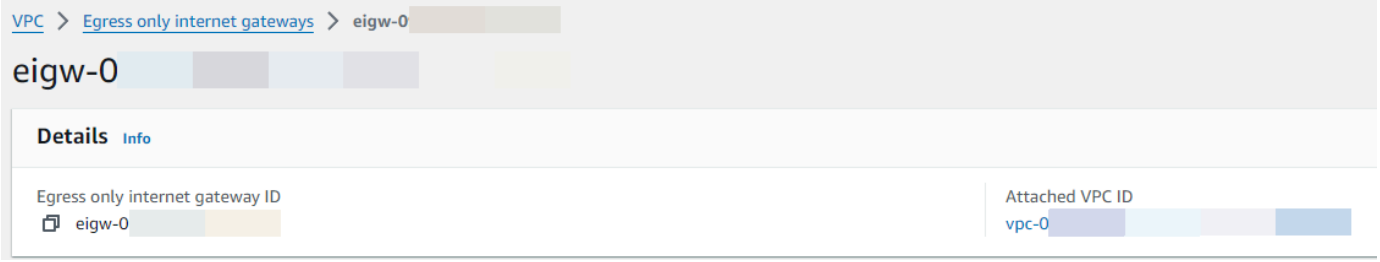
7. [Save changes] (変更の保存) をクリックします。

Egress-Only のインターネットゲートウェイ (IPv6 のみ) を作成する

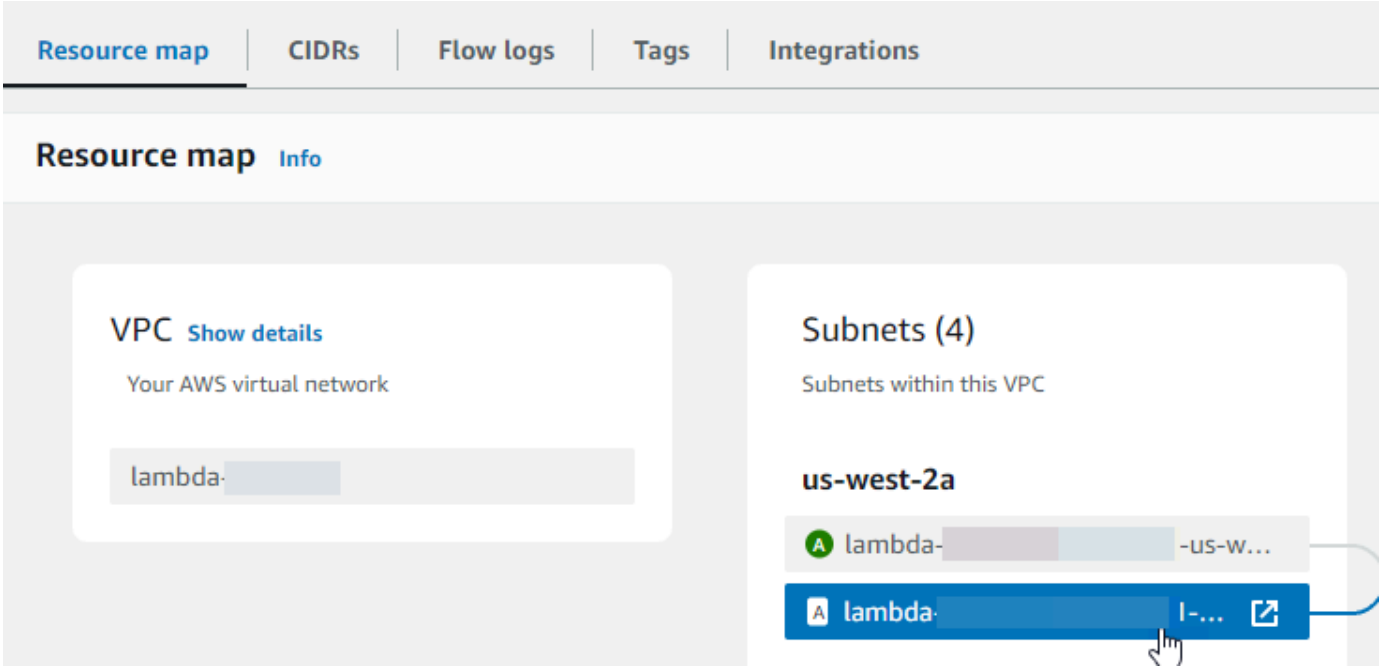
次の手順に従って Egress-Only のインターネットゲートウェイを作成し、プライベートサブネットのルートテーブルに追加します。

Egress-Only インターネットゲートウェイを作成するには

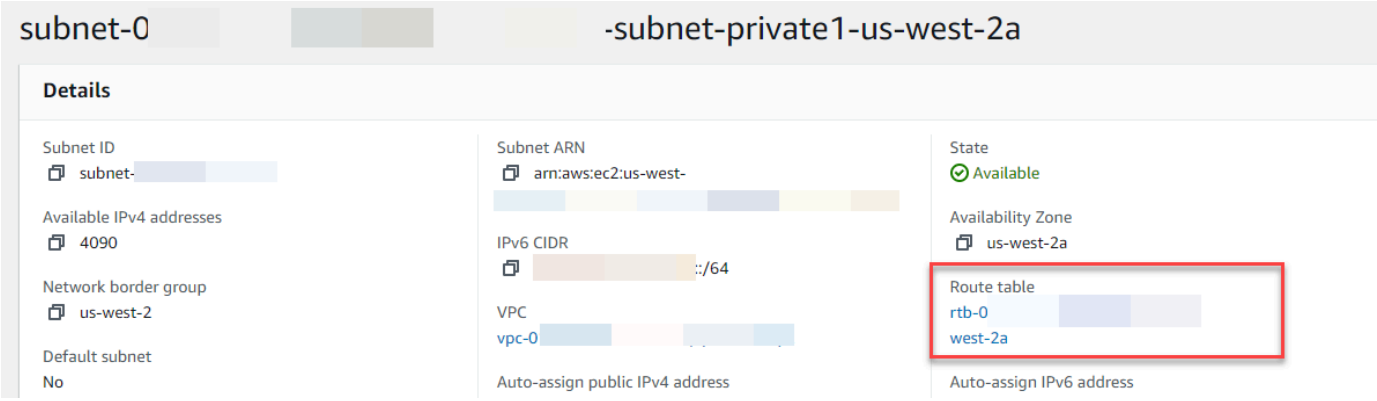
1. ナビゲーションペインで、[Egress Only インターネットゲートウェイ] を選択します。
2. [Egress Only インターネットゲートウェイの作成] を選択します。
3. (オプション) 名前を入力します。
4. Egress-Only インターネットゲートウェイを作成する VPC を選択します。
5. [Egress Only インターネットゲートウェイの作成] を選択します。
6. [アタッチされた VPC ID] のリンクを選択します。



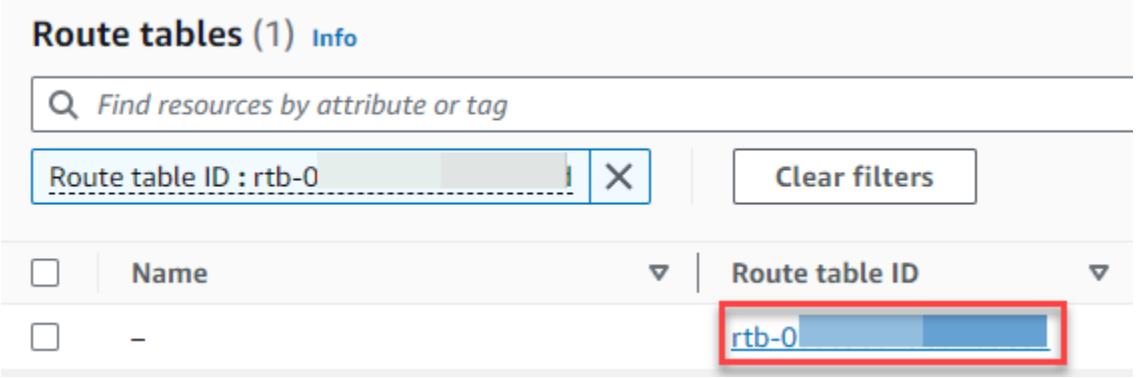
7. [VPC ID] のリンクを選択し、VPC の詳細ページを開きます。
8. [リソースマップ] セクションまでスクロールダウンし、プライベートサブネットを選択します。サブネットの詳細は新しいタブに表示されます。



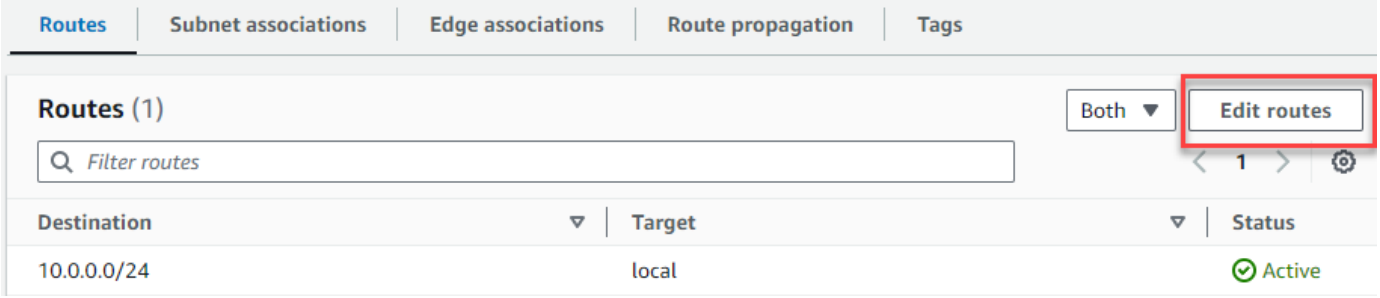
9. [ルートテーブル] のリンクを選択します。



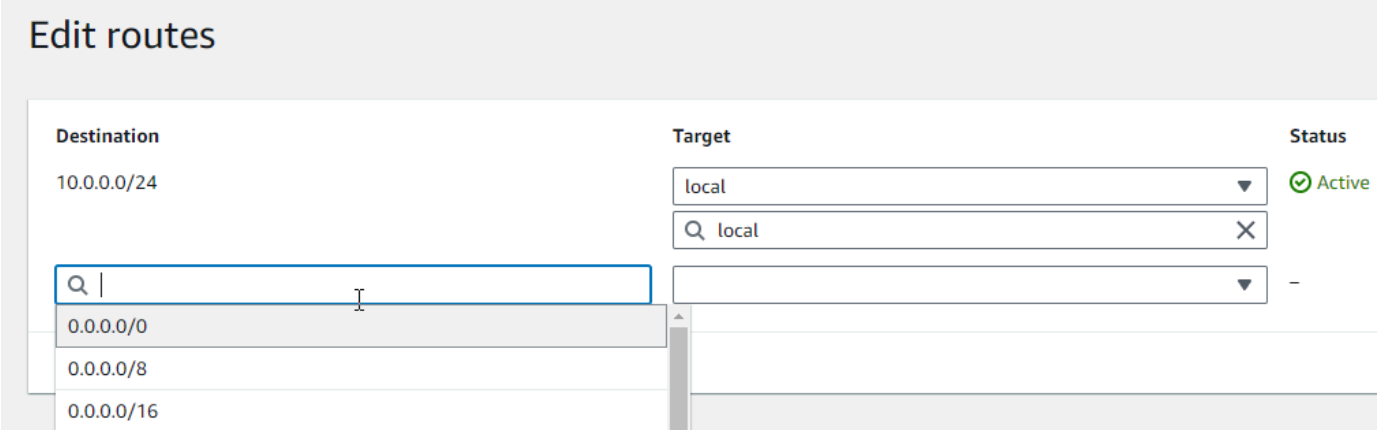
10. [ルートテーブル ID] を選択し、ルートテーブルの詳細ページを開きます。



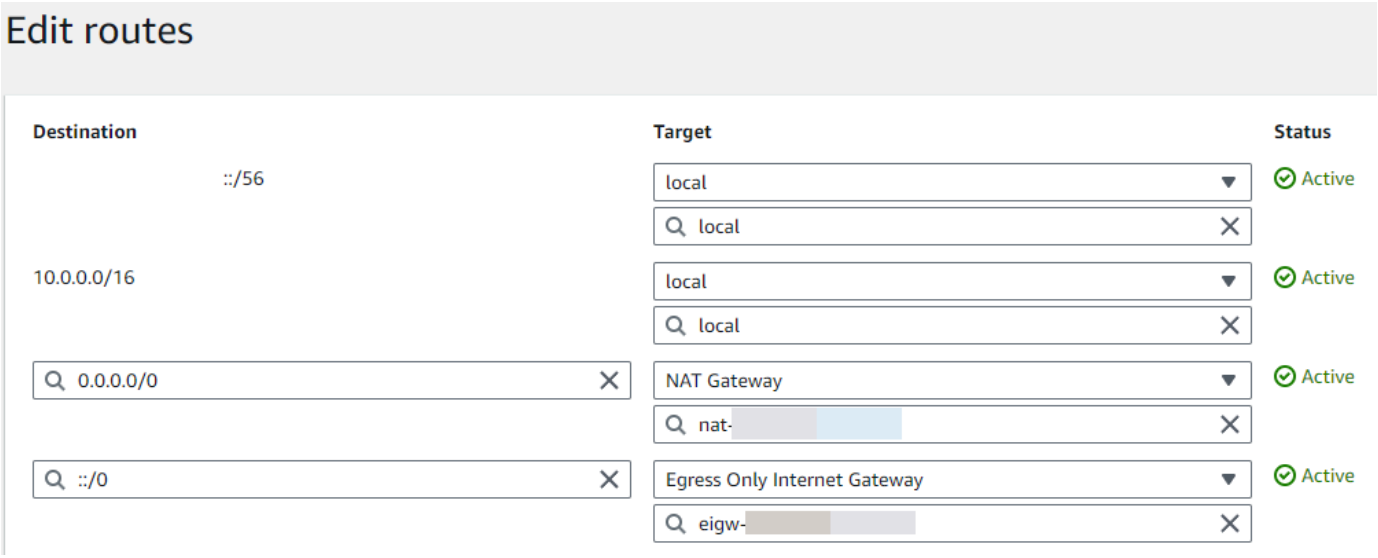
11. [ルート] で、[ルートを編集する] を選択します。



12. [ルートを追加する] を選択し、[宛先] ボックスに `::/0` と入力します。



13. [ターゲット] では、[Egress Only インターネットゲートウェイ] を選択し、先ほど作成したゲートウェイを選択します。



14. [Save changes] (変更の保存) をクリックします。

Lambda 関数を設定

関数の作成時に VPC を設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. [Create function (関数の作成)] を選択します。
3. [基本的な情報] の [関数名] に、関数の名前を入力します。
4. [Advanced settings (詳細設定)] を展開します。
5. [VPC を有効にする] を選択したら、VPC を選択します。
6. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。
7. [サブネット] では、すべてのプライベートサブネットを選択します。プライベートサブネットは、NAT ゲートウェイ経由でインターネットにアクセスできます。関数をパブリックサブネットに接続しても、インターネットにアクセスできません。

> **Note**
>
> [デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

8. [セキュリティグループ] では、アウトバウンドトラフィックを許可するセキュリティグループを選択します。
9. [Create function (関数の作成)] を選択します。

Lambda は、「AWSLambdaVPCAccessExecutionRole」 AWS 管理ポリシーをで実行ロールを自動的に作成します。このポリシーのアクセス許可は、VPC 設定の Elastic Network Interface を作成するためにのみに必要であり、関数を呼び出すためではありません。最小特権のアクセス許可を適用するには、関数および VPC 設定を作成した後に、実行ロールから [AWSLambdaVpcAccessExecutionRole] ポリシーを削除できます。詳細については、「必要な IAM 許可」を参照してください。

既存の関数に対して VPC を設定するには

既存の関数に VPC 設定を追加するには、関数の実行ロールに Elastic Network Interface を作成して管理する許可 が必要です。「AWSLambdaVPCAccessExecutionRole」 AWS 管理ポリシーには、必要な許可が含まれています。最小特権のアクセス許可を適用するには、VPC 設定を作成した後に、実行ロールから [AWSLambdaVpcAccessExecutionRole] ポリシーを削除できます。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] タブを選択したら、[VPC] を選択します。
4. [VPC] で、[編集] を選択します。
5. VPC を選択します。
6. (オプション) アウトバウンド IPv6 トラフィックを許可するには、[デュアルスタックサブネットの IPv6 トラフィックを許可] をクリックします。
7. [サブネット] では、すべてのプライベートサブネットを選択します。プライベートサブネットは、NAT ゲートウェイ経由でインターネットにアクセスできます。関数をパブリックサブネットに接続しても、インターネットにアクセスできません。

   > **Note**
   >
   > [デュアルスタックサブネットの IPv6 トラフィックを許可する] を選択した場合は、選択したすべてのサブネットに IPv4 CIDR ブロックと IPv6 CIDR ブロックが必要です。

8. [セキュリティグループ] では、アウトバウンドトラフィックを許可するセキュリティグループを選択します。

9. [Save] を選択します。

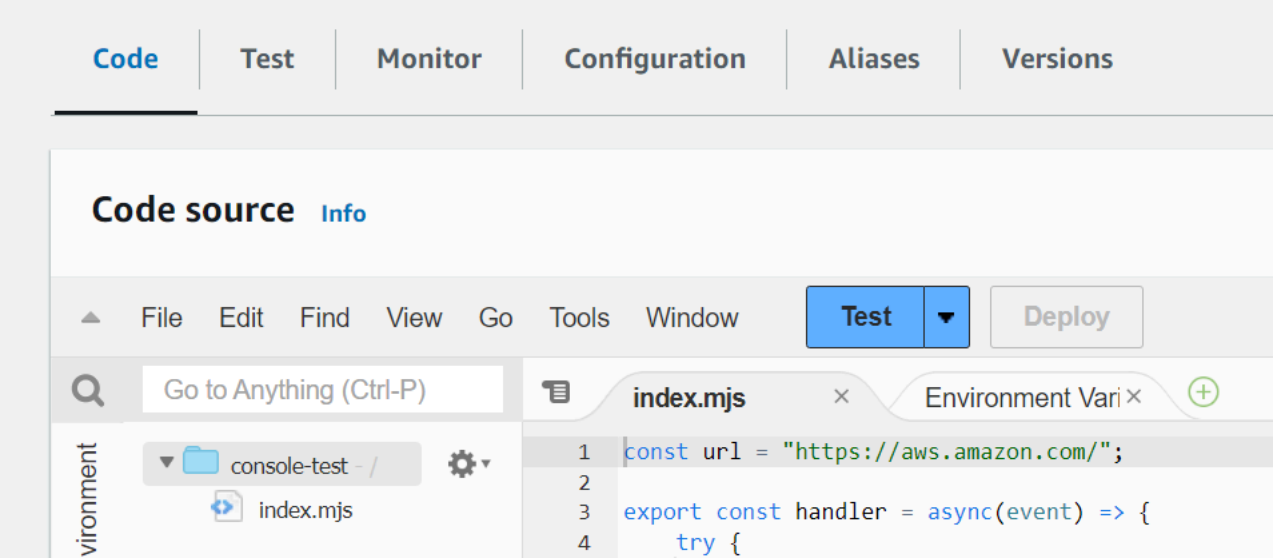
関数をテストする

次のサンプルコードを使用し、VPC に接続された関数がパブリックインターネットにアクセスできることを確認します。正常に処理された場合、コードは `200` ステータスコードを返します。失敗した場合、関数がタイムアウトします。

Node.js

この例では、`nodejs18.x` 以降のランタイムで利用できる `fetch` を使用します。

1. Lambda コンソールの [コードソース] ペインで、次のコードを [index.mjs] ファイルに貼り付けます。この関数はパブリックエンドポイントに HTTP GET リクエストを行い、HTTP レスポンスコードを返し、関数がパブリックインターネットにアクセスできるかどうかをテストします。

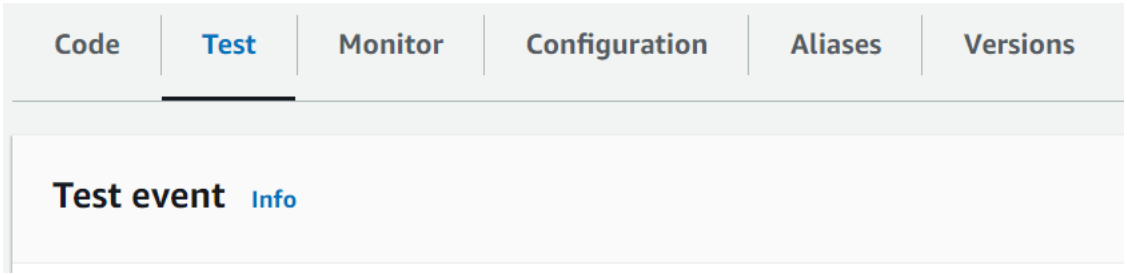


Example — async/await を持つ HTTP リクエスト

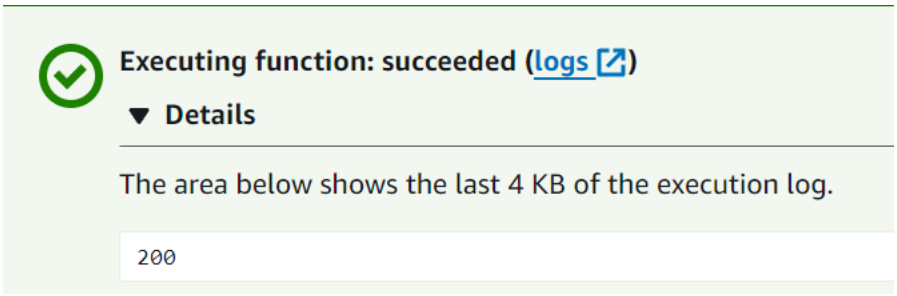
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available with Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
```

```
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

2. [デプロイ] を選択します。
3. [テスト] タブを選択します。



4. [テスト] を選択します。
5. 関数は 200 ステータスコードを返します。つまり、この関数はアウトバウンドのインターネットアクセスができることを意味します。



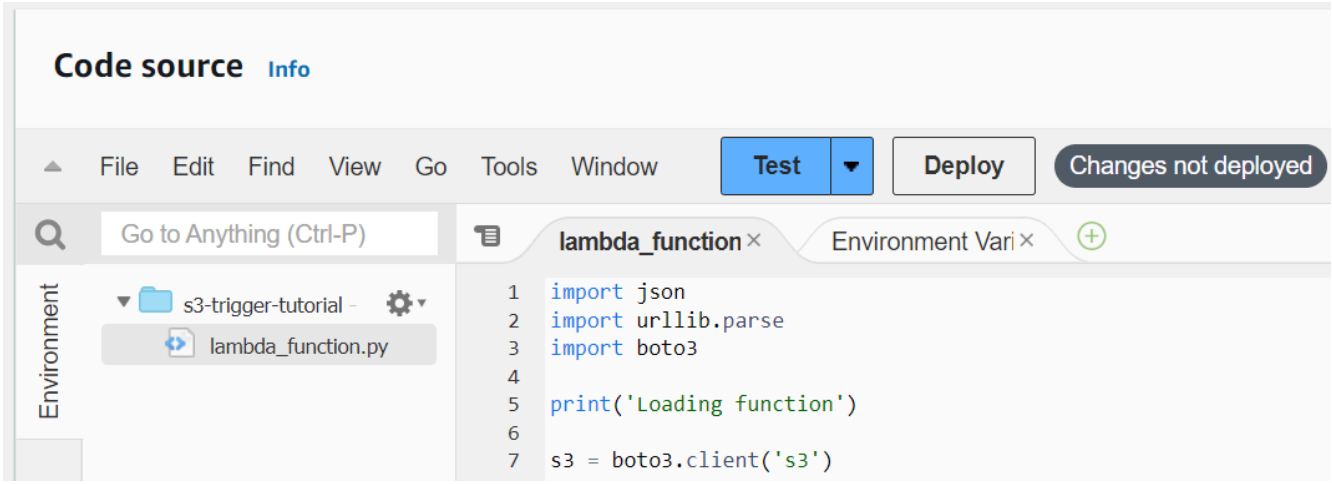
関数がパブリックインターネットにアクセスできない場合、次のようなエラーメッセージが表示されます。

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

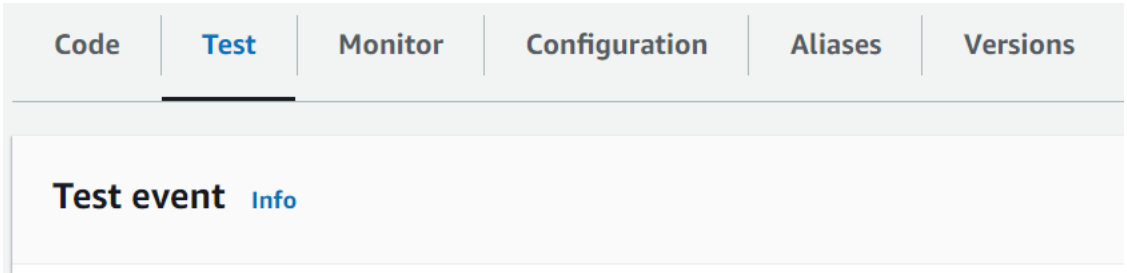
1. Lambda コンソールの [コードソースペイン] ペインで、次のコードを [lambda_function.py] ファイルに貼り付けます。この関数はパブリックエンドポイントに HTTP GET リクエストを

行い、HTTP レスポンスコードを返し、関数がパブリックインターネットにアクセスできるかどうかをテストします。

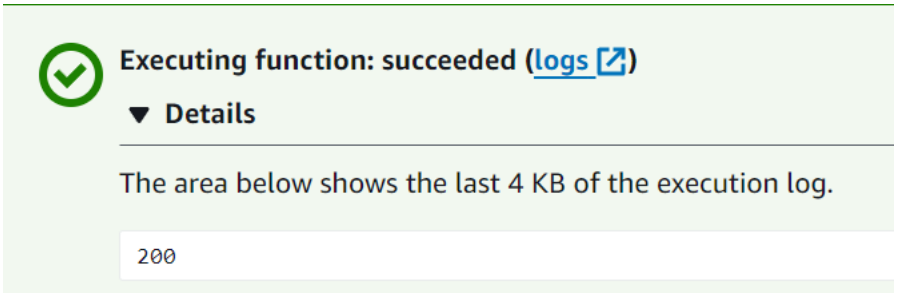



```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. [デプロイ] を選択します。
3. [テスト] タブを選択します。



4. [テスト] を選択します。
5. 関数は 200 ステータスコードを返します。つまり、この関数はアウトバウンドのインターネットアクセスができることを意味します。



関数がパブリックインターネットにアクセスできない場合、次のようなエラーメッセージが表示されます。

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Lambda 用のインバウンドインターフェイス VPC エンドポイントの接続

Amazon Virtual Private Cloud (Amazon VPC) を使用して AWS リソースをホストする場合、VPC と Lambda の間の接続を確立できます。この接続を使用して、パブリックインターネットを交差せずに Lambda 関数を呼び出すことができます。

VPC と Lambda とのプライベート接続を確立するには、インターフェイス VPC エンドポイントを作成します。インターフェイスエンドポイントは、インターネットゲートウェイ、NAT デバイス、VPN 接続、AWS Direct Connect 接続のいずれも必要とせずに Lambda API にプライベートにアクセスできる AWS PrivateLink を利用しています。VPC のインスタンスは、パブリック IP アドレスがなくても Lambda API と通信できます。VPC と Lambda との間のトラフィックは、AWS ネットワークを離れません。

各インターフェイスエンドポイントは、サブネット内の 1 つ以上の Elastic Network Interface によって表されます。ネットワークインターフェイスは、Lambda へのトラフィックのエントリポイントとなるプライベート IP アドレスを提供します。

セクション

- Lambda インターフェイスエンドポイントに関する考慮事項
- Lambda のインターフェイスエンドポイントの作成
- Lambda のインターフェイスエンドポイントポリシーを作成する

Lambda インターフェイスエンドポイントに関する考慮事項

Lambda のインターフェイスエンドポイントを設定する前に、Amazon VPC ユーザーガイドの「インターフェイスエンドポイントのプロパティと制限」を確認してください。

VPC から任意の Lambda API オペレーションを呼び出すことができます。例えば、VPC 内で `Invoke` API を呼び出すことで、Lambda 関数を呼び出すことができます。Lambda API の完全なリストについては、Lambda API リファレンスの「アクション」を参照してください。

`use1-az3` は、Lambda VPC 関数のキャパシティが制限されたリージョンです。このアベイラビリティーゾーンのサブネットを Lambda 関数で使用しないでください。これは、停止した場合にゾーンレベルの冗長性が低下する可能性があるためです。

永続的な接続のための Keep-alive

Lambda は長期間アイドル状態の接続を削除するため、永続的な接続を維持するには、Keep-alive ディレクティブを使用します。関数を呼び出すときにアイドル状態の接続を再利用しようとすると、接続エラーが発生します。永続的な接続を維持するには、ランタイムに関連付けられている keep-alive ディレクティブを使用します。例については、AWS SDK for JavaScript 開発者ガイドの「Node.js で Keep-alive を使用して接続を再利用する」を参照してください。

請求に関する考慮事項

インターフェイスエンドポイントを介して Lambda 関数にアクセスするための追加料金はありません。Lambda の料金の詳細については、「AWS Lambda の料金」を参照してください。

Lambda のインターフェイスエンドポイントには、AWS PrivateLink の標準料金が適用されます。AWS アカウントは、各アベイラビリティーゾーンでインターフェイスエンドポイントがプロビジョニングされる時間ごと、およびインターフェイスエンドポイントを介して処理されたデータに対して請求されます。インターフェイスエンドポイント料金の詳細については、「AWS PrivateLink の料金」を参照してください。

VPC ピアリングに関する考慮事項

他の VPC は、インターフェイスエンドポイントを使用して VPC ピアリング接続 によって VPC に接続できます。VPC ピアリングは、2 つの VPC 間のネットワーク接続です。自分が所有者である 2 つの VPC 間や、他の AWS アカウント内の VPC との間で、VPC ピアリング接続を確立できます。VPC は 2 つの異なる AWS リージョンの間でも使用できます。

ピア接続された VPC 間のトラフィックは AWS ネットワーク上に留まり、パブリックインターネットを経由しません。VPC がピア接続されると、Amazon Elastic Compute Cloud (Amazon EC2) インスタンス、Amazon Relational Database Service (Amazon RDS) インスタンス、または双方の VPC の VPC 対応 Lambda 関数などのリソースは、いずれかの VPC で作成されたインターフェイスエンドポイントを介して Lambda API にアクセスできます。

Lambda のインターフェイスエンドポイントの作成

Amazon VPC コンソールまたは AWS Command Line Interface (AWS CLI) を使用して、Lambda のインターフェイスエンドポイントを作成できます。詳細については、Amazon VPC ユーザーガイドのインターフェイスエンドポイントの作成を参照してください。

Lambda のインターフェイスエンドポイントを作成するには (コンソール)

1. Amazon VPC コンソールの [Endpoints (エンドポイント)] ページを開きます。

2. [Create endpoint] (エンドポイントの作成) を選択します。
3. [サービスカテゴリ] で、[AWS サービス] が選択されていることを確認します。
4. [サービス名] で [com.amazonaws.***region***.lambda を選択します。[タイプ] が [インターフェイス] であることを確認します。
5. VPC とサブネットを選択します。
6. インターフェイスエンドポイントのプライベート DNS を有効にするには、[Enable DNS Name] (DNS 名を有効にする) でチェックボックスをオンにします。AWS のサービスの VPC エンドポイントに対してプライベート DNS ホスト名を有効にすることをお勧めします。これにより、AWS SDK を介して行われたリクエストなど、パブリックサービスエンドポイントを使用するリクエストが VPC エンドポイントに解決されるようになります。
7. [セキュリティグループ] で、1 つ以上のセキュリティグループを選択します。
8. [エンドポイントの作成] を選択します。

プライベート DNS オプションを使用するには、VPC の `enableDnsHostnames` および `enableDnsSupportattributes` を設定する必要があります。詳細については、Amazon VPC ユーザーガイドの「VPC の DNS サポートを表示および更新する」を参照してください。エンドポイントのプライベート DNS を有効にすると、リージョンのデフォルト DNS 名 (`lambda.us-east-1.amazonaws.com` など) を使用して、Lambda への API リクエストを実行できます。サービスエンドポイントの詳細については、「AWS 全般のリファレンス」の「サービスエンドポイントとクォータ」を参照してください。

詳細については、「Amazon VPC ユーザーガイド」の「インターフェイスエンドポイントを介したサービスへのアクセス」を参照してください。

AWS CloudFormation を使用してエンドポイントを作成および設定する方法については、AWS CloudFormation ユーザーガイドの「AWS::EC2::VPCEndpoint」リソースを参照してください。

Lambda のインターフェイスエンドポイントを作成するには (AWS CLI)

`create-vpc-endpoint` コマンドを使用し、VPC ID、VPC エンドポイントタイプ (インターフェイス)、サービス名、エンドポイントを使用するサブネット、およびエンドポイントネットワークインターフェイスに関連付けるセキュリティグループを指定します。次に例を示します。

```
aws ec2 create-vpc-endpoint --vpc-id vpc-ec43eb89 --vpc-endpoint-type Interface --
service-name \
   com.amazonaws.us-east-1.lambda --subnet-id subnet-abababab --security-group-id
 sg-1a2b3c4d
```

Lambda のインターフェイスエンドポイントポリシーを作成する

インターフェイスエンドポイントを使用できるユーザーと、ユーザーがアクセスできる Lambda 関数を制御するために、エンドポイントにエンドポイントポリシーをアタッチできます。このポリシーでは、以下の情報を指定します。

- アクションを実行できるプリンシパル。
- プリンシパルが実行できるアクション。
- プリンシパルがアクションを実行できるリソース。

詳細については、Amazon VPC ユーザーガイドの「VPC エンドポイントによるサービスのアクセスコントロール」を参照してください。

例: Lambda アクションのインターフェイスエンドポイントポリシー

Lambda のエンドポイントポリシーの例を次に示します。エンドポイントにアタッチされると、このポリシーにより、ユーザー `MyUser` は 関数 `my-function` を呼び出すことができます。

> **Note**
>
> リソースには、修飾関数 ARN と非修飾関数 ARN の両方を含める必要があります。

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
```

```
            ]
        }
    ]
}
```

Lambda 関数のファイルシステムアクセスの設定

Amazon Elastic File System (Amazon EFS) ファイルシステムをローカルディレクトリにマウントするように関数を設定できます。Amazon EFS を使用すると、関数コードは共有リソースに安全かつ高い同時実行数でアクセスして変更できます。

セクション

実行ロールとユーザーアクセス許可

ファイルシステムにユーザー設定の AWS Identity and Access Management (IAM) ポリシーがない場合、EFS は、ファイルシステムマウントターゲットを使用してファイルシステムに接続できるすべてのクライアントへのフルアクセスを許可するデフォルトのポリシーを使用します。ファイルシステムにユーザー設定の IAM ポリシーがある場合、関数の実行ロールには正しい `elasticfilesystem` 許可が必要です。

実行ロールのアクセス許可

- elasticfilesystem:ClientMount
- elasticfilesystem:ClientWrite (読み取り専用接続には不要)

これらのアクセス許可は、AmazonElasticFileSystemClientReadWriteAccess 管理ポリシーに含まれています。さらに、実行ロールは、ファイルシステムの VPC に接続するために必要な許可を備えている必要があります。

ファイルシステムを設定する場合、Lambda はアクセス許可を使用してマウントターゲットを検証します。ファイルシステムに接続するように関数を設定するには、ユーザーに以下のアクセス許可が必要です。

ユーザーアクセス許可

- elasticfilesystem:DescribeMountTargets

ファイルシステムとアクセスポイントの設定

関数が接続するすべてのアベイラビリティーゾーンにマウントターゲットを持つファイルシステムを Amazon EFS に作成します。パフォーマンスと耐障害性のために、少なくとも 2 つのアベイラビリティーゾーンを使用します。たとえば、シンプルな設定では、別々のアベイラビリティーゾーンに 2 つのプライベートサブネットを持つ VPC を作成できます。この関数は両方のサブネットに接続し、それぞれのサブネットでマウントターゲットを使用できます。NFS トラフィック (ポート 2049) が、関数およびマウントターゲットで使用されるセキュリティグループによって許可されていることを確認します。

> **Note**
>
> ファイルシステムを作成するときは、後で変更できないパフォーマンスモードを選択します。汎用 モードではレイテンシーが低く、最大 I/O モードではより高い最大スループットと IOPS がサポートされます。選択のヘルプについては、Amazon Elastic File System ユーザーガイドの「Amazon EFS のパフォーマンス」を参照してください。

アクセスポイントは、関数の各インスタンスを、接続先のアベイラビリティーゾーンの適切なマウントターゲットに接続します。最大のパフォーマンスを得るには、ルート以外のパスを持つアクセスポイントを作成し、各ディレクトリに作成するファイル数を制限します。次の例では、ファイルシステムに `my-function` という名前のディレクトリを作成し、標準ディレクトリ許可 (755) で所有者 ID を 1001 に設定します。

Example アクセスポイントの設定

- 名前 - `files`
- ユーザー ID - `1001`
- グループ ID - `1001`
- パス - `/my-function`
- アクセス許可 - `755`
- 所有者ユーザー ID - `1001`
- グループユーザー ID - `1001`

関数がアクセスポイントを使用する場合、ユーザー ID 1001 とディレクトリへのフルアクセス許可が与えられます。

詳細については、Amazon Elastic File System ユーザーガイドの次のトピックを参照してください。

- Amazon EFS のリソースの作成
- ユーザー、グループ、アクセス許可の操作

ファイルシステムへの接続 (コンソール)

関数は、VPC 内のローカルネットワークを介してファイルシステムに接続します。関数が接続するサブネットは、ファイルシステムのマウントポイントを含む同じサブネットでも、NFS トラフィック (ポート 2049) をファイルシステムにルーティングできる同じアベイラビリティーゾーン内のサブネットでもかまいません。

> **Note**
>
> 関数が VPC にまだ接続されていない場合は、「」を参照してくださいLambda 関数に Amazon VPC 内のリソースへのアクセスを許可する

ファイルシステムアクセスを設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定]、[ファイルシステム] の順にクリックします。
4. [ファイルシステム] で、[ファイルシステムの追加] を選択します。
5. 以下のプロパティを設定します。
   - EFS ファイルシステム - 同じ VPC 内のファイルシステムのアクセスポイント。
   - ローカルマウントパス - ファイルシステムが Lambda 関数にマウントされている場所 (`/mnt/` で始まります)。

> **料金**
>
> Amazon EFS では、ストレージとスループットに対して料金が発生します。レートはストレージクラスによって異なります。詳細については、「Amazon EFS の料金」を参照してください。

Lambda では、VPC 間のデータ転送に対して料金が発生します。これは、関数の VPC がファイルシステムを持つ別の VPC にピア接続されている場合にのみ該当します。レートは、同じリージョン内の VPC 間の Amazon EC2 データ転送と同じです。詳細については、「Lambda の料金」を参照してください。

Lambda と Amazon EFS の統合の詳細については、「Lambda で Amazon EFS を使用する」を参照してください。

Lambda 関数に別の AWS アカウントの Amazon EFS ファイルシステムを使用する

Amazon EFS ファイルシステムを別の AWS アカウント にマウントする関数を設定できます。ファイルシステムをマウントする前に、次のことを確認する必要があります。

- VPC ピアリングが設定され、各 VPC のルートテーブルに適切なルートが追加されている。
- マウントする Amazon EFS ファイルシステムのセキュリティグループが、Lambda 関数に関連付けられたセキュリティグループからのインバウンドアクセスを許可するように設定されている。
- 各 VPC にアベイラビリティーゾーン (AZ) ID の一致するサブネットが作成されている。
- 両方の VPC で DNS ホスト名が有効である。

Lambda 関数が別の AWS アカウント の Amazon EFS ファイルシステムにアクセスできるようにするため、そのファイルシステムには関数にアクセス許可を付与するファイルシステムポリシーも必要です。ファイルシステムポリシーの作成方法については、「Amazon Elastic File System ユーザーガイド」の「Creating file system policies」を参照してください。

以下は、指定したアカウントの Lambda 関数に、ファイルシステム上ですべての API アクションを実行するアクセス許可を付与するポリシーの例を示しています。

```
{
    "Version": "2012-10-17",
    "Id": "efs-lambda-policy",
    "Statement": [
        {
            "Sid": "efs-lambda-statement",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::{LAMBDA-ACCOUNT-ID}:root"
```

```
            },
            "Action": "*",
            "Resource": "arn:aws:elasticfilesystem:{REGION}:{ACCOUNT-ID}:file-
system/{FILE SYSTEM ID}"
        }
    ]
}
```

> **Note**
>
> 示されているポリシーの例では、ワイルドカード文字 (「*」) を使用して、指定した AWS アカウント の Lambda 関数にファイルシステム上で任意の API オペレーションを実行するためのアクセス許可を付与しています。これには、ファイルシステムの削除も含まれます。他の AWS アカウント がファイルシステム上で実行できる操作を制限するには、許可するアクションを明示的に指定してください。利用可能な API オペレーションのリストについては、「Amazon Elastic File System のアクション、リソース、条件キー」を参照してください。

クロスアカウントのファイルシステムのマウントを設定するには、AWS Command Line Interface (AWS CLI) の `update-function-configuration` オペレーションを使用します。

別の AWS アカウント でファイルシステムをマウントするには、次のコマンドを実行します。独自の関数名を使用して、マウントするファイルシステム用に Amazon リソースネーム (ARN) を Amazon EFS アクセスポイントの ARN に置き換えます。`LocalMountPath` は関数がファイルシステムにアクセスできるパスで、`/mnt/` で始まります。Lambda のマウントパスがファイルシステムのアクセスポイントのパスと一致することを確認してください。例えば、アクセスポイントが `/efs` である場合、Lambda のマウントパスは `/mnt/efs` になる必要があります。

```
aws lambda update-function-configuration --function-name MyFunction \
--file-system-configs Arn=arn:aws:elasticfilesystem:us-east-1:222222222222:access-
point/fsap-01234567,LocalMountPath=/mnt/test
```

Lambda 関数のエイリアスの作成

Lambda 関数のエイリアスを作成できます。Lambda のエイリアスとは、更新可能な関数のバージョンへのポインタです。関数のユーザーは、エイリアス Amazon リソースネーム (ARN) を使用して関数のバージョンにアクセスできます。新しいバージョンをデプロイする場合は、新しいバージョンを使用するようにエイリアスを更新するか、2 つのバージョン間でトラフィックを分割することができます。

セクション

関数エイリアスの作成 (コンソール)

Lambda コンソールを使用して、関数エイリアスを作成できます。

エイリアスを作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [エイリアス]、[エイリアスの作成] の順にクリックします。
4. [Create alias (エイリアスの作成)] ページで、以下の操作を行います。
    a. [Name (名前)] で、エイリアスの名前を入力します。
    b. (オプション) [Description (説明)] で、エイリアスの説明を入力します。
    c. [Version (バージョン)] で、エイリアスが参照する関数のバージョンを選択します。
    d. (オプション) エイリアスのルーティングを設定するには、[Weighted alias (加重エイリアス)] を展開します。詳細については、「エイリアスのルーティング設定」を参照してください。
    e. [Save] を選択します。

Lambda API を使用したエイリアスの管理

AWS Command Line Interface (AWS CLI) を使用してエイリアスを作成するには、`create-alias` コマンドを使用します。

```
aws lambda create-alias --function-name my-function --name alias-name --function-version version-number --description " "
```

関数の新しいバージョンを参照するようにエイリアスを変更するには、`update-alias` コマンドを使用します。

```
aws lambda update-alias --function-name my-function --name alias-name --function-version version-number
```

エイリアスを削除するには、`delete-alias` コマンドを使用します。

```
aws lambda delete-alias --function-name my-function --name alias-name
```

前のステップの AWS CLI コマンドは、以下の Lambda API オペレーションに対応しています。

- CreateAlias
- UpdateAlias
- DeleteAlias

AWS SAM と AWS CloudFormation を使用したエイリアスの管理

AWS Serverless Application Model (AWS SAM) と AWS CloudFormation を使用して、関数エイリアスを作成し、管理することができます。

AWS SAM テンプレートで関数エイリアスを宣言する方法については、「AWS SAM デベロッパーガイド」の「AWS::Serverless::Function」ページを参照してください。AWS CloudFormation を使用したエイリアスの作成と設定については、「AWS CloudFormation ユーザーガイド」の「AWS::Lambda::Alias」を参照してください。

エイリアスの使用

各エイリアスには一意の ARN があります。エイリアスは関数のバージョンのみを参照でき、別のエイリアスを参照することはできません。関数の新しいバージョンを指すよう、エイリアスを更新できます。

Amazon Simple Storage Service (Amazon S3) などのイベントソースが、Lambda 関数を呼び出します。これらのイベントソースは、イベント発生時に呼び出す関数を識別するマッピングを維持します。マッピング設定で Lambda 関数のエイリアスを指定する場合、関数のバージョンが変更されたときにマッピングを更新する必要はありません。詳細については、「Lambda がストリームおよびキューベースのイベントソースからのレコードを処理する方法」を参照してください。

リソースポリシーでは、Lambda 関数を使用するためのイベントソースにアクセス許可を付与できます。ポリシーでエイリアス ARN を指定した場合、関数のバージョンが変更されたときにポリシーを更新する必要はありません。

リソースポリシー

リソースベースのポリシーを使用して、サービス、リソース、またはアカウントに、関数に対するアクセス許可を付与できます。このアクセス許可の範囲は、ポリシーの適用先がエイリアス、バージョン、または関数全体のいずれになるかによって決まります。たとえば、エイリアス名 (`helloworld:PROD` など) を使用する場合は、アクセス許可で、エイリアス ARN (`helloworld`) を使用した `helloworld:PROD` 関数の呼び出しを許可できます。

エイリアスまたは特定のバージョンを使用しないで関数を呼び出そうとすると、アクセス許可エラーが発生します。このアクセス許可エラーは、エイリアスに関連付けられた関数のバージョンを直接呼び出そうとしても発生します。

例えば、以下の AWS CLI コマンドは、Amazon S3 が `DOC-EXAMPLE-BUCKET` に代わって動作しているときに、`helloworld` 関数の PROD エイリアスを呼び出すアクセス許可を Amazon S3 に付与します。

```
aws lambda add-permission --function-name helloworld \
--qualifier PROD --statement-id 1 --principal s3.amazonaws.com --action
 lambda:InvokeFunction \
--source-arn arn:aws:s3:::DOC-EXAMPLE-BUCKET --source-account 123456789012
```

ポリシーでのリソース名の使用の詳細については、「ポリシーのリソースセクションと条件セクションの微調整」を参照してください。

エイリアスのルーティング設定

エイリアスでルーティング設定を使用して、トラフィックの一部を 2 番目の関数バージョンに送信します。たとえば、エイリアスを設定して、ほとんどのトラフィックを既存のバージョンに送信し、

トラフィックの一部を新しいバージョンに送信するように設定することで、新しいバージョンを展開するリスクを軽減できます。

Lambda は単純な確率モデルを使用して、2 つの関数バージョン間でトラフィックを分配することに注意してください。低いトラフィックレベルでは、各バージョンで設定されたトラフィックの割合と実際の割合の間に大きな差異が生じる場合があります。関数がプロビジョニングされた同時実行を使用する場合、エイリアスルーティングがアクティブである間に、プロビジョニングされた同時実行インスタンスの数を高く設定することで、過剰呼び出しを防ぐことができます。

エイリアスは最大 2 つの Lambda 関数バージョンを指すことができます。バージョンは次の基準を満たす必要があります。

- 両方のバージョンに同じ実行ロールが必要です。
- どちらのバージョンも、同じ配信不能キュー設定を持つか、配信不能キュー設定がない必要があります。
- 両方のバージョンを公開する必要があります。エイリアスが $LATEST を指すことはできません。

エイリアスのルーティングを構成するには

> **Note**
>
> 関数に少なくとも 2 つの公開バージョンがあることを確認します。追加のバージョンを作成するには、「Lambda 関数のバージョン」の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [エイリアス]、[エイリアスの作成] の順にクリックします。
4. [Create alias (エイリアスの作成)] ページで、以下の操作を行います。
   a. [Name (名前)] で、エイリアスの名前を入力します。
   b. (オプション) [Description (説明)] で、エイリアスの説明を入力します。
   c. [Version (バージョン)] で、エイリアスが参照する最初の関数のバージョンを選択します。
   d. [Weighted alias (加重エイリアス)] を展開します。
   e. [Additional version (追加のバージョン)] で、エイリアスが参照する 2 番目の関数のバージョンを選択します。

f. [Weight (%) (重み (%))] で、関数の重み値を入力します。重みとは、エイリアスが呼び出されたときにそのバージョンに割り当てられるトラフィックの割合 (%) です。最初のバージョンは残余重みを受け取ります。たとえば、[Additional version] に 10 パーセントを指定した場合、最初のバージョンには自動的に 90 パーセントが割り当てられます。

g. [Save] を選択します。

CLI を使用したエイリアスルーティングの設定

`create-alias` および `update-alias` AWS CLI コマンドを使用して、2 つの関数のバージョン間のトラフィックの重みを設定します。エイリアスを作成または更新するときは、`routing-config` パラメータでトラフィックの重みを指定します。

以下の例では、関数のバージョン 1 を参照する routing-alias という名前の Lambda 関数のエイリアスを作成します。関数のバージョン 2 は、トラフィックの 3% を受信します。残りの 97% のトラフィックはバージョン 1 にルーティングされます。

```
aws lambda create-alias --name routing-alias --function-name my-function --function-
version 1  \
--routing-config AdditionalVersionWeights={"2"=0.03}
```

バージョン 2 への着信トラフィックのパーセンテージを増やすには、`update-alias` コマンドを使用します。次の例では、トラフィックを 5% に増やします。

```
aws lambda update-alias --name routing-alias --function-name my-function  \
--routing-config AdditionalVersionWeights={"2"=0.05}
```

すべてのトラフィックをバージョン 2 にルーティングするには、`update-alias` コマンドを使用して、エイリアスがバージョン 2 を参照するように `function-version` プロパティを変更します。このコマンドでは、ルーティング設定もリセットされます。

```
aws lambda update-alias --name routing-alias --function-name my-function  \
--function-version 2 --routing-config AdditionalVersionWeights={}
```

前のステップの AWS CLI コマンドは、以下の Lambda API オペレーションに対応しています。

- CreateAlias
- UpdateAlias

起動されたバージョンの特定

2 つの関数バージョン間でトラフィックの重みを設定している場合、どちらの Lambda 関数バージョンが呼び出されたかを特定するための 2 つの方法があります。

- CloudWatch Logs – すべての関数呼び出しで、Amazon CloudWatch Logs に対して呼び出されたバージョン ID を含むSTARTログエントリを Lambda が自動的に発信します。次に例を示します。

  ```
  19:44:37 START RequestId: request id Version: $version
  ```

  エイリアスの呼び出しでは、Lambda はExecuted Versionディメンションを使用して、呼び出されたバージョンでメトリクスデータをフィルタリングします。詳細については、「Lambda 関数のメトリクスの使用」を参照してください。
- 応答のペイロード (同期呼び出し) – 同期関数呼び出しの応答には、呼び出された関数バージョンを示す x-amz-executed-version ヘッダーが含まれます。

Lambda 関数のバージョン

バージョンを使用して、関数のデプロイを管理できます。たとえば、安定した実稼働バージョンのユーザーに影響を与えることなく、ベータテスト用の新しいバージョンの関数を公開できます。Lambda は、関数を公開するたびに新しいバージョンの関数を作成します。新しいバージョンは、関数の未公開バージョンのコピーです。未公開バージョンの名前は $LATEST です。

> **Note**
>
> 新しいバージョンの関数の作成するには、最初に未公開バージョン ($LATEST) を変更する必要があります。これらの変更には、コードの更新や設定の変更が含まれます。$LATEST が以前に公開されたバージョンと同じ場合、変更を $LATEST にデプロイするまで新しいバージョンを作成することはできません。

関数のバージョンを公開すると、そのコード、ランタイム、アーキテクチャ、メモリ、レイヤー、その他のほとんどの設定はイミュータブルになります。つまり、$LATEST から新しいバージョンを公開しないと、これらの設定を変更することはできません。公開済みの関数のバージョンには、次の項目を設定できます。

- トリガ
- 送信先
- プロビジョニングされた同時実行数
- 非同期呼び出し
- データベース接続とプロキシ

> **Note**
>
> [自動] モードでランタイム管理コントロールを使用している場合は、関数のバージョンで使用されるランタイムバージョンが自動的に更新されます。[Function update] (関数の更新) または [Manual] (手動) モードを使用している場合、ランタイムバージョンは更新されません。詳細については、「the section called “ランタイム更新”」を参照してください。

セクション

- 関数バージョンの作成

- バージョンの使用
- アクセス許可の付与

関数バージョンの作成

関数コードと設定は、未公開バージョンの関数でのみ変更できます。バージョンを公開すると、Lambda は、そのバージョンのユーザー向けに一貫したエクスペリエンスを保つためにコードとほとんどの設定をロックします。

Lambda コンソールを使用して、関数のバージョンを作成できます。

新しい関数バージョンを作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択し、[バージョン] をクリックします。
3. バージョンの設定ページで、[新しいバージョンを発行] をクリックします。
4. (オプション) バージョンの説明を入力します。
5. [Publish] (発行) を選択します。

代替手段として、PublishVersion API 操作を使用して関数のバージョンを発行できます。

以下の AWS CLI コマンドは、関数の新しいバージョンを発行します。レスポンスは、バージョン番号およびバージョンがサフィックスとしてついた関数 ARN を含む、新しいバージョンの設定情報を返します。

```
aws lambda publish-version --function-name my-function
```

以下の出力が表示されます。

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs20.x",
  ...
}
```

> **Note**
>
> Lambda は、バージョニングのために単調増加するシーケンス番号を割り当てます。関数を削除して再作成した後でも、Lambda がバージョン番号を再利用することは一切ありません。

バージョンの使用

修飾 ARN または非修飾 ARN を使用して、Lambda 関数を参照できます。

- 修飾 ARN - バージョンのサフィックスが付いた関数 ARN です。以下の例では、`helloworld` 関数のバージョン 42 を参照しています。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- 非修飾 ARN - バージョンのサフィックスが付いていない関数 ARN です。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

関連するすべての API オペレーションで、修飾 ARN または非修飾 ARN を使用できます。ただし、非修飾 ARN を使用してエイリアスを作成することはできません。

関数のバージョンを公開しないことにした場合、関数を呼び出すには、修飾 ARN または非修飾 ARN をイベントソースマッピングで使用します。非修飾 ARN を使用して関数を呼び出すと、Lambda によって暗黙的に $LATEST が呼び出されます。

コードが公開されたことがない場合、またはコードが最後に公開されたバージョンから変更された場合にのみ、Lambda によって新しい関数のバージョンが公開されます。変更がない場合、関数のバージョンは最後に公開されたバージョンのままです。

各 Lambda 関数のバージョンの修飾 ARN は一意です。バージョンを公開した後は、ARN または関数コードを変更することはできません。

アクセス許可の付与

リソースベースのポリシーまたはアイデンティティベースのポリシーを使用して、関数へのアクセスを許可できます。このアクセス許可の範囲は、ポリシーの適用先が関数になるか、関数の 1 つの

バージョンになるかによって決まります。ポリシーでの関数のリソース名の詳細については、「ポリシーのリソースセクションと条件セクションの微調整」を参照してください。

関数のエイリアスを使用すると、イベントソースおよび AWS Identity and Access Management (IAM) ポリシーの管理を簡素化することができます。詳細については、「Lambda 関数のエイリアスの作成」を参照してください。

ストリームレスポンスに Lambda 関数を設定する

Lambda 関数 URL を設定して、レスポンスペイロードをクライアントにストリーミングで返すようにできます。レスポンスストリーミングは、最初のバイトまでの時間 (TTFB) のパフォーマンスを向上させることで、レイテンシーの影響を受けやすいアプリケーションに役立ちます。これは、レスポンスの一部が利用可能になったときにクライアントに返送できるためです。さらに、レスポンスストリーミングを使用して、より大きなペイロードを返す関数を構築できます。レスポンスストリームのペイロードには、バッファされるレスポンスの 6 MB と比較して 20 MB のソフトリミットがあります。応答をストリーミングするということは、関数が応答全体をメモリに収める必要がないことも意味します。応答サイズが非常に大きい場合は、関数に設定する必要のあるメモリ量を減らすことができます。

Lambda が応答をストリーミングする速度は、応答サイズによって異なります。関数の応答では、最初の 6 MB のストリーミングレートには上限がありません。応答が 6 MB を超える場合、残りの応答には帯域幅の上限が適用されます。ストリーミング帯域幅の詳細については、レスポンスストリーミングの帯域幅制限 を参照してください。

ストリーミングレスポンスにはコストが発生します。詳細については、「AWS Lambdaの料金」を参照してください。

Lambda は Node.js マネージドランタイムでのレスポンスストリーミングをサポートしています。その他の言語の場合は、カスタムランタイム API 統合を備えたカスタムランタイムを使用してレスポンスをストリーミングするか、Lambda Web Adapter を使用することができます。Lambda 関数 URLs 、 AWS SDK、または Lambda InvokeWithResponseStream API を使用してレスポンスをストリーミングできます。

> **Note**
>
> Lambda コンソールで関数をテストするとき、レスポンスは常にバッファリングされた状態で表示されます。

レスポンスストリーミング対応関数の作成

レスポンスストリーミング関数のハンドラー記述は、一般的なハンドラーパターンとは異なります。ストリーミング関数の作成時は、必ず次の事項を確認してください。

- ネイティブ Node.js ランタイムの `awslambda.streamifyResponse()` デコレーターで関数をラップします。

- 確実にすべてのデータ処理を完了させるため、ストリームを正常に終了させます。

ストリームレスポンスにハンドラー関数を設定する

Lambda が関数のレスポンスをストリーミングする必要があることをランタイムに示すには、関数を `streamifyResponse()` デコレーターでラップする必要があります。これにより、レスポンスをストリーミングするために適切なロジックパスを使用するようランタイムに指示し、関数がレスポンスをストリーミングできるようにします。

`streamifyResponse()` デコレーターは、次のパラメータを受けつける関数を受けつけます。

- `event` — HTTP メソッド、クエリパラメータ、リクエスト本文など、関数 URL の呼び出しイベントに関する情報を提供します。
- `responseStream` — 書き込み可能なストリームを提供します。
- `context` – 呼び出し、関数、実行環境に関する情報を含むメソッドおよびプロパティを提供します。

`responseStream` オブジェクトは「Node.js `writableStream`」です。他のこのようなストリームと同様に、`pipeline()` メソッドを使用する必要があります。

Example レスポンスストリーミング対応ハンドラー

```
const pipeline = require("util").promisify(require("stream").pipeline);
const { Readable } = require('stream');

exports.echo = awslambda.streamifyResponse(async (event, responseStream, _context) => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

`responseStream` にはストリームに書き込む `write()` メソッドが用意されていますが、「`pipeline()`」を可能な限り使用することをお勧めします。`pipeline()` を使用すると、書き込み可能なストリームが読み取り速度の速いストリームによって抑制されることがなくなります。

ストリームを終了する

ハンドラーが戻る前にストリームが適切に終了することを確認してください。pipeline() メソッドはこれを自動的に処理します。

他のユースケースでは、responseStream.end() メソッドを呼び出してストリームを適切に終了させます。このメソッドは、ストリームにこれ以上データを書き込む必要がないことを通知します。pipeline() または pipe() でストリームに書き込む場合、このメソッドは不要です。

Example pipeline() でストリームを終了する例

```
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  await pipeline(requestStream, responseStream);
});
```

Example pipeline() なしでストリームを終了する例

```
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Lambda 関数 URL を使用してレスポンスストリーミング対応関数を呼び出す

> **Note**
>
> レスポンスをストリーミングするには、関数 URL を使用して関数を呼び出す必要があります。

関数 URL の呼び出しモードを変更することで、レスポンスストリーミング対応関数を呼び出すことができます。呼び出しモードは、Lambda が関数を呼び出すために使用する API オペレーションを決定します。利用可能な呼び出しモードは以下のとおりです。

- BUFFERED – これはデフォルトのオプションです。Lambda は Invoke API オペレーションを使用して関数を呼び出します。ペイロードが完了すると、呼び出し結果が表示されます。最大ペイロードサイズは 6 MB です。
- RESPONSE_STREAM — ペイロード結果が利用可能になったら、関数がそれをストリーミングできるようにします。Lambda は InvokeWithResponseStream API オペレーションを使用して関数を呼び出します。レスポンスペイロードの最大サイズは 20 MB です。ただし、「クォータ引き上げをリクエスト」することができます。

Invoke API オペレーションを直接呼び出すことにより、レスポンスストリーミングなしで関数を呼び出すことができます。ただし、呼び出しモードを BUFFERED に変更しない限り、Lambda は関数 URL を経由するすべての呼び出しレスポンスペイロードをストリーミングします。

関数 URL の呼び出しモードを設定するには (コンソール)

1. Lambda コンソールの [関数] ページを開きます。
2. 呼び出しモードを設定する関数の名前を選択します。
3. [設定] タブを開き、次に [関数 URL] をクリックします。
4. [編集] を選択し、次に [追加設定] を選択します。
5. [呼び出しモード] で目的の呼び出しモードを選択します。
6. [保存] をクリックします。

関数 URL (AWS CLI) の呼び出しモードを設定するには

```
aws lambda update-function-url-config --function-name my-function --invoke-mode
 RESPONSE_STREAM
```

関数 URL (AWS CloudFormation) の呼び出しモードを設定するには

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
```

```
    InvokeMode: RESPONSE_STREAM
```

関数 URL の設定の詳細については、「Lambda 関数 URL」を参照してください。

レスポンスストリーミングの帯域幅制限

関数のレスポンスペイロードの最初の 6 MB の帯域幅には上限がありません。この最初のバースト後、Lambda はレスポンスを最大 2 Mbps のレートでストリーミングします。関数のレスポンスが 6 MB を超えない場合、この帯域幅制限は適用されません。

> **Note**
>
> 帯域幅制限は関数のレスポンスペイロードにのみ適用され、関数によるネットワークアクセスには適用されません。

上限のない帯域幅は、関数の処理速度を含む多くの要素によって異なります。通常、関数のレスポンスでは、最初の 6 MB で 2 Mbps を超えるレートが期待できます。関数が AWS 外部の宛先にレスポンスをストリーミングする場合、ストリーミングレートは外部インターネット接続の速度にも依存します。

チュートリアル: 関数 URL を使用してレスポンスストリーミング Lambda 関数を作成する

このチュートリアルでは、レスポンスストリームを返す関数 URL エンドポイントを持つ .zip ファイルアーカイブとして定義された Lambda 関数を作成します。関数 URL の設定の詳細については、「関数 URL の作成と管理」を参照してください。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する

AWS リソースにアクセスするためのアクセス権限をLambda 関数に付与する実行ロールを作成します。

実行ロールを作成するには

1. AWS Identity and Access Management (IAM) コンソールの ロールページを開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。
   - [信頼できるエンティティタイプ] – [AWS のサービス]
   - [ユースケース] – [Lambda]
   - アクセス許可 - AWSLambdaBasicExecutionRole。
   - [ロール名] – **response-streaming-role**

AWSLambdaBasicExecutionRole ポリシーには、ログを Amazon CloudWatch Logs に書き込むために関数が必要とするアクセス許可が含まれています。ロールを作成した後、Amazon リソースネーム (ARN) を書き留めてください。これは次のステップで必要になります。

レスポンスストリーミング関数を作成する (AWS CLI)

AWS Command Line Interface (AWS CLI) を使用して、関数 URL エンドポイントでレスポンスストリーミング Lambda 関数を作成します。

レスポンスをストリーミングできる関数を作成するには

1. 以下のコード例を index.mjs という名前のファイルにコピーします。

```
import util from 'util';
import stream from 'stream';
const { Readable } = stream;
const pipeline = util.promisify(stream.pipeline);

/* global awslambda */
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));
  await pipeline(requestStream, responseStream);
});
```

2. デプロイパッケージを作成します。

```
zip function.zip index.mjs
```

3. create-function コマンドを使用して Lambda 関数を作成します。--role の値を、前のステップで書き留めたロールの ARN に置き換えます。

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs16.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

関数 URL を作成するには

1. 関数に、関数 URL へのアクセスを許可するリソースベースのポリシーを追加します。--principal の値を AWS アカウント ID に置き換えます。

```
aws lambda add-permission \
```

```
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id 12345 \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM \
  --statement-id url
```

2. create-function-url-config コマンドを使用して、関数の URL エンドポイントを作成します。

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

関数 URL エンドポイントをテストする

関数を呼び出して統合をテストします。関数 URL をブラウザで開くことも、curl を使用することもできます。

```
curl --request GET "<function_url>" --user "<key:token>" --aws-sigv4 "aws:amz:us-
east-1:lambda" --no-buffer
```

関数 URL では、認証タイプ IAM_AUTH を使用します。つまり、AWS アクセスキーとシークレットキーの両方でリクエストに署名する必要があります。前のコマンドで、<key:token> を AWS アクセスキー ID に置き換えます。プロンプトが表示されたら、AWS シークレットキーを入力します。AWS シークレットキーがない場合は、代わりに一時的な AWS 認証情報を使用できます。

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。

4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[Delete] (削除) を選択します。

Lambda 関数のデプロイ

Lambda 関数にコードをデプロイする際は、zip ファイルアーカイブをアップロードするか、コンテナーイメージを作成してアップロードします。

トピック

.zip ファイルアーカイブ

.zip ファイルアーカイブには、アプリケーションコードとその依存関係が集録されています。Lambda コンソールまたはツールキットを使用して関数を作成する際、コードの.zip ファイルアーカイブが、Lambda により自動的に作成されます。

Lambda API、コマンドラインツール、または AWS SDK を使用して関数を管理する場合は、デプロイパッケージを作成する必要があります。また、関数でコンパイルされた言語を使用している場合、または関数に依存関係を追加する場合も、デプロイパッケージを作成する必要があります。デプロイパッケージは、関数のコードをデプロイする際に、Amazon Simple Storage Service (Amazon S3) もしくはローカルマシンからアップロードします。

Lambda コンソール、AWS Command Line Interface (AWS CLI) を使用してデプロイパッケージとして .zip ファイルをアップロードしたり、Amazon Simple Storage Service (Amazon S3) バケットにアップロードしたりできます。

デプロイパッケージファイルのアクセス許可

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

コンテナイメージ

Docker コマンドラインインターフェイス (CLI) などのツールを使用すると、コードと依存関係をコンテナーイメージとしてパッケージ化することができます。その後、Amazon Elastic Container Registry (Amazon ECR) でホストされているコンテナレジストリに、イメージをアップロードできます。

関数を呼び出すと、Lambda は実行環境にコンテナイメージをデプロイします。Lambda は拡張機能を初期化してから、関数の初期化コード (メインハンドラ外のコード) を実行します。関数の初期化期間は、請求される実行時間に含まれていることに注意してください。

その後、Lambda は、関数設定 (ENTRYPOINT および CMD コンテナイメージ設定) で指定されたコードエントリポイントを呼び出して関数を実行します。

AWSには、関数のコードをコンテナーイメージとしてビルドする際にベースとして使用できるイメージのセットが、オープンソースとして用意されています。また、他のコンテナレジストリから代わりのベースイメージを取得して使用することもできます。AWS には、代わりのベースイメージに追加して Lambda サービスとの互換性を確保するための、オープンソースのランタイムクライアントも用意されています。

さらに AWS では、Docker CLI などのツールを使用して関数をローカルでテストするための、ランタイムインターフェイスエミュレーターも利用できます。

> **Note**
>
> Lambda がサポートする命令セットアーキテクチャの 1 つと互換性を持つように、各コンテナイメージを作成します。Lambda は各命令セットアーキテクチャのベースイメージを提供し、両方のアーキテクチャをサポートするベースイメージも提供します。
> 関数用に構築するイメージでは、アーキテクチャの 1 つだけをターゲットにする必要があります。

関数をコンテナイメージとしてパッケージ化してデプロイした場合でも、追加料金は発生しません。コンテナーイメージとしてデプロイした関数を呼び出した際に、その呼び出しリクエストと実行時間

の長さに応じた課金が行われます。Amazon ECR にコンテナイメージを保存した場合にも料金が発生します。詳細については、Amazon ECR の料金を参照してください。

イメージのセキュリティ

Lambda が元のソース (Amazon ECR) から最初にコンテナイメージをダウンロードするとき、コンテナイメージは、認証されたコンバージェント暗号化方式を使用して最適化、暗号化、および保存されます。顧客データの復号化に必要なすべてのキーは、AWS KMS カスタマー管理のキーを使用して保護されます。Lambda によるカスタマー管理のキーの使用状況を追跡および監査するために、AWS CloudTrail ログを表示できます。

Lambda 関数の .zip ファイルアーカイブとしてのデプロイ

Lambda 関数を作成する場合、関数コードをデプロイパッケージにパッケージ化します。Lambda は、コンテナイメージと .zip ファイルアーカイブの 2 種類のデプロイパッケージをサポートします。関数を作成するワークフローは、デプロイパッケージの種類によって異なります。コンテナイメージとして定義される関数の作成については、the section called “コンテナイメージ”を参照してください。

Lambda コンソールと Lambda API を使用して、.zip ファイルアーカイブで定義された関数を作成できます。更新済みの .zip ファイルをアップロードして、関数コードを変更することもできます。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

トピック

関数の作成

.zip ファイルアーカイブで定義した関数を作成する場合、その関数のコードテンプレート、言語バージョン、実行ロールを選択します。Lambda が関数を作成したら、関数コードを追加します。

関数を作成するには

1. Lambda コンソールの [関数ページ] を開きます。

2. [Create function (関数の作成)] を選択します。
3. [Author from scratch] (ゼロから作る) または [Use a blueprint] (設計図の使用) を選択して関数を作成します。
4. [基本的な情報] で、以下を実行します。
   a. [関数名] に関数名を入力します。関数名の長さは 64 文字に制限されています。
   b. [Runtime] (ランタイム) で、関数で使用する言語バージョンを選択します。
   c. (オプション) アーキテクチャで、関数に使用する命令セットアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用のデプロイパッケージを構築するときは、この命令セットアーキテクチャと互換性があることを確認してください。
5. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。実行ロールを使用することも、既存のロールを使用することもできます。
6. (オプション) [詳細設定] を展開します。関数の [Code signing configuration] (コード署名設定) を選択します。関数がアクセスできるように (Amazon VPC) を設定することもできます。
7. [Create function] (関数の作成) をクリックします。

Lambda によって、新しい関数が作成されます。コンソールを使用して関数コードを追加し、他の関数のパラメータや機能を設定できるようになりました。コードのデプロイに関する手順については、関数が使用するランタイムのハンドラーページを参照してください。

Node.js

.zip ファイルアーカイブで Node.js Lambda 関数をデプロイする

Python

Python Lambda 関数で .zip ファイルアーカイブを使用する

Ruby

Ruby Lambda 関数で .zip ファイルアーカイブを使用する

Java

.zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする

Go

.zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする

C#

- .zip ファイルアーカイブを使用して C# Lambda 関数を構築し、デプロイする

PowerShell

- .zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする

コンソールのコードエディタの使用

コンソールで、単一のソースファイルを含む Lambda 関数が作成されます。スクリプト言語の場合、このファイルを編集し、組み込みのコードエディタを使用してファイルをさらに追加することができます。変更を保存するには [保存] を選択します。コードを実行するには、[Test] (テスト) を選択します。

> **Note**
>
> Lambda コンソールでは、AWS Cloud9 を使用して、ブラウザに統合開発環境を提供します。また、AWS Cloud9 を使用して、独自の環境で Lambda 関数を開発することもできます。詳細については、AWS Cloud9 ユーザーガイドの「AWS Toolkit を使用した AWS Lambda 関数の使用」を参照してください。

関数コードを保存すると、Lambda コンソールは .zip ファイルアーカイブのデプロイパッケージを作成します。コンソール外で (SDE を使用して) 関数コードを開発するときは、デプロイパッケージを作成して、Lambda 関数にコードをアップロードします。

関数コードの更新

スクリプト言語 (Node.js, Python, and Ruby) の場合は、組み込みのコードエディタで関数コードを編集することができます。コードが 3 MBを超える場合、またはライブラリを追加する必要がある場合、またはエディタでサポートされていない言語 (Java, Go, C#) の場合は、関数コードを .zip アーカイブとしてアップロードする必要があります。.zip ファイルアーカイブが 50 MB 未満の場合は、ローカルマシンから .zip ファイルアーカイブをアップロードできます。ファイルが 50 MB を超える場合は、Amazon S3 バケットから関数にファイルをアップロードします。

関数コードを.zip アーカイブとしてアップロードするには

1. Lambda コンソールの [関数ページ] を開きます。

2. 更新する関数を選択し、[Code] (コード) タブを選択します。
3. [Code source (コードソース)] で、[Upload from (アップロード元)] を選択します。
4. [.zip file (.zip ファイル)]、[Upload (アップロード)] の順に選択します。
   - ファイルチューザで新しいイメージバージョンを選択し、[Open (開く)]、[Save (保存)] の順に選択します。
5. (ステップ 4 の代わりに) Amazon S3 の場所を選択します。
   - テキストボックスに .zip ファイルアーカイブの S3 リンク URL を入力し、[Save] (保存) を選択します。

ランタイムの変更

新しいランタイムを使用するように関数の設定を更新する場合は、新しいランタイムとの互換性を確保するために関数コードを更新する必要がある場合があります。別のランタイムを使用するように関数設定を更新した場合は、ランタイムおよびアーキテクチャと互換性のある新しい関数コードを提供する必要があります。関数コードのデプロイパッケージを作成する方法については、関数が使用するランタイムのハンドラーページを参照してください。

ランタイムを変更する

1. Lambda コンソールの [関数ページ] を開きます。
2. 更新する関数を選択し、[Code] (コード) タブを選択します。
3. コードエディタの下にある [Runtime settings] (ランタイム設定) セクションまで下にスクロールします。
4. [編集] を選択します。
   a. [Runtime] (ランタイム) で、ランタイム識別子を選択します。
   b. [Handler] (ハンドラ) で、関数のファイル名とハンドラを指定します。
   c. アーキテクチャで、関数に使用する命令セットアーキテクチャを選択します。
5. [Save] を選択します。

アーキテクチャの変更

命令セットアーキテクチャを変更する前に、関数コードがターゲットアーキテクチャと互換性があることを確認する必要があります。

Node.js、Python、Ruby を使用し、組み込みエディタで関数コードを編集する場合、既存のコードが変更されずに実行されることがあります。

ただし、.zip ファイルアーカイブのデプロイパッケージを使用して関数コードを提供する場合は、ターゲットランタイムおよび命令セットアーキテクチャ用に正しくコンパイルされ、構築された新しい .zip ファイルアーカイブを準備する必要があります。手順については、関数ランタイムのハンドラーページを参照してください。

命令セットアーキテクチャを変更するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 更新する関数を選択し、[Code] (コード) タブを選択します。
3. [Runtime settings] (ランタイム設定) で、[Edit] (編集) を選択します。
4. アーキテクチャで、関数に使用する命令セットアーキテクチャを選択します。
5. [Save] を選択します。

Lambda API の使用

.zip ファイルアーカイブを使用する関数を作成および設定するには、以下の API オペレーションを使用します。

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

AWS CloudFormation

AWS CloudFormation を使用して、.zip ファイルアーカイブを使用する Lambda 関数を作成できます。AWS CloudFormation テンプレートでは、Lambda 関数は `AWS::Lambda::Function` のリソースにより指定されます。`AWS::Lambda::Function` リソースのプロパティの詳細については、AWS CloudFormation ユーザーガイドの「AWS::Lambda::Function」を参照してください。

`AWS::Lambda::Function` リソースで、次のプロパティを設定して .zip ファイルアーカイブとして定義された関数を作成します。

- AWS::Lambda::Function
  - PackageType - `Zip` に設定します。

- コード — Amazon S3 バケット名と.zip ファイル名を S3Bucket および S3Key のフィールドに入力します。Node.js または Python では、Lambda 関数のインラインソースコードを提供できます。
- ランタイム — ランタイム値を設定します。
- アーキテクチャ — AWS Graviton2 プロセッサを使用するには、アーキテクチャ値をarm64に設定します。デフォルトでは、アーキテクチャ値はx86_64です。

コンテナイメージを使用した Lambda 関数の作成

AWS Lambda 関数のコードは、スクリプトまたはコンパイルされたプログラム、さらにそれらの依存関係で構成されます。デプロイパッケージを使用して、Lambda に関数コードをデプロイします。Lambda は、コンテナイメージと .zip ファイルアーカイブの 2 種類のデプロイパッケージをサポートします。

Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- Lambda の AWS ベースイメージを使用する

  AWS ベースイメージには、言語ランタイム、Lambda と関数コード間のやり取りを管理するランタイムインターフェイスクライアント、ローカルテスト用のランタイムインターフェイスエミュレーターがプリロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、当該言語のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することもできます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、当該言語のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

コンテナイメージから Lambda 関数を作成するには、イメージをローカルでビルドして、Amazon Elastic Container Registry (Amazon ECR) リポジトリにアップロードします。次に、関数の作成時にリポジトリ URI を指定します。Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

このページでは、Lambda 互換のコンテナイメージを作成するためのベースイメージタイプと要件について説明します。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

トピック

要件

「AWS Command Line Interface (AWS CLI) バージョン 2」および「Docker CLI」をインストールします。さらに、次の各要件にも注意してください。

- コンテナイメージには、Lambda Runtime API が実装されている必要があります。AWS からの、オープンソースのランタイムインターフェイスクライアントには、この API が実装されています。ランタイムインターフェイスクライアントを必要なベースイメージに追加することで、Lambda と互換性を持たせることができます。

- コンテナイメージは、読み取り専用のファイルシステム上で実行可能である必要があります。機能コードは、512 MB から 10,240 MB の間で、1 MB 刻みで書き込み可能な `/tmp` ディレクトリにアクセスできます。
- デフォルトの Lambda ユーザーは、関数コードを実行するために必要なすべてのファイルを読み取ることができる必要があります。Lambda は、最小特権のアクセス許可を持つデフォルトの Linux ユーザーを定義することで、セキュリティのベストプラクティスに従います。ご自身のアプリケーションコードが、外部の Linux ユーザーによる実行が制限されているファイルに依存していないことを確認します。
- Lambda では、Linux ベースのコンテナイメージのみがサポートされます。
- Lambda は、マルチアーキテクチャのベースイメージを提供します。ただし、関数用に構築するイメージは、アーキテクチャの 1 つだけをターゲットにする必要があります。Lambda は、マルチアーキテクチャのコンテナイメージを使用する関数をサポートしません。

Lambda の AWS ベースイメージを使用する

Lambda に AWS ベースイメージの 1 つを使用して、関数コードのコンテナイメージを構築します。ベースイメージには、Lambda でコンテナイメージを実行するために必要な言語ランタイムおよびその他のコンポーネントがプリロードされます。関数コードと依存関係をベースイメージに追加し、コンテナイメージとしてパッケージ化します。

AWS では、Lambda 用の AWS ベースイメージの更新を定期的に実施しています。Dockerfile の FROM プロパティにイメージ名が含まれている場合、Docker クライアントは Amazon ECR リポジトリから最新バージョンのイメージを取り出します。更新されたベースイメージを使用するには、コンテナイメージを再ビルドして、関数のコードを更新する必要があります。

Node.js 20、Python 3.12、Java 21、AL2023 以降のベースイメージは、Amazon Linux 2023 の最小コンテナイメージに基づいています。以前のベースイメージでは Amazon Linux 2 が使用されています。AL2023 ランタイムには、デプロイのフットプリントが小さいことや、`glibc` などのライブラリのバージョンが更新されていることなど、Amazon Linux 2 に比べていくつかの利点があります。

AL2023 ベースのイメージでは、Amazon Linux 2 のデフォルトのパッケージマネージャである `yum` の代わりに `microdnf` (`dnf` としてシンボリックリンク) がパッケージマネージャとして使用されています。`microdnf` は `dnf` のスタンドアロン実装です。AL2023 ベースのイメージに含まれるパッケージのリストについては、「Comparing packages installed on Amazon Linux 2023 Container Images」の「Minimal Container」列を参照してください。AL2023 と Amazon Linux 2 の違いの詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。

> **Note**
>
> AWS Serverless Application Model (AWS SAM) を含む AL2023 ベースのイメージをローカルで実行するには、Docker バージョン 20.10.10 以降を使用する必要があります。

AWS ベースイメージを使用してコンテナイメージを構築するには、お好みの言語での手順を選択してください。

- Node.js
- TypeScript (Node.js ベースイメージを使用)
- Python
- Java
- Go
- .NET
- Ruby

AWS の OS 専用ベースイメージを使用する

AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、当該言語のランタイムインターフェイスクライアントをイメージに含める必要があります。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
al2023	OS 専用ランタイム	Amazon Linux 2023	GitHub の OS 専用ランタイムの Dockerfile	
al2	OS 専用ランタイム	Amazon Linux 2	GitHub の OS 専用ランタイムの Dockerfile	

Amazon Elastic コンテナレジストリ公開ギャラリー: gallery.ecr.aws/lambda/provided

非 AWS ベースイメージを使用する

Lambda では、次のいずれかのイメージマニフェストの形式に準拠するイメージをサポートしています。

- Docker Image Manifest V2 Schema 2 (Docker バージョン 1.10 以降で使用)
- Open Container Initiative (OCI) 仕様 (v1.0.0 以降)

Lambda は、すべてのレイヤーを含めて最大 10 GB の非圧縮のイメージサイズをサポートします。

> **Note**
>
> イメージに Lambda との互換性を持たせるには、当該言語のランタイムインターフェイスクライアントをイメージに含める必要があります。

ランタイムインターフェイスクライアント

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間のやり取りを管理する Lambda Runtime API を拡張する必要があります。AWS では、オープンソースのランタイムインターフェイスクライアントを次の言語で提供しています。

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust — Rust ランタイムクライアントは実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

使用している言語に AWS の提供するランタイムインターフェイスクライアントがない場合、独自のランタイムインターフェイスクライアントを作成する必要があります。

Amazon ECR のアクセス許可

コンテナイメージから Lambda 関数を作成する前に、そのイメージをローカルでビルドし、Amazon ECR リポジトリにアップロードする必要があります。関数の作成時に、Amazon ECR リポジトリ URI を指定します。

関数を作成するユーザーまたはロールのアクセス許可に、`GetRepositoryPolicy` および `SetRepositoryPolicy` が含まれていることを確認してください。

例えば、IAM コンソールを使用して、次のポリシーでロールを作成します。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Amazon ECR リポジトリポリシー

Amazon ECR のコンテナイメージと同じアカウント内の関数の場合、Amazon ECR リポジトリポリシーに `ecr:BatchGetImage` および `ecr:GetDownloadUrlForLayer` のアクセス許可を追加できます。次の例は、最小ポリシーを示しています。

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
```

```
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

Amazon ECR のリポジトリへのアクセス許可について、詳しくは「Amazon Elastic Container Registry ユーザーガイド」の「プライベートリポジトリポリシー」を参照してください。

Amazon ECR リポジトリにこれらの権限が含まれていない場合、Lambda は `ecr:BatchGetImage` および `ecr:GetDownloadUrlForLayer` をコンテナイメージリポジトリのアクセス許可に追加します。Lambda は、Lambda を呼び出すプリンシパルに `ecr:getRepositoryPolicy` および `ecr:setRepositoryPolicy` のアクセス許可がある場合にのみ、これらのアクセス許可を追加できます。

Amazon ECR リポジトリへのアクセス許可を表示または編集するには、「Amazon Elastic Container Registry ユーザーガイド」の「プライベートリポジトリポリシーステートメントの設定」を参照してください。

Amazon ECR クロスアカウント許可

同じリージョン内の別のアカウントで、アカウントが所有するコンテナイメージを使用する関数を作成できます。次の例では、Amazon ECR リポジトリへのアクセス許可ポリシーで、アカウント番号 123456789012 にアクセス権を付与するために、次のステートメントが必要です。

- CrossAccountPermission — アカウント 123456789012 が、この ECR リポジトリからイメージを使用する Lambda 関数を作成および更新できるようにします。
- LambdaECRImageCrossAccountRetrievalPolicy – Lambda は、関数が長期間呼び出されない場合、最終的に関数の状態を非アクティブに設定します。このステートメントは、123456789012 が所有する関数に代わって Lambda が最適化およびキャッシュのためにコンテナイメージを取得できるようにするために必要です。

Example クロスアカウント許可をリポジトリに追加する

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
```

```
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      }
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "StringLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      }
    }
  ]
}
```

複数のアカウントにアクセス権を付与するには、CrossAccountPermission ポリシーの [Principal] (プリンシパル) リストと LambdaECRImageCrossAccountRetrievalPolicy の [Condition] (条件) 評価リストにアカウント ID を追加します。

AWS Organization で複数のアカウントを扱う場合は、ECR の許可ポリシーで各アカウント ID を列挙することをお勧めします。このアプローチは、IAM ポリシーで狭い範囲の許可を設定するという AWS セキュリティのベストプラクティスに沿ったものです。

Lambda のアクセス許可に加えて、関数を作成するユーザーまたはロールには、BatchGetImage および GetDownloadUrlForLayer アクセス許可も必要です。

関数のライフサイクル

新規または更新済みのコンテナイメージをアップロードすると、Lambda は、関数が呼び出しを処理する前にイメージを最適化します。最適化プロセスには数秒かかる場合があります。この関数

は、プロセスが完了するまで Pending 状態のままです。その後、関数は Active 状態に移行します。Pending 状態のときに関数を呼び出すことはできますが、関数に対する他のオペレーションは失敗します。イメージ更新の進行中に発生した呼び出しは、以前のイメージのコードを実行します。

数週間にわたって関数が呼び出されない場合、Lambda は最適化されたバージョンを再利用し、関数は Inactive 状態に移行します。関数を再度アクティブにするには、関数を呼び出す必要があります。Lambda は最初の呼び出しを拒否し、関数は Lambda がイメージを再最適化するまで Pending 状態に入ります。その後、関数は Active 状態に戻ります。

Lambda は、Amazon ECR リポジトリから関連するコンテナイメージを定期的に取得します。対応するコンテナイメージが Amazon ECR に存在しなくなった場合、またはアクセス許可が失効した場合、関数は Failed 状態になり、Lambda が関数呼び出しに対して失敗を返します。

Lambda API を使用して、関数の状態に関する情報を取得できます。詳細については、「Lambda 関数の状態」を参照してください。

Lambda 関数の呼び出しメソッドについて

「Lambda 関数をデプロイ」した後、いくつかの方法で呼び出すことができます。

- Lambda コンソール - Lambda コンソールを使用し、テストイベントをすばやく作成して関数を呼び出します。
- AWS SDK - AWS SDK を使用して関数をプログラムによって呼び出します。
- 「呼び出し」 API - Lambda 呼び出し API を使用して関数を直接呼び出します。
- AWS Command Line Interface (AWS CLI) — `aws lambda invoke` AWS CLI コマンドを使用して、コマンドラインから関数を直接呼び出します。
- 関数 URL の HTTP(S) エンドポイント — 関数 URL を使用し、関数を呼び出すために使用できる専用の HTTP(S) エンドポイントを作成します。

これらのメソッドはすべて関数を直接呼び出す方法です。Lambda では、アプリケーションの他の場所で発生するイベントに基づいて関数を呼び出すことが一般的な使用例です。一部のサービスは、新しいイベントごとに Lambda 関数を呼び出すことができます。これはトリガーと呼ばれます。ストリームおよびキューベースのサービスの場合、Lambda はレコードのバッチで関数を呼び出します。これはイベントソースマッピングと呼ばれます。

関数を呼び出す際は、同期的に呼び出すか非同期的に呼び出すかを選択できます。同期呼び出しでは、イベントを処理する関数を待ってレスポンスを返します。非同期呼び出しでは、Lambda はイベントをキューに入れて処理し、すぐにレスポンスを返します。呼び出し API の `InvocationType` リクエストパラメータは、Lambda が関数を呼び出す方法を判定します。`RequestResponse` の値は同期呼び出しを示し、`Event` の値は非同期呼び出しを示します。

関数の呼び出しでエラーが発生したら、同期呼び出しの場合は応答のエラーメッセージを確認し、手動で呼び出しを再試行してください。非同期呼び出しの場合、Lambda は再試行を自動的に処理し、呼び出しレコードを 宛先 に送信できます。

同期呼び出し

関数を同期的に呼び出すと、Lambda は関数を実行し、レスポンスを待ちます。関数の実行が終了すると、Lambda は、呼び出された関数のバージョンなどの追加データとともに、関数のコードからのレスポンスを返します。AWS CLI を使用して関数を同期的に呼び出すには、`invoke` コマンドを使用します。

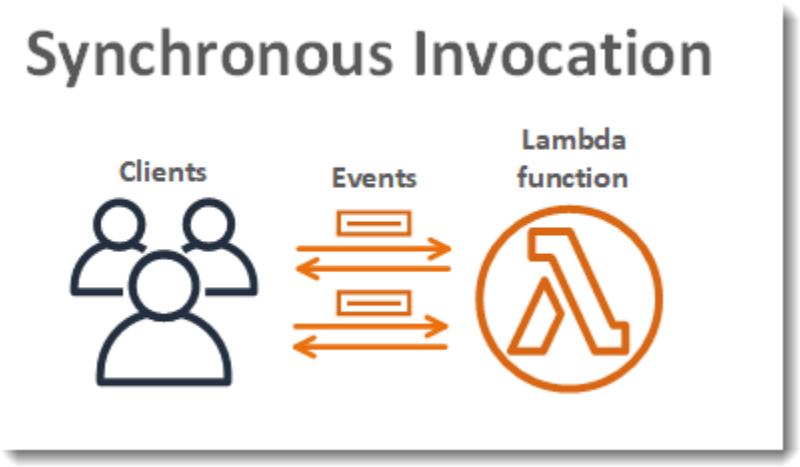
```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{ "key": "value" }' response.json
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

次の図は、Lambda 関数を同期的に呼び出すクライアントを示しています。Lambda はイベントを関数に直接送信し、関数の応答を呼び出し側に返します。



`payload` は、JSON 形式のイベントを含む文字列です。AWS CLI が関数からのレスポンスを書き込むファイルの名前は `response.json` です。関数がオブジェクトまたはエラーを返す場合、レスポ

ンス本文は JSON 形式のオブジェクトまたはエラーです。関数がエラーなしで終了した場合、レスポンス本文は ですnull。

> **Note**
>
> Lambda は外部拡張機能の完了を待たずにレスポンスを送信します。外部拡張機能は、実行環境内の独立したプロセスとして実行され、関数の呼び出しが完了した後も引き続き実行されます。詳細については、「Lambda 拡張機能を使用して Lambda 関数を補強する」を参照してください。

端末に表示されるコマンドの出力には、Lambda からのレスポンスのヘッダーにある情報が含まれます。これには、イベントを処理したバージョン (エイリアスを使用する場合に役立つ)、および Lambda. から返されるステータスコードが含まれます。Lambda が関数を実行できた場合は、関数がエラーを返したとしても、ステータスコードは 200 です。

> **Note**
>
> タイムアウトが長い関数では、同期呼び出し中にレスポンスを待機している間に、クライアントが切断される場合があります。HTTP クライアント、SDK、ファイアウォール、プロキシ、またはオペレーティングシステムを構成して、タイムアウトまたはキープアライブ設定での長い接続を許可するようにしてください。

Lambda が関数を実行できない場合は、エラーが出力に表示されます。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload value response.json
```

次のような出力が表示されます。

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
 at [Source: (byte[])"value"; line: 1, column: 11]
```

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
```

```
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

パラメータ、ヘッダー、エラーの完全なリストを含む Invoke API の詳細については、「呼び出し」を参照してください。

関数を直接呼び出す場合は、エラーレスポンスを確認し、再試行することができます。また、AWS CLI と AWS SDK は、クライアントのタイムアウト、スロットル、およびサービスエラーで自動的に再試行します。詳細については、「Lambda での再試行動作について」を参照してください。

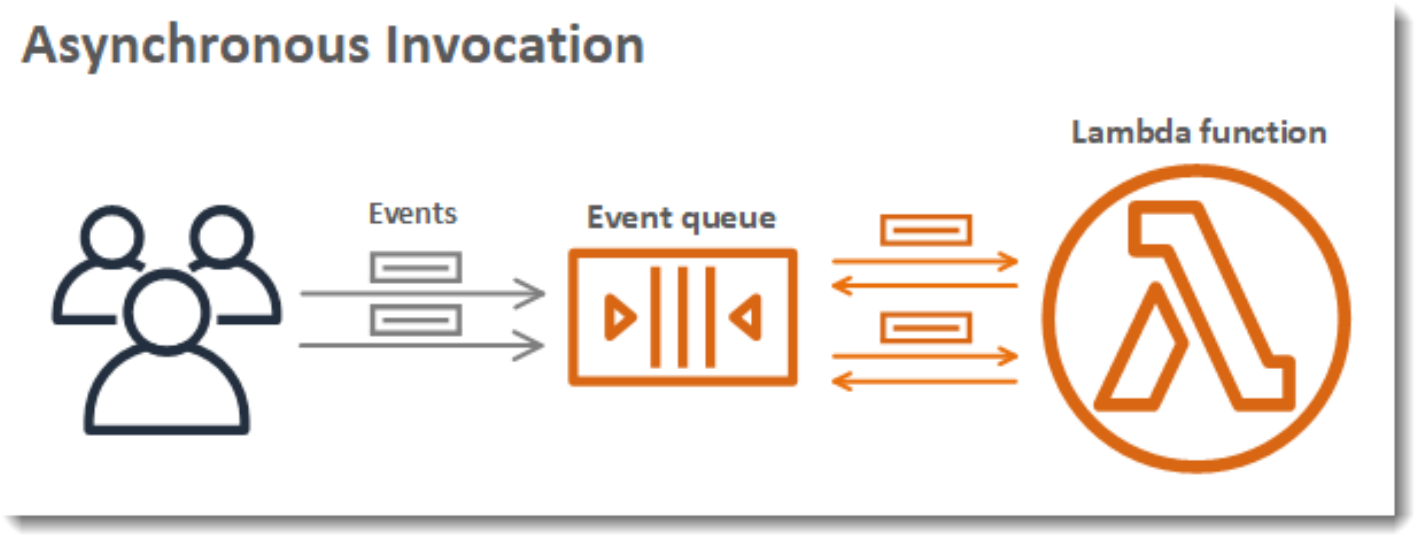
非同期呼び出し

Amazon Simple Storage Service (Amazon S3) や Amazon Simple Notification Service (Amazon SNS) などの複数の AWS のサービス では、関数を非同期的に呼び出してイベントを処理します。関数を非同期的に呼び出す場合は、関数コードからのレスポンスを待機しません。イベントを Lambda に渡すと、Lambda が残りを処理します。Lambda がエラーを処理する方法を設定し、Amazon Simple Queue Service (Amazon SQS) または Amazon EventBridge (EventBridge) などのダウンストリームリソースに呼び出しレコードを送信して、アプリケーションのコンポーネントをつなぎ合わせることができます。

セクション

Lambda が非同期呼び出しを処理する方法

次の図は、クライアントによる Lambda 関数の非同期的呼び出しを示しています。Lambda は、イベントを関数に送信する前にキューに入れます。



非同期呼び出しの場合、Lambda はリクエストをキューに入れ、追加情報のない成功のレスポンスを返します。別のプロセスがキューからイベントを読み取って関数に送信します。関数を非同期的に呼び出すには、呼び出しタイプパラメータを Event に設定します。

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
{
    "StatusCode": 202
}
```

出力ファイル (`response.json`) には情報は含まれないものの、このコマンドを実行すると作成されます。Lambda がイベントをキューに追加することができない場合、エラーメッセージがコマンド出力に表示されます。

Lambda は関数の非同期イベントキューを管理し、エラー発生時に再試行を行います。関数から エラーが返された場合、Lambda はその関数をさらに 2 回再試行します。その際、最初の 2 回の試行の間に 1 分間、2 回目と 3 回目の間に 2 分間の待機時間があります。関数エラーには、関数のコードによって返されるエラーと、タイムアウトなど関数のランタイムによって返されるエラーが含まれます。

関数にすべてのイベントを処理するために十分な同時実行数がない場合は、追加のリクエストはスロットリングされます。スロットルエラー (429) およびシステムエラー (500 番台) の場合、Lambda はイベントをキューに返し、最大 6 時間、関数を再度実行しようとします。再試行間隔は、最初の試行後 1 秒から最大 5 分まで指数関数的に増加します。キューに多くのエントリが含まれている場合、Lambda は再試行の間隔を長くして、キューからイベントを読み取る速度を低下させます。

関数がエラーを返さない場合でも、キュー自体は最終的に一貫しているため、同じイベントを Lambda から複数回受信する可能性があります。関数が受信するイベントに対応できない場合、イベントは関数に送信されずにキューから削除される場合があります。関数コードが重複イベントを適切に処理し、すべての呼び出しを処理するのに十分な同時実行数があることを確認してください。

キューが非常に長い場合、新しいイベントは Lambda によって関数に送信される前に期限切れになる場合があります。期限切れになったイベントや、すべての処理試行に失敗したイベントは Lambda によって破棄されます。関数のエラー処理を設定して、Lambda による再試行の回数を減らしたり、未処理のイベントをより速やかに破棄したりできます。

また、呼び出しレコードを別のサービスに送信するように Lambda を設定することもできます。Lambda は、非同期呼び出しの次の送信先をサポートしています。SQS FIFO キューと SNS FIFO トピックはサポートされていないことに注意してください。

- Amazon SQS — 標準の SQS キュー。
- Amazon SNS – 標準 SNS トピック。
- AWS Lambda — Lambda 関数。
- Amazon EventBridge - Amazon EventBridge イベントバス。

呼び出しレコードには、JSON 形式のリクエストとレスポンスに関する詳細が含まれます。処理に成功したイベント用とすべての処理試行に失敗したイベント用に別々の送信先を設定できます。または、破棄されたイベント用に標準 Amazon SQS キューまたは標準 Amazon SNS トピックをデッドレターキューとして設定することもできます。デッドレターキューの場合、Lambda はイベントのコンテンツのみを送信し、レスポンスの詳細は送信しません。

設定した送信先に Lambda がレコードを送信できない場合、Lambda は Amazon CloudWatch に `DestinationDeliveryFailures` メトリクスを送信します。これは、設定に Amazon SQS FIFO キューや Amazon SNS FIFO トピックなど、サポートされていない送信先タイプが含まれている場合に発生する可能性があります。また、配信エラーはアクセス許可エラーが原因で発生する可能性があります。Lambda 呼び出しメトリクスの詳細については、「呼び出しメトリクス」を参照してください。

> **Note**
>
> 関数がトリガーされないようにするには、その関数の予約済同時実行数をゼロに設定します。非同期で呼び出された関数の予約された同時実行数をゼロに設定すると、Lambda は設定したデッドレターキューまたは障害発生時のイベント送信先に再試行なしで新しいイベントの送信を開始します。予約された同時実行数をゼロに設定している間に送信されたイベントを処理するには、デッドレターキューまたは障害発生時のイベント送信先からイベントを消費する必要があります。

非同期呼び出しのエラー処理の設定

Lambda コンソールを使用して、関数、バージョン、またはエイリアスのエラー処理を設定します。

エラー処理を設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定]、[Asynchronous invocation (非同期呼び出し)] の順に選択します。
4. [Asynchronous invocation (非同期呼び出し)] で、[Edit (編集)] を選択します。
5. 以下を設定します。
   - [Maximum age of event] (イベントの最大有効期間) - Lambda が非同期イベントキューにイベントを保持する最大時間 (最大 6 時間)。
   - [Retry attempts] (再試行) - 関数がエラーを返したときに Lambda が再試行する回数 (0 ～ 2)。
6. [Save] を選択します。

呼び出しイベントが最大有効期間を超えるか、すべての再試行に失敗すると、Lambda はそのイベントを破棄します。破棄されたイベントのコピーを保持するには、失敗したイベントの送信先を設定します。

非同期呼び出しの送信先の設定

非同期呼び出しのレコードを保持するには、関数に送信先を追加します。成功した呼び出しと失敗した呼び出しのいずれかを送信先に送るように選択できます。各関数に複数の送信先を設定できるため、成功したイベントと失敗したイベントの送信先を別々に設定できます。送信先に送られる各レコードは、呼び出しに関する詳細を含む JSON ドキュメントです。エラー処理の設定と同様に、関数、関数のバージョン、またはエイリアスに対して送信先を設定できます。

> **Note**
>
> また、Amazon Kinesis、Amazon DynamoDB、セルフマネージド Apache Kafka、および Amazon MSK の各タイプのイベントソースマッピングについて、失敗した呼び出しのレコードを保持することもできます。

以下の表は、非同期呼び出しレコードでサポートされている送信先の一覧です。Lambda が選択した送信先にレコードを正常に送信するには、関数の実行ロールにも関連するアクセス権限が含まれていることを確認してください。この表では、各送信先タイプで JSON 呼び出しレコードを受け取る方法についても説明しています。

送信先タイプ	必要なアクセス許可	宛先固有の JSON 形式
Amazon SQS キュー	sqs:SendMessage	Lambda は呼び出しレコードを `Message` として宛先に渡します。
Amazon SNS トピック	sns:Publish	Lambda は呼び出しレコードを `Message` として宛先に渡します。
Lambda 関数	InvokeFunction	Lambda は呼び出しレコードをペイロードとして関数に渡します。
EventBridge	events:PutEvents	• Lambda は呼び出しレコードを PutEvents 呼び出しの `detail` として渡します。 • 「`source`」 イベントフィールドの値は 「`lambda`」 です。 • `detail-type` イベントフィールドの値は、「Lambda 関数の呼び出し結果 - 成功」または「Lambda 関数の呼び出し結果 - 失敗」のいずれかです。 • 「`resource`」 イベントフィールドには、関数と宛先の Amazon リソースネーム (ARN) が含まれます。 • 他のイベントフィールドについては、Amazon

送信先タイプ	必要なアクセス許可	宛先固有の JSON 形式
		EventBridge イベントを参照してください。

次の例は、関数エラーが原因で 3 つの処理試行に失敗したイベントの呼び出しレコードを示しています。呼び出しレコードには、イベント、レスポンス、レコードが送信された理由に関する詳細が含まれます。

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

次のステップでは、Lambda コンソールを使用して関数の送信先を設定する方法について説明します。

非同期呼び出しレコードの送信先の設定

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [機能の概要] で、[送信先を追加] を選択します。
4. [Source (送信元)] で、[Asynchronous invocation (非同期呼び出し)] を選択します。
5. [条件] で、以下のオプションから選択します。
   - [On failure] (失敗時) - イベントがすべての処理試行に失敗した場合、または最大有効期間を超えた場合に、レコードを送信します。
   - [On success] (正常) - 関数が非同期呼び出しを正常に処理したときに、レコードを送信します。
6. [送信先タイプ] で、呼び出しレコードを受信するリソースのタイプを選択します。
7. [送信先] で、リソースを選択します。
8. [Save] を選択します。

呼び出しが条件に一致すると、Lambda は呼び出しに関する詳細を含む JSON ドキュメントを送信先に送ります。

宛先固有の JSON 形式

- Amazon SQS と Amazon SNS (「SnsDestination」 と 「SqsDestination」) の場合、呼び出しレコードは 「Message」 として送信先に渡されます。
- Lambda 「LambdaDestination」 の場合、呼び出しレコードはペイロードとして関数に渡されます。
- EventBridge 「EventBridgeDestination」 の場合、呼び出しレコードは PutEvents の呼び出しで 「detail」 として渡されます。「source」 イベントフィールドの値は 「lambda」 です。「detail-type」イベントフィールドの値は、Lambda 関数の呼び出し結果 - 成功または Lambda 関数の呼び出し結果 - 失敗のいずれかです。「resource」 イベントフィールドには、関数と宛先の Amazon リソースネーム (ARN) が含まれます。他のイベントフィールドについては、Amazon EventBridge イベントを参照してください。

次の例は、関数エラーが原因で 3 つの処理試行に失敗したイベントの呼び出しレコードを示しています。

Example 呼び出しレコード

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

呼び出しレコードには、イベント、レスポンス、レコードが送信された理由に関する詳細が含まれます。

送信先へのリクエストのトレース

AWS X-Ray を使用すると、各リクエストがキューに入れられ、Lambda 関数によって処理され、送信先サービスに渡される過程の接続されたビューを確認できます。関数または関数を呼び出すサービスの X-Ray トレーシングを有効にすると、Lambda は X-Ray ヘッダーをリクエストに追加し、ヘッダーを送信先サービスに渡します。アップストリームサービスからのトレースは、ダウンストリーム Lambda 関数および宛先サービスからのトレースに自動的にリンクされ、アプリケーション全体のエンドツーエンドビューを作成します。トレースの詳細については、「AWS X-Ray を使用した Lambda 関数呼び出しの視覚化」を参照してください。

非同期呼び出し設定 API

AWS CLI または AWS SDK で非同期呼び出し設定を管理するには、以下の API オペレーションを使用します。

- PutFunctionEventInvokeConfig
- GetFunctionEventInvokeConfig
- UpdateFunctionEventInvokeConfig
- ListFunctionEventInvokeConfigs
- DeleteFunctionEventInvokeConfig

AWS CLI で非同期呼び出しを設定するには、put-function-event-invoke-config コマンドを使用します。以下の例では、最大イベント有効期間が 1 時間で再試行なしの関数を設定しています。

```
aws lambda put-function-event-invoke-config --function-name error \
--maximum-event-age-in-seconds 3600 --maximum-retry-attempts 0
```

次のような出力が表示されます。

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

put-function-event-invoke-config コマンドは、関数、バージョン、またはエイリアスの既存の設定を上書きします。他の設定をリセットしないでオプションを設定するには、update-function-event-invoke-config を使用します。以下の例では、イベントを処理できない場合に、destination という名前の標準 SQS キューにレコードを送信するように Lambda を設定します。

```
aws lambda update-function-event-invoke-config --function-name error \
```

```
--destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-2:123456789012:destination"}}'
```

以下の出力が表示されます。

```
{
    "LastModified": 1573687896.493,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {
            "Destination": "arn:aws:sqs:us-east-2:123456789012:destination"
        }
    }
}
```

デッドレターキュー

障害発生時の送信先に代わる方法として、配信不能キューを使用して関数を設定し、破棄されたイベントを保存してさらに処理できます。配信不能キューは、イベントがすべての処理試行に失敗した場合や処理されずに期限切れになった場合に使用されるという点で、障害発生時の送信先と同じように動作します。ただし、配信不能キューは関数のバージョン固有の設定の一部であるため、バージョンを公開すると口ックされます。また、障害発生時の送信先は、追加のターゲットをサポートし、関数のレスポンスに関する詳細を呼び出しレコードに含めます。

デッドレターキュー内のイベントを再処理するには、デッドレターキューを Lambda 関数のイベントソースとして設定します。または、イベントを手動で取得することもできます。

デッドレターキューには標準 Amazon SQS キューまたは標準 Amazon SNS トピックを選択できます。FIFO キューと Amazon SNS FIFO トピックはサポートされていません。キューまたはトピックがない場合は、新規に作成します。ユースケースに一致するターゲットタイプを選択します。

- Amazon SQS キュー - キューは、失敗したイベントが取得されるまでそれらを保持します。Lambda 関数や CloudWatch アラームなどの単一のエンティティが失敗したイベントを処理することが予想される場合は、Amazon SQS 標準キューを選択します。詳細については、「Amazon SQS での Lambda の使用」を参照してください。

  Amazon SQS コンソールで、キューを作成します。

- Amazon SNS トピック - トピックは、失敗したイベントを 1 つまたは複数の送信先に中継します。複数のエンティティが失敗したイベントに対して動作することが予想される場合は、Amazon SNS 標準トピックを選択します。例えば、E メールアドレス、Lambda 関数、および/または HTTP エンドポイントにイベントを送信するようにトピックを設定できます。詳細については、Amazon SNS 通知を使用した Lambda 関数の呼び出し を参照してください。

  Amazon SNS コンソールでトピックを作成します。

キューまたはトピックにイベントを送信するには、関数に追加のアクセス権限が必要です。関数の実行ロールに必要なアクセス権限と持つポリシーを追加します。

- Amazon SQS - sqs:SendMessage
- Amazon SNS - sns:Publish

ターゲットキューまたはトピックがカスタマー管理のキーで暗号化されている場合、その実行ロールはキーのリソースベースのポリシーのユーザーでもある必要があります。

ターゲットを作成し、関数の実行ロールを更新した後、デッドレターキューを関数に追加します。同じターゲットにイベントを送信するように複数の関数を設定できます。

デッドレターキューを設定するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定]、[Asynchronous invocation (非同期呼び出し)] の順に選択します。
4. [Asynchronous invocation (非同期呼び出し)] で、[Edit (編集)] を選択します。
5. [DLQ resource] (DLQ リソース) を [Amazon SQS] または [Amazon SNS] に設定します。
6. ターゲットとなるキューまたはトピックを選択します。
7. [Save] を選択します。

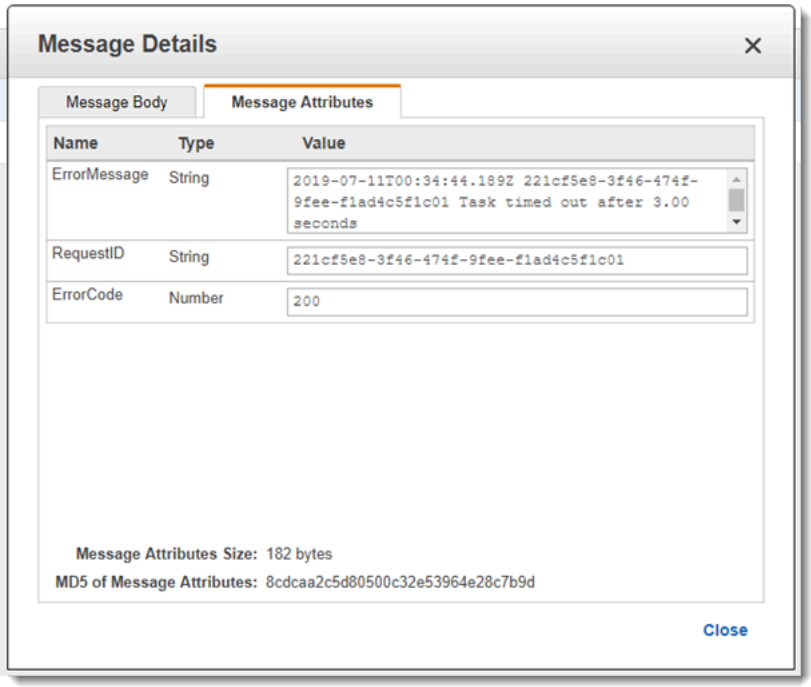
AWS CLI を使用してデッドレターキューを設定するには、`update-function-configuration` コマンドを使用します。

```
aws lambda update-function-configuration --function-name my-function \
--dead-letter-config TargetArn=arn:aws:sns:us-east-2:123456789012:my-topic
```

Lambda は、イベントをそのまま属性の追加情報と共にデッドレターキューに送信します。この情報を使用して、関数が返したエラーを特定するか、イベントをログまたは AWS X-Ray トレースと関連付けることができます。

デッドレターキューメッセージの属性

- RequestID (文字列) - 呼び出しリクエストの ID。リクエスト ID は関数ログに表示されます。また、X-Ray SDK を使用して、トレース内の属性のリクエスト ID を記録することもできます。その後、X-Ray コンソールで、リクエスト ID によってトレースを検索できます。
- ErrorCode (数値) - HTTP ステータスコード。
- ErrorMessage (文字列) - エラーメッセージの最初の 1 KB。



Lambda がデッドレターキューにメッセージを送信できない場合、イベントは削除され、DeadLetterErrors メトリクスが発行されます。このような状況は、アクセス権限がない、またはメッセージの合計サイズがターゲットとなるキューやトピックの制限を超えてしまった場合に発生します。例えば、本文のサイズが 256 KB に近い Amazon SNS 通知が、エラーとなる関数をトリガーしたとします。その場合、Amazon SNS によって追加されたイベントデータと Lambda によって追加された属性の組み合わせによって、メッセージがデッドレターキューで許可されている最大サイズを超過することがあります。

Amazon SQS をイベントソースとして使用する場合、Amazon SQS キューでは Lambda 関数ではなくデッドレターキューを設定します。詳細については、「Amazon SQS での Lambda の使用」を参照してください。

Lambda がストリームおよびキューベースのイベントソースからのレコードを処理する方法

イベントソースマッピングは、ストリームおよびキューベースのサービスからアイテムを読み取り、レコードのバッチを使用して関数を呼び出す Lambda リソースです。以下のサービスは、イベントソースマッピングを使用して Lambda 関数を呼び出します。

- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- セルフマネージド Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)
- Amazon DocumentDB (MongoDB 互換) (Amazon DocumentDB)

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

イベントソースマッピングと直接トリガーの違い

一部の AWS サービスは、トリガーを使用して Lambda 関数を直接呼び出すことができます。これらのサービスはイベントを Lambda にプッシュし、指定されたイベントが発生すると即時に関数が呼び出されます。トリガーは、個別のイベントやリアルタイム処理に適しています。Lambda コンソールを使用してトリガーを作成すると、コンソールは対応する AWS サービスと連携して、そのサービスでイベント通知を設定します。実際には、トリガーは Lambda ではなく、イベントを生成するサービスによって保存および管理されます。トリガーを使用して Lambda 関数を呼び出すサービスの例をいくつか示します。

- Amazon Simple Storage Service (Amazon S3): オブジェクトがバケット内で作成、削除、または変更されたときに関数を呼び出します。詳細については、「チュートリアル: Amazon S3 トリガーを使用して Lambda 関数を呼び出す」を参照してください。
- Amazon Simple Notification Service (Amazon SNS): メッセージが SNS トピックに発行されたときに関数を呼び出します。詳細については、「チュートリアル: Amazon Simple Notification Service での AWS Lambda の使用」を参照してください。
- Amazon API Gateway: API リクエストが特定のエンドポイントに対して行われたときに関数を呼び出します。詳細については、「Amazon API Gateway エンドポイントを使用した Lambda 関数の呼び出し」を参照してください。

イベントソースマッピングは、Lambda サービス内で作成および管理される Lambda リソースです。イベントソースマッピングは、大量のストリーミングデータやキューからのメッセージを処理できるように設計されています。ストリームまたはキューからのレコードをバッチで処理すると、レコードを個別に処理するよりも効率的です。

バッチ処理動作

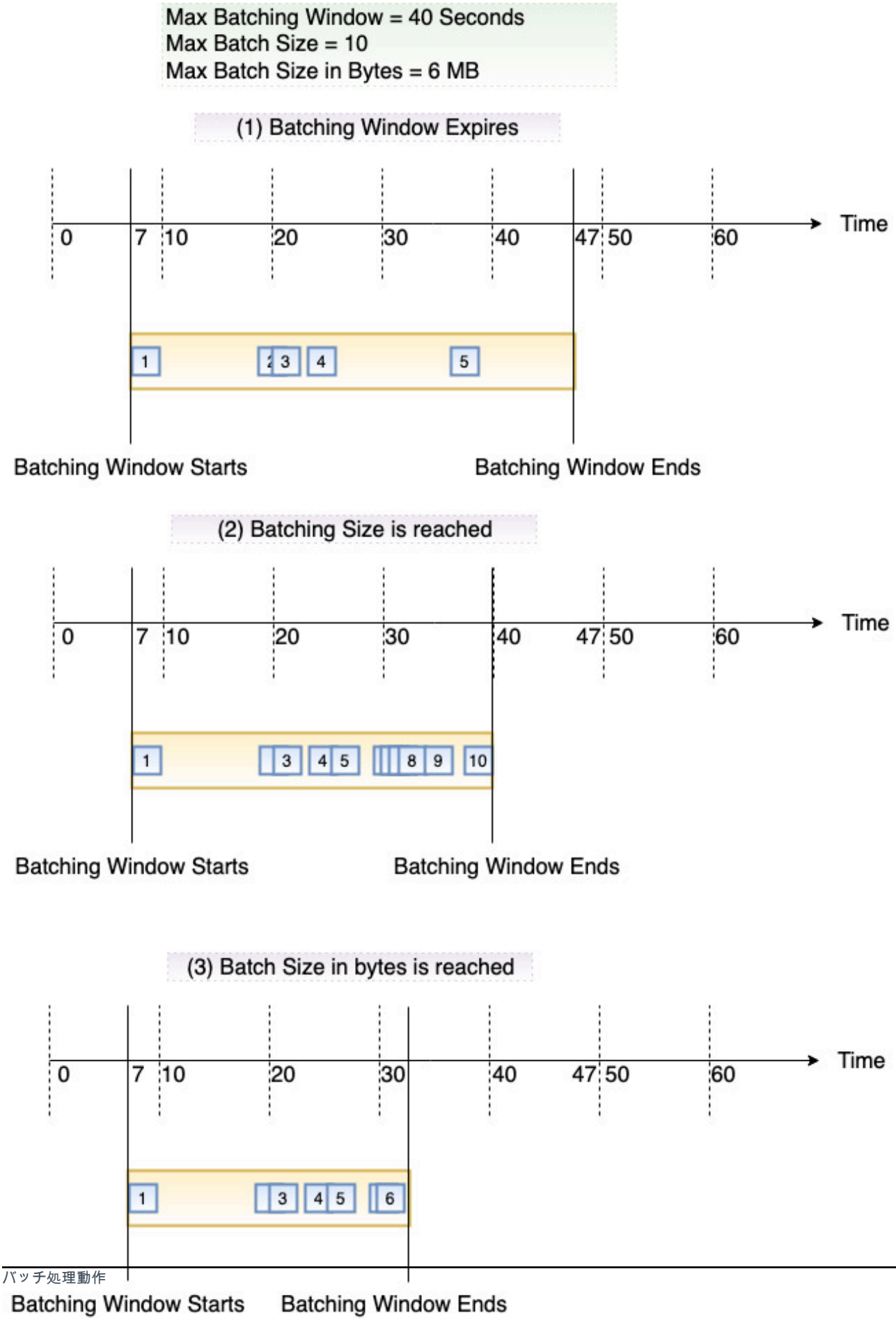
デフォルトで、イベントソースマッピングは、Lambda が関数に送信する単一のペイロードにレコードをまとめてバッチ処理します。バッチ処理の動作を微調整するには、バッチ処理ウィンドウ (MaximumBatchingWindowInSeconds) とバッチサイズ (BatchSize) を設定します。バッチ処理ウィンドウとは、レコードを単一のペイロードにまとめるための最大時間です。バッチサイズとは、単一のバッチ内にあるレコードの最大数です。Lambda は、以下の 3 つの条件のいずれかが満たされたときに関数を呼び出します。

- バッチ処理ウィンドウが最大値に到達した。デフォルトのバッチ処理ウィンドウの動作は、特定のイベントソースによって異なります。
  - Kinesis、DynamoDB、および Amazon SQS イベントソースの場合: デフォルトのバッチ処理ウィンドウは 0 秒です。これは、バッチサイズが満たされた場合、またはペイロードサイズ制限に達した場合にのみ、Lambda が関数にバッチを送信することを意味します バッチ処理ウィンドウを設定するには、`MaximumBatchingWindowInSeconds` を設定します。このパラメータは、秒単位で 0 秒から 300 秒までの任意の値に設定できます。バッチ処理ウィンドウを設定する場合、前の関数の呼び出しが完了するとすぐに次のウィンドウが開始されます。
  - Amazon MSK、セルフマネージド Apache Kafka、Amazon MQ、および Amazon DocumentDB イベントソースの場合: バッチ処理ウィンドウはデフォルトで 500 ミリ秒に設定されます。`MaximumBatchingWindowInSeconds` は、秒単位で 0 秒から 300 秒までの任意の値に設定できます。バッチ処理ウィンドウは、最初のレコードが到着するとすぐに開始されます。

> **Note**
>
> `MaximumBatchingWindowInSeconds` は秒単位の増分でしか変更できないため、いったん変更すると、デフォルトのバッチ処理ウィンドウである 500 ミリ秒に戻すことはできません。デフォルトのバッチ処理ウィンドウを復元するには、新しいイベントソースマッピングを作成する必要があります。

- バッチサイズに適合した。最小バッチサイズは 1 です。デフォルトのバッチサイズと最大バッチサイズは、イベントソースに応じて異なります。これらの値の詳細については、`CreateEventSourceMapping` API オペレーションの BatchSize の仕様を参照してください。
- ペイロードサイズが 6 MB に到達した。この上限を変更することはできません。

以下の図は、これら 3 つの条件を説明するものです。バッチ処理ウィンドウが `t = 7` 秒で開始されるとします。最初のシナリオでは、バッチ処理ウィンドウが 5 個のレコードを蓄積した後、`t = 47` 秒の時点で最大値の 40 秒に到達します。2 番目のシナリオでは、バッチ処理ウィンドウの期限が切れる前にバッチサイズが 10 個になるため、バッチ処理ウィンドウが早く終了します。3 番目のシナリオでは、バッチ処理ウィンドウの期限が切れる前に最大ペイロードサイズに到達するため、バッチ処理ウィンドウが早く終了します。


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
2
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

イベントソースマッピング API

AWS Command Line Interface (AWS CLI) または AWS SDK を使用してイベントソースを管理するには、以下の API 操作を使用できます。

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Amazon DynamoDB で AWS Lambda を使用する

> **Note**
>
> Lambda 関数以外のターゲットにデータを送信したい、または送信する前にデータをエンリッチしたいという場合は、「Amazon EventBridge Pipes」を参照してください。

AWS Lambda 関数を使用して、Amazon DynamoDB ストリームのレコードを処理します。DynamoDB Streams では、Lambda 関数を使用して、DynamoDB テーブルが更新されるたびに追加の作業を実行することができます。

Lambda はストリームからレコードを読み取り、関数を、ストリームレコードを含むイベントと共に同期的に呼び出します。Lambda はバッチ単位でレコードを読み取り、関数を呼び出してバッチからレコードを処理します。

セクション

- イベントの例
- ポーリングストリームとバッチストリーム
- ポーリングとストリームの開始位置
- DynamoDB Streams でのシャードの同時読み込み
- 実行ロールのアクセス許可
- アクセス許可を追加し、イベントソースマッピングを作成するには
- エラー処理

- Amazon CloudWatch メトリクス
- 時間枠
- バッチアイテムの失敗をレポートする
- Amazon DynamoDB Streams 設定パラメータ
- チュートリアル: Amazon DynamoDB Streams で AWS Lambda を使用する
- サンプル関数コード
- DynamoDB アプリケーション用の AWS SAM テンプレート

イベントの例

Example

```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
```

```
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

ポーリングストリームとバッチストリーム

Lambda は、レコードの DynamoDB ストリームにあるシャードを 1 秒あたり 4 回の基本レートでポーリングします。レコードが利用可能になると、Lambda は関数を呼び出し、結果を待機します。処理が成功すると、Lambda は、レコードをさらに受け取るまでポーリングを再開します。

デフォルトで、Lambda はレコードが使用可能になると同時に関数を呼び出します。Lambda がイベントソースから読み取るバッチにレコードが 1 つしかない場合、Lambda は関数に 1 つのレコードしか送信しません。少数のレコードで関数を呼び出さないようにするには、バッチ処理ウィンドウを設定することで、最大 5 分間レコードをバッファリングするようにイベントソースに指示できます。関数を呼び出す前に、Lambda は、完全なバッチを収集する、バッチ処理ウィンドウの期限が切れる、またはバッチが 6 MB のペイロード制限に到達するまでイベントソースからのレコードの読み取りを継続します。詳細については、「バッチ処理動作」を参照してください。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

DynamoDB ストリームの 1 つのシャードを複数の Lambda 呼び出しで同時に処理するには、ParallelizationFactor 設定を構成します。Lambda がシャードからポーリングする同時バッチの数は、1 (デフォルト) ~ 10 の並列化係数で指定できます。シャードごとの同時実行バッチの数を増やしても、Lambda はアイテム (パーティションおよびソートキー) レベルで順序立った処理を確実に行います。

ポーリングとストリームの開始位置

イベントソースマッピングの作成時および更新時のストリームのポーリングは、最終的に一貫性があることに注意してください。

- イベントソースマッピングの作成時、ストリームからのイベントのポーリングが開始されるまでに数分かかる場合があります。
- イベントソースマッピングの更新時、ストリームからのイベントのポーリングが停止および再開されるまでに数分かかる場合があります。

つまり、LATEST をストリームの開始位置として指定すると、イベントソースマッピングの作成または更新中にイベントを見逃す可能性があります。イベントを見逃さないようにするには、ストリームの開始位置を TRIM_HORIZON として指定します。

DynamoDB Streams でのシャードの同時読み込み

単一リージョンのテーブルがグローバルテーブルでない場合、同じ DynamoDB Streams のシャードから、同時に 2 つまでの Lambda 関数を読み込むように設計できます。この制限を超えると、リクエストのスロットリングが発生する場合があります。グローバルテーブルでは、リクエストのスロットリングを回避するために、同時関数の数を 1 に制限することをお勧めします。

実行ロールのアクセス許可

「AWSLambdaDynamoDBExecutionRole」 AWS 管理ポリシーには、Lambda が DynamoDB ストリームから読み取るために必要な許可が含まれています。この管理ポリシーを関数の実行ロールに追加します。

標準 SQS キューまたは標準 SNS トピックに失敗したバッチのレコードを送信するには、関数に追加の許可が必要になります。各送信先サービスには、次のように異なるアクセス許可が必要です。

- Amazon SQS - sqs:SendMessage
- Amazon SNS - sns:Publish

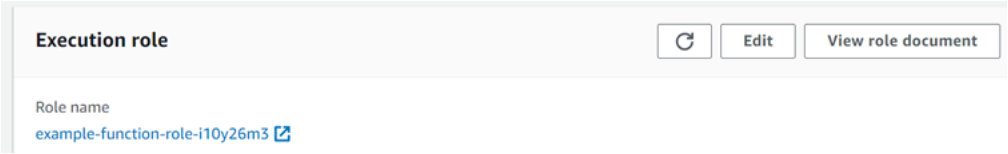
アクセス許可を追加し、イベントソースマッピングを作成するには

イベントソースマッピングを作成し、ストリームから Lambda 関数にレコードを送信するように Lambda に通知します。複数のイベントソースマッピングを作成することで、複数の Lambda 関数で同じデータを処理したり、1 つの関数で複数のストリームの項目を処理したりできます。

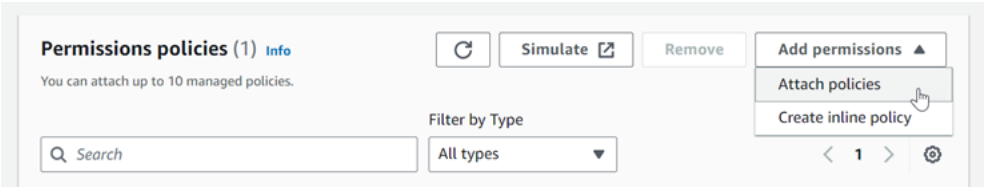
DynamoDB ストリームから読み取るように関数を設定するには、「AWSLambdaDynamoDBExecutionRole」 AWS 管理ポリシーを実行ロールにアタッチし、[DynamoDB] トリガーを作成します。

アクセス許可を追加してトリガーを作成するには

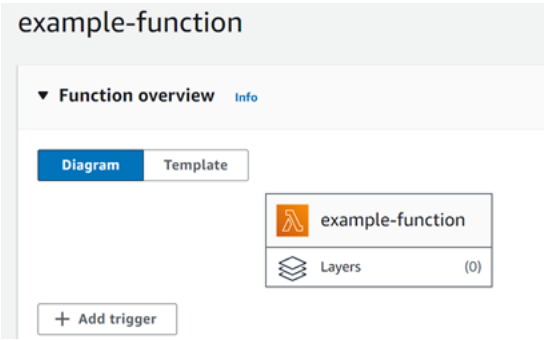
1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Configuration] (設定) タブを開き、次に [Permissions] (アクセス許可) をクリックします。
4. [実行ロール] で、実行ロールのリンクを選択します。このリンクを選択すると、IAM コンソールでロールが開きます。



5. [アクセス許可を追加]、[ポリシーをアタッチ] の順に選択します。



6. [検索] フィールドに AWSLambdaDynamoDBExecutionRole を入力します。実行ロールにポリシーを追加 関数が DynamoDB ストリームから読み取るために必要な許可を含む AWS 管理ポリシーです。このポリシーの詳細については、「AWS 管理ポリシーリファレンス」の「AWSLambdaDynamoDBExecutionRole」を参照してください。
7. Lambda コンソールの関数に戻ります。[関数の概要] で [トリガーを追加] をクリックします。



8. トリガーのタイプを選択します。
9. 必須のオプションを設定し、[Add] (追加) を選択します。

Lambda は、DynamoDB イベントソースの次のオプションをサポートしています。

イベントソースオプション

- DynamoDB テーブル - レコードの読み取り元の DynamoDB テーブル。
- バッチサイズ - 各バッチで関数に送信されるレコードの数。最大 10,000。Lambda は、イベントの合計サイズが同期呼び出しのペイロード上限 (6 MB) を超えない限り、バッチ内のすべてのレコードを単一の呼び出しで関数に渡します。
- バッチウィンドウ - 関数を呼び出す前にレコードを収集する最大時間 (秒数) を指定します。
- 開始位置 - 新規レコードのみ、または既存のすべてのレコードを処理します。
  - 最新 - ストリームに追加された新しいレコードを処理します。

- 水平トリム - ストリーム内のすべてのレコードを処理します。

  既存のレコードを処理した後、関数に戻り、新しいレコードの処理が続行されます。

- [障害発生時の宛先] — 処理できないレコードの標準 SQS キューまたは標準 SNS トピックです。Lambda は、古すぎる、または再試行回数の上限に達したレコードのバッチを廃棄すると、バッチに関する詳細をキューまたはトピックに送信します。
- 再試行回数 - 関数がエラーを返したときに Lambda が再試行する回数の上限です。これは、バッチが関数に到達しなかったサービスエラーやスロットルには適用されません。
- レコードの最大有効期間 — Lambda が関数に送信するレコードの最大経過時間。
- エラー発生時のバッチ分割 — 関数がエラーを返した場合、再試行する前にバッチを 2 つに分割します。元のバッチサイズ設定は変更されません。
- シャードごとの同時バッチ — 同じシャードからの複数のバッチを同時に処理します。
- 有効 - イベントソースマッピングを有効にするには、true に設定します。レコードの処理を停止するには、false に設定します。Lambda は、処理された最新のレコードを追跡し、マッピングが再度有効になるとその時点から処理を再開します。

> **Note**
>
> DynamoDB トリガーの一部として Lambda によって呼び出される GetRecords API コールに対しては、料金は発生しません。

後でイベントソース設定を管理するには、デザイナーでトリガーを選択します。

エラー処理

DynamoDB イベントソースマッピングのエラー処理は、エラーが関数の呼び出し前に発生するか、関数の呼び出し中に発生するかによって異なります。

- 呼び出し前: スロットリングまたはその他の問題によって Lambda イベントソースマッピングが関数を呼び出すことができない場合、レコードの有効期限が切れるか、イベントソースマッピングで設定された最大有効期間 (MaximumRecordAgeInSeconds) を超えるまで再試行します。
- 呼び出し中: 関数は呼び出されたがエラーが返された場合、Lambda はレコードの有効期限が切れるか、最大有効期間 (MaximumRecordAgeInSeconds) を超えるか、設定された再試行クォータ (MaximumRetryAttempts) に達するまで再試行します。関数エラーの場

合、BisectBatchOnFunctionError を設定することもできます。これは、失敗したバッチを 2 つの小さなバッチに分割し、不良レコードを分離してタイムアウトを回避します。バッチを分割しても、再試行クォータは消費されません。

エラー処理の対策に失敗すると、Lambda はレコードを破棄し、ストリームからのバッチ処理を継続します。デフォルト設定では、不良レコードによって、影響を受けるシャードでの処理が最大 1 日間ブロックされる可能性があります。これを回避するには、関数のイベントソースマッピングを、適切な再試行回数と、ユースケースに適合する最大レコード経過時間で設定します。

失敗した呼び出しの送信先の設定

失敗したイベントソースマッピング呼び出しの記録を保持するには、関数のイベントソースマッピングに送信先を追加します。送信先に送られる各レコードは、失敗した呼び出しに関するメタデータを含む JSON ドキュメントです。任意の Amazon SNS トピックまたは Amazon SQS キューを送信先として設定できます。実行ロールには、送信先に対するアクセス許可が必要です。

- SQS 送信先の場合: sqs:SendMessage
- SNS 送信先の場合: sns:Publish

障害発生時の送信先をコンソールを使用して設定するには、以下の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [機能の概要] で、[送信先を追加] を選択します。
4. [ソース] には、[イベントソースマッピング呼び出し] を選択します。
5. [イベントソースマッピング] では、この関数用に設定されているイベントソースを選択します。
6. [条件] には [失敗時] を選択します。イベントソースマッピング呼び出しでは、これが唯一受け入れられる条件です。
7. [送信先タイプ] では、Lambda が呼び出しレコードを送信する送信先タイプを選択します。
8. [送信先] で、リソースを選択します。
9. [Save] を選択します。

AWS Command Line Interface (AWS CLI) を使用して障害発生時の送信先を設定することもできます。例えば、次の create-event-source-mapping コマンドは、SQS を障害発生時の送信先として持つイベントソースマッピングを `MyFunction` に追加します。

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

次の update-event-source-mapping コマンドは、2 回の再試行後、またはレコードが 1 時間以上経過した場合に失敗した呼び出しレコードを SNS 送信先に送信するように、イベントソースマッピングを更新します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

更新された設定は非同期に適用され、プロセスが完了するまで出力に反映されません。現在のステータスを表示するには、get-event-source-mapping コマンドを使用します。

送信先を削除するには、destination-config パラメータの引数として空の文字列を指定します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

次の例は、DynamoDB ストリームの呼び出しレコードを示しています。

Example 呼び出しレコード

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
```

```
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

この情報は、トラブルシューティングのためにストリームから影響を受けるレコードを取得する際に使用できます。実際のレコードは含まれていないので、有効期限が切れて失われる前に、このレコードを処理し、ストリームから取得する必要があります。

Amazon CloudWatch メトリクス

関数がレコードのバッチの処理を完了すると、Lambda により `IteratorAge` メトリクスが発生します。メトリクスは、処理が終了したとき、バッチの最後のレコードがどれくらい時間が経過したレコードであったかを示します。関数が新しいイベントを処理する場合、イテレーターの有効期間を使用して、レコードが追加されてから関数によって処理されるまでのレイテンシーを推定できます。

イテレーターの有効期間が増加傾向の場合、関数に問題があることを示している可能性があります。詳しくは、「Lambda 関数のメトリクスの使用」を参照してください。

時間枠

Lambda 関数は、連続ストリーム処理アプリケーションを実行できます。ストリームは、アプリケーションを継続的に流れる無限のデータを表します。この継続的に更新される入力からの情報を分析するために、時間に関して定義されたウィンドウを使用して、含まれるレコードをバインドできます。

タンブリングウィンドウは、一定の間隔で開閉する別個のタイムウィンドウです。ディフォルトでは、Lambda 呼び出しはステートレス — 外部データベースがない場合、複数の連続した呼び出しでデータを処理するために使用することはできません。ただし、タンブリングウィンドウを使用して、

呼び出し間で状態を維持できます。この状態は、現在のウィンドウに対して以前に処理されたメッセージの集計結果が含まれます。状態は、シャードごとに最大 1 MB にすることができます。このサイズを超えると、Lambda はウィンドウを早期に終了します。

ストリームの各レコードは、特定のウィンドウに属しています。Lambda は各レコードを少なくとも 1 回処理しますが、各レコードが 1 回だけ処理される保証はありません。エラー処理などのまれなケースでは、一部のレコードが複数回処理されることがあります。レコードは常に最初から順番に処理されます。レコードが複数回処理される場合、順不同で処理されます。

集約と処理

ユーザー管理関数は、集約と、その集約の最終結果を処理するために呼び出されます。Lambda は、ウィンドウで受信したすべてのレコードを集約します。これらのレコードは、個別の呼び出しとして複数のバッチで受け取ることができます。各呼び出しは状態を受け取ります。したがって、タンブリングウィンドウを使用する場合、Lambda 関数の応答に `state` プロパティが含まれている必要があります。応答に `state` プロパティが含まれてないと、Lambda はこれを失敗した呼び出しと見なします。この条件を満たすために、関数は次の JSON 形式の `TimeWindowEventResponse` オブジェクトを返すことができます。

Example **`TimeWindowEventResponse`値**

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Java 関数の場合は、`Map<String, String>`を使用して状態を表すことをお勧めします。

ウィンドウの最後で、フラグ`isFinalInvokeForWindow`が`true`に設定され、これが最終状態であり、処理の準備ができていることが示されます。処理が完了すると、ウィンドウが完了し、最終的な呼び出しが完了し、状態は削除されます。

ウィンドウの最後に、Lambda は集計結果に対するアクションの最終処理を使用します。最終処理が同期的に呼び出されます。呼び出しが成功すると、関数はシーケンス番号をチェックポイントし、ス

トリーム処理が続行されます。呼び出しが失敗した場合、Lambda 関数は呼び出しが成功するまで処理を一時停止します。

Example DynamodbtimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
```

```
            }
         },
         "NewImage":{
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"New item!"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"222",
         "SizeBytes":59,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN":"stream-ARN"
   },
   {
      "eventID":"3",
      "eventName":"REMOVE",
      "eventVersion":"1.0",
      "eventSource":"aws:dynamodb",
      "awsRegion":"us-east-1",
      "dynamodb":{
         "Keys":{
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
```

```
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

構成

イベントソースマッピングを作成または更新するときに 、タンブリングウィンドウを設定できます。タンブリングウィンドウを設定するには、ウィンドウを秒単位で指定します (TumblingWindowInSeconds)。次の例のAWS Command Line Interface (AWS CLI)コマンドは、タンブルウィンドウが120秒に設定されたストリーミングイベントソースマッピングを作成します。集約と処理のために Lambda 関数が定義した関数の名前は `tumbling-window-example-function` です。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambdaは、レコードがストリームに挿入された時間に基づいて、タンブルするウィンドウ境界を決定します。すべてのレコードには、Lambda が境界の決定に使用するおおよそのタイムスタンプがあります。

ウィンドウの集合をタンブルしても、再共有はサポートされません。シャードが終了すると、Lambda はウィンドウが閉じているとみなし、子シャードは新しい状態で自身のウィンドウを開始します。

タンブルウィンドウは、既存の再試行ポリシーmaxRetryAttemptsおよびmaxRecordAgeを完全にサポートします。

Example Handler.py - 集約と処理

次の Python 関数は、最終状態を集約して処理する方法を示しています。

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

バッチアイテムの失敗をレポートする

イベントソースからストリーミングデータを使用および処理する場合、デフォルトでは、バッチが完全に成功した場合にのみ、バッチの最大シーケンス番号に Lambda チェックポイントが設定されます。Lambda は、他のすべての結果を完全な失敗として扱い、再試行の上限までバッチの処理を再試行します。ストリームからのバッチの処理中に部分的な成功を許可するには、ReportBatchItemFailuresをオンにします 。部分的な成功を許可すると、レコードの再試

行回数を減らすことができますが、成功したレコードの再試行の可能性を完全に妨げるわけではありません。

ReportBatchItemFailures をオンにするには、列挙値 **ReportBatchItemFailures** を FunctionResponseTypes リストに含めます。このリストは、関数で有効になっているレスポンスタイプを示します。このリストは、イベントソースマッピングを作成または更新するときに設定できます。

レポートの構文

バッチアイテムの失敗に関するレポートを設定する場合、StreamsEventResponse クラスはバッチアイテムの失敗のリストとともに返されます。StreamsEventResponseオブジェクトを使用して、バッチ処理で最初に失敗したレコードのシーケンス番号を返すことができます。また、正しいレスポンスシンタックスを使用して、独自のカスタムクラスを作成することもできます。次の JSON 構造体は、必要な応答構文を示しています。

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> batchItemFailures 配列に複数の項目が含まれている場合、Lambda はシーケンス番号が最も小さいレコードをチェックポイントとして使用します。その後、Lambda はそのチェックポイントからすべてのレコードを再試行します。

成功条件と失敗の条件

次のいずれかを返すと、Lambda はバッチを完全な成功として処理します:

- 空のbatchItemFailureリストです。
- null の batchItemFailure リスト
- 空の EventResponse
- ヌル EventResponse

次のいずれかを返すと、Lambda はバッチを完全な失敗として処理します:

- 空の文字列`itemIdentifier`
- ヌル `itemIdentifier`
- `itemIdentifier`間違えているキー名

Lambda は、再試行戦略に基づいて失敗を再試行します。

バッチを 2 分割します

呼び出しが失敗し、`BisectBatchOnFunctionError` オンになっている場合、バッチは`ReportBatchItemFailures`設定に関係なく 2 分割されます。

部分的なバッチ成功レスポンスを受信し、`BisectBatchOnFunctionError` と `ReportBatchItemFailures` の両方がオンになっている場合、バッチは返されたシーケンス番号で 2 分割され、Lambda は残りのレコードのみを再試行します。

バッチで失敗したメッセージ ID のリストを返す関数コードの例を次に示します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
```

```
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
```

```
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

TypeScript を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { DynamoDBBatchItemFailure, DynamoDBStreamEvent } from "aws-lambda";

export const handler = async (event: DynamoDBStreamEvent):
 Promise<DynamoDBBatchItemFailure[]> => {

    const batchItemsFailures: DynamoDBBatchItemFailure[] = []
    let curRecordSequenceNumber

    for(const record of event.Records) {
        curRecordSequenceNumber = record.dynamodb?.SequenceNumber

        if(curRecordSequenceNumber) {
            batchItemsFailures.push({
                itemIdentifier: curRecordSequenceNumber
            })
        }
    }
```

```
    return batchItemsFailures
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB バッチ項目失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
```

```
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
```

```
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Amazon DynamoDB Streams 設定パラメータ

すべての Lambda イベントソースタイプは、同じCreateEventSourceMapping および UpdateEventSourceMapping API オペレーションを共有しています。ただし、DynamoDB Streams に適用されるのは一部のパラメータのみです。

DynamoDB Streams に適用されるイベントソースパラメータ

[Parameter] (パラメータ)	必須	デフォルト	メモ
BatchSize	N	100	最大: 10,000
BisectBatchOnFunctionError	N	false	

[Parameter] (パラメータ)	必須	デフォルト	メモ
DestinationConfig	N		廃棄されたレコードの標準 Amazon SQS キューまたは標準 Amazon SNS トピックの送信先。
有効	N	true	
EventSourceArn	Y		データストリームまたはストリームコンシューマーの ARN。
FilterCriteria	N		Lambda のイベントフィルタリング
FunctionName	Y		
FunctionResponseTypes	N		関数がバッチ内の特定の失敗を報告できるようにするには、`FunctionResponseTypes` に値 `ReportBatchItemFailures` を含めます。詳細については、「バッチアイテムの失敗をレポートする」を参照してください。
MaximumBatchingWindowInSeconds	N	0	

[Parameter] (パラメータ)	必須	デフォルト	メモ
MaximumRecordAgeInSeconds	N	-1	-1 は無制限を意味し、失敗したレコードは有効期限が切れるまで再試行されます。「DynamoDB ストリームのデータ保持制限」は 24 時間です。 最小: -1 最大: 604,800
MaximumRetryAttempts	N	-1	-1 に設定すると無制限になり、失敗したレコードはレコードの有効期限が切れるまで再試行されます。 最小: 0 最大: 10,000
ParallelizationFactor	N	1	最大: 10
StartingPosition	Y		TRIM_HORIZON または LATEST
TumblingWindowInSeconds	N		最小: 0 最大: 900

チュートリアル: Amazon DynamoDB Streams で AWS Lambda を使用する

このチュートリアルでは、Amazon DynamoDB ストリームからのイベントを処理する Lambda 関数を作成します。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する

AWS リソースにアクセスするためのアクセス権限を関数に付与する実行ロールを作成します。

実行ロールを作成するには

1. IAM コンソールの [ロールページ] を開きます。

2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。

   - 信頼されたエンティティ – Lambda
   - アクセス許可 - AWSLambdaDynamoDBExecutionRole
   - ロール名 - **lambda-dynamodb-role**

AWSLambdaDynamoDBExecutionRole には、DynamoDB から項目を読み取り、 に CloudWatch Logs ログを書き込むために、関数が必要とするアクセス許可があります。

関数を作成する

DynamoDB イベントを処理する Lambda 関数を作成します。関数コードは、受信イベントデータの一部を CloudWatch ログに書き込みます。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;
```

```
public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
```

```
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で DynamoDB イベントの消費。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
```

```
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
```

```
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

TypeScript を使用した Lambda での DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
```

```
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で DynamoDB イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB イベントを利用します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
```

```
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

関数を作成するには

1. サンプルコードを example.js という名前のファイルにコピーします。
2. デプロイパッケージを作成します。

```
zip function.zip example.js
```

3. create-function コマンドを使用して Lambda 関数を作成します。

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs18.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

Lambda 関数をテストする

このセットアップでは、invokeAWS Lambda の CLI コマンドと次のサンプルの DynamoDB イベントを使用して、Lambda 関数を手動で呼び出します。次の内容を input.txt という名前のファイルにコピーします。

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
```

```
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"New item!"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"222",
          "SizeBytes":59,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
       "eventSourceARN":"stream-ARN"
    },
    {
       "eventID":"3",
       "eventName":"REMOVE",
       "eventVersion":"1.0",
       "eventSource":"aws:dynamodb",
       "awsRegion":"us-east-1",
       "dynamodb":{
          "Keys":{
             "Id":{
                "N":"101"
             }
          },
          "OldImage":{
             "Message":{
                "S":"This item has changed"
             },
             "Id":{
                "N":"101"
             }
          },
          "SequenceNumber":"333",
          "SizeBytes":38,
          "StreamViewType":"NEW_AND_OLD_IMAGES"
       },
```

```
            "eventSourceARN":"stream-ARN"
        }
    ]
}
```

次の invoke コマンドを実行します。

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

この関数はレスポンス本文で文字列 `message` を返します。

`outputfile.txt` ファイルで出力を確認します。

ストリーミングが有効になった DynamoDB テーブルを作成する

ストリーミングが有効な Amazon DynamoDB テーブルを作成します。

DynamoDB テーブルを作成するには

1. DynamoDB コンソールを開きます。
2. [Create table] を選択します。
3. 次の設定でテーブルを作成します。
   - テーブル名 – **`lambda-dynamodb-stream`**
   - プライマリキー – **`id`** (文字列)
4. [作成] を選択します。

ストリームを有効化するには

1. DynamoDB コンソールを開きます。
2. [テーブル] を選択します。
3. [lambda-dynamodb-stream] テーブルを選択します。

4. [Exports and streams] (エクスポートとストリーミング)で、[DynamoDB stream details] (DynamoDB ストリーミングの詳細) を選択します。
5. [オンにする] を選択します。
6. [ビュータイプ] には、[キー属性のみ] を選択します。
7. [ストリームをオンにする] を選択します。

ストリーム ARN をメモします。こちらは、次のステップでストリームを Lambda 関数に関連付ける際に必要になります。ストリームの有効化の詳細については、「DynamoDB Streams を使用したテーブルアクティビティのキャプチャ」を参照してください。

AWS Lambda でイベントソースを追加する

AWS Lambda でイベントソースマッピングを作成します。このイベントソースのマッピングは、DynamoDB ストリームを Lambda 関数に関連付けます。このイベントソースのマッピングを作成すると、AWS Lambda はストリームのポーリングを開始します。

次の AWS CLI `create-event-source-mapping` コマンドを実行します。コマンドの実行後、UUID をメモします。この UUID は、イベントソースマッピングを削除するときなど、コマンドでイベントソースマッピングを参照する場合に必要になります。

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

これにより、指定された DynamoDB ストリームと Lambda 関数の間にマッピングが作成されます。DynamoDB ストリームを複数の Lambda 関数と関連付け、同じLambda 関数を複数のストリームと関連付けることができます。ただし、Lambda 関数は、共有するストリーム用に、読み取りスループットを共有します。

次のコマンドを実行して、イベントソースのマッピングのリストを取得できます。

```
aws lambda list-event-source-mappings
```

このリストでは、作成済みのすべてのイベントソースのマッピングが返され、各マッピングに対して `LastProcessingResult` などが示されます。問題がある場合、このフィールドは情報メッセージを提供するために使用されます。`No records processed` (AWS Lambda がポーリングを開始していないか、ストリームにレコードがないことを示す) や、`OK` (AWS Lambda がストリームから正常にレコードを読み取り、Lambda 関数を呼び出したことを示す) などの値は、問題がないことを示しています。問題がある場合は、エラーメッセージが返されます。

イベントソースマッピングが多数ある場合、関数の name パラメータを使用して結果を絞り込みます。

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

セットアップをテストする

エンドツーエンドエクスペリエンスをテストします。テーブルの更新を実行すると、DynamoDB はイベントレコードをストリームに書き込みます。ストリームをポーリングしている AWS Lambda は、ストリームで新しいレコードを検出し、イベントを Lambda 関数に渡して、ユーザーに代わって関数を実行します。

1. DynamoDB コンソールで、テーブルに項目を追加、更新、削除します。DynamoDB は、これらのアクションのレコードをストリームに書き込みます。
2. AWS Lambda は、ストリームをポーリングし、ストリームの更新を検出すると、ストリームで見つかったイベントデータを渡して Lambda 関数を呼び出します。
3. 関数が実行され、Amazon CloudWatch にログが作成されます。報告されたログは Amazon CloudWatch コンソールで確認できます。

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。

4. テキスト入力フィールドにロールの名前を入力し、[Delete] (削除) を選択します。

DynamoDB テーブルを削除するには

1. DynamoDB コンソールで [Tables (テーブル)] ページを開きます。
2. 作成したテーブルを選択します。
3. [削除] を選択します。
4. テキストボックスに「**delete**」と入力します。
5. [テーブルの削除] を選択します。

サンプル関数コード

サンプルコードは以下の言語で利用可能です。

トピック

- Node.js
- Java 11
- C#
- Python 3
- Go

Node.js

次の例では、DynamoDB からメッセージを処理し、その内容を口グに記録します。

Example ProcessDynamoDBStream.js

```
console.log('Loading function');

exports.lambda_handler = function(event, context, callback) {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(function(record) {
        console.log(record.eventID);
        console.log(record.eventName);
        console.log('DynamoDB Record: %j', record.dynamodb);
    });
```

```
    callback(null, "message");
};
```

サンプルコードを zip ファイルに圧縮し、デプロイパッケージを作成します。手順については、「.zip ファイルアーカイブで Node.js Lambda 関数をデプロイする」を参照してください。

Java 11

次の例では、DynamoDB からのメッセージを処理して、その内容を口グに記録します。`handleRequest` は、AWS Lambda が呼び出してデータを処理するハンドラーです。このハンドラーは、定義済みの `DynamodbEvent` クラスを使用します。このクラスは `aws-lambda-java-events` ライブラリで定義されています。

Example DDB.EventProcessorjava

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler2;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;

public class DDBEventProcessor implements
        RequestHandler2<DynamodbEvent, String> {

    public String handleRequest(DynamodbEvent ddbEvent, Context context) {
        for (DynamodbStreamRecord record : ddbEvent.getRecords()){
           System.out.println(record.getEventID());
           System.out.println(record.getEventName());
           System.out.println(record.getDynamodb().toString());

        }
        return "Successfully processed " + ddbEvent.getRecords().size() + " records.";
    }
}
```

ハンドラーが例外をスローせずに正常に戻った場合、Lambda はレコードの入力バッチが正しく処理されたと見なし、ストリーム内の新しいレコードの読み取りを開始します。ハンドラーによって例外がスローされる場合、Lambda はレコードの入力バッチが処理されていないと見なし、レコードの同じバッチで関数を再度呼び出します。

依存関係

- aws-lambda-java-core
- aws-lambda-java-events

Lambda ライブラリの依存関係を使ってコードを構築し、デプロイパッケージを作成します。手順については、「.zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする」を参照してください。

C#

次の例では、DynamoDB からのメッセージを処理して、その内容を口グに記録します。ProcessDynamoEvent は、AWS Lambda が呼び出してデータを処理するハンドラーです。このハンドラーは、定義済みの DynamoDbEvent クラスを使用します。このクラスは Amazon.Lambda.DynamoDBEvents ライブラリで定義されています。

Example ProcessingDynamoDBStreams.cs

```
using System;
using System.IO;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

using Amazon.Lambda.Serialization.Json;

namespace DynamoDBStreams
{
    public class DdbSample
    {
        private static readonly JsonSerializer _jsonSerializer = new JsonSerializer();

        public void ProcessDynamoEvent(DynamoDBEvent dynamoEvent)
        {
            Console.WriteLine($"Beginning to process {dynamoEvent.Records.Count}
 records...");

            foreach (var record in dynamoEvent.Records)
            {
                Console.WriteLine($"Event ID: {record.EventID}");
                Console.WriteLine($"Event Name: {record.EventName}");
```

```
                string streamRecordJson = SerializeObject(record.Dynamodb);
                Console.WriteLine($"DynamoDB Record:");
                Console.WriteLine(streamRecordJson);
            }

            Console.WriteLine("Stream processing complete.");
        }

        private string SerializeObject(object streamRecord)
        {
            using (var ms = new MemoryStream())
            {
                _jsonSerializer.Serialize(streamRecord, ms);
                return Encoding.UTF8.GetString(ms.ToArray());
            }
        }
    }
}
```

.NET Core プロジェクトの `Program.cs` を上記のサンプルに置き換えます。手順については、「.zip ファイルアーカイブを使用して C# Lambda 関数を構築し、デプロイする」を参照してください。

Python 3

次の例では、DynamoDB からメッセージを処理し、その内容を口グに記録します。

Example ProcessDynamoDBStream.py

```
from __future__ import print_function

def lambda_handler(event, context):
    for record in event['Records']:
        print(record['eventID'])
        print(record['eventName'])
    print('Successfully processed %s records.' % str(len(event['Records'])))
```

サンプルコードを zip ファイルに圧縮し、デプロイパッケージを作成します。手順については、「Python Lambda 関数で .zip ファイルアーカイブを使用する」を参照してください。

Go

次の例では、DynamoDB からメッセージを処理し、その内容を口グに記録します。

Example

```
import (
    "strings"

    "github.com/aws/aws-lambda-go/events"
)

func handleRequest(ctx context.Context, e events.DynamoDBEvent) {

    for _, record := range e.Records {
        fmt.Printf("Processing request data for event ID %s, type %s.\n",
 record.EventID, record.EventName)

        // Print new values for attributes of type String
        for name, value := range record.Change.NewImage {
            if value.DataType() == events.DataTypeString {
                fmt.Printf("Attribute name: %s, value: %s\n", name, value.String())
            }
        }
    }
}
```

go build を使って実行可能ファイルを構築し、デプロイパッケージを作成します。手順については、「.zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする」を参照してください。

DynamoDB アプリケーション用の AWS SAM テンプレート

を使用してこのアプリケーションをビルドすることができます。。AWS SAMAWS SAM テンプレートの詳細については、AWS SAM 開発者ガイドの「AWS Serverless Application Model テンプレートの基礎」を参照してください。

チュートリアルアプリケーションのサンプル AWS SAM テンプレートを以下に示します。下のテキストを .yaml ファイルにコピーし、以前作成した ZIP パッケージの隣に保存します。Handler および Runtime パラメータ値は、前のセクションで関数を作成したときのものと一致する必要があります。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
```

```
Resources:
  ProcessDynamoDBStream:
    Type: AWS::Serverless::Function
    Properties:
      Handler: handler
      Runtime: runtime
      Policies: AWSLambdaDynamoDBExecutionRole
      Events:
        Stream:
          Type: DynamoDB
          Properties:
            Stream: !GetAtt DynamoDBTable.StreamArn
            BatchSize: 100
            StartingPosition: TRIM_HORIZON

  DynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
      StreamSpecification:
        StreamViewType: NEW_IMAGE
```

パッケージおよびデプロイコマンドを使用してサーバーレスアプリケーションをパッケージングしてデプロイする方法については、AWS Serverless Application Model 開発者ガイドの「サーバーレスアプリケーションのデプロイ」を参照してください。

Lambda が Amazon Kinesis Data Streams からのレコードを処理する方法

Lambda 関数を使用して、Amazon Kinesis データストリームのレコードを処理できます。Lambda 関数を Kinesis Data Streams 共有スループットコンシューマー (標準イテレーター) にマップすることも、拡張ファンアウトを使用する専用スループットコンシューマーにマップすることもできます。標準イテレーターの場合、Lambda は HTTP プロトコルを使用して、Kinesis ストリームの各シャードにレコードがあるかどうかをポーリングします。イベントソースマッピングは、シャードの他のコンシューマーと読み取りスループットを共有します。

Kinesis Data Streams の詳細については、Reading Data from Amazon Kinesis Data Streams を参照してください。

> **Note**
>
> Kinesis は、各シャードに対して課金し、拡張ファンアウトの場合はストリームから読み取られたデータに対して課金します。料金の詳細については、Amazon Kinesis の料金を参照してください。

ポーリングストリームとバッチストリーム

Lambda はデータストリームからレコードを読み取り、関数を、ストリームのレコードを含むイベントと共に同期的に呼び出します。Lambda はバッチ単位でレコードを読み取り、関数を呼び出してバッチからレコードを処理します。各バッチには、単一のシャード/データストリームのレコードが含まれます。

デフォルトで、Lambda はレコードが使用可能になると同時に関数を呼び出します。Lambda がイベントソースから読み取るバッチにレコードが 1 つしかない場合、Lambda は関数に 1 つのレコードしか送信しません。少数のレコードで関数を呼び出さないようにするには、バッチ処理ウィンドウを設定することで、最大 5 分間レコードをバッファリングするようにイベントソースに指示できます。関数を呼び出す前に、Lambda は、完全なバッチを収集する、バッチ処理ウィンドウの期限が切れる、またはバッチが 6 MB のペイロード制限に到達するまでイベントソースからのレコードの読み取りを継続します。詳細については、「バッチ処理動作」を参照してください。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

Kinesis データストリームの 1 つのシャードを複数の Lambda 呼び出しで同時に処理するには、ParallelizationFactor 設定を構成します。Lambda がシャードからポーリングする同時バッチの数は、1 (デフォルト) ~ 10 の並列化係数で指定できます。例えば、`ParallelizationFactor` を 2 に設定すると、最大 200 個の Lambda 呼び出しを同時に実行して、100 個の Kinesis データシャードを処理できます (ただし、`ConcurrentExecutions` メトリックの実際の値は異なったものとなり

ます)。これにより、データボリュームが揮発性で IteratorAge が高いときに処理のスループットをスケールアップすることができます。シャードごとの同時実行バッチの数を増やしても、Lambda はパーティションキーレベルで順序立った処理を確実に行います。

Kinesis 集約で ParallelizationFactor を使用することもできます。イベントソースマッピングの動作は、拡張ファンアウトを使用しているかどうかによって異なります。

- 拡張ファンアウトなし: 集約イベント内のすべてのイベントは、同じパーティションキーを持つ必要があります。パーティションキーは、集約イベントのパーティションキーとも一致する必要があります。集約イベント内のイベントに異なるパーティションキーがある場合、Lambda ではパーティションキーによるイベントが順序通りに処理されないことがあります。
- 拡張ファンアウトあり: まず、Lambda は集約イベントを個々のイベントにデコードします。集約イベントには、含まれるイベントとは異なるパーティションキーを設定できます。ただし、パーティションキーに一致しないイベントは削除され、失われます。Lambda ではこれらのイベントは処理されず、設定された障害時の送信先には送信されません。

イベントの例

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
```

```
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

Lambda を使用した Amazon Kinesis Data Streams レコードの処理

Lambda を使用して Amazon Kinesis Data Streams レコードを処理するには、ストリーム用のコンシューマーを作成し、次に Lambda イベントソースマッピングを作成します。

データストリームと関数の設定

Lambda 関数は、データストリームのコンシューマーアプリケーションです。シャードごとに 1 つのレコードのバッチを一度に処理します。Lambda 関数を共有スループットコンシューマー (標準イテレーター) にマップすることも、拡張ファンアウトを使用する専用スループットコンシューマーにマップすることもできます。

- 標準イテレーター: Lambda は、レコードの Kinesis ストリームにある各シャードを 1 秒あたり 1 回の基本レートでポーリングします。利用可能なレコードが増えると、Lambda は関数がストリームに追いつくまでバッチを処理し続けます。イベントソースマッピングは、シャードの他のコンシューマーと読み取りスループットを共有します。
- 拡張ファンアウト: レイテンシーを最小限に抑え、読み取りスループットを最大化するには、拡張ファンアウトを使用してデータストリームコンシューマーを作成します。拡張ファンアウトを使用するコンシューマーは、ストリームから読み取る他のアプリケーションに影響を及

ぼさないように、専用の接続を各シャードに割り当てます。ストリームのコンシューマーは HTTP/2 を使用して、長時間にわたる接続とリクエストヘッダーの圧縮でレコードを Lambda にプッシュすることによってレイテンシーを短縮します。ストリームコンシューマーは、Kinesis RegisterStreamConsumer API を使用して作成できます。

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

次のような出力が表示されます。

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

関数がレコードを処理する速度を上げるには、データストリームにシャードを追加します。Lambda は、各シャードのレコードを順番に処理します。関数からエラーが返された場合、シャードのさらなるレコードの処理は停止されます。シャードが増えると、一度に処理されるバッチが増え、同時実行のエラーの影響を下げることができます。

同時実行のバッチの合計分を処理できるように関数をスケールアップできない場合は、関数のクォータ引き上げをリクエストするか、同時実行数を予約します。

Lambda 関数を呼び出すためのイベントソースマッピングを作成する

データストリームからのレコードを使用して Lambda 関数を呼び出すには、イベントソースマッピングを作成します。複数のイベントソースマッピングを作成することで、複数の Lambda 関数で同じデータを処理したり、1 つの関数で複数のデータストリームの項目を処理したりできます。複数のストリームから項目を処理する場合、各バッチには 1 つのシャードまたはストリームのレコードのみが含まれます。

別の AWS アカウント のストリームからのレコードを処理するようにイベント ソース マッピングを構成できます。詳細については、「the section called “クロスアカウントマッピング”」を参照してください。

イベントソースマッピングを作成する前に、Kinesis データストリームから読み取るためのアクセス許可を Lambda 関数に付与する必要があります。Lambda には、Kinesis データストリームに関連するリソースを管理するために次のアクセス許可が必要です。

- kinesis:DescribeStream
- kinesis:DescribeStreamSummary
- kinesis:GetRecords
- kinesis:GetShardIterator
- kinesis:ListShards
- kinesis:ListStreams
- kinesis:SubscribeToShard

AWS マネージドポリシー AWSLambdaKinesisExecutionRole には、これらのアクセス許可が含まれています。次の手順の説明に従って、この管理ポリシーを関数に追加します。

AWS Management Console

関数に Kinesis アクセス許可を追加するには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [構成] タブで、[アクセス許可] を選択します。
3. [実行ロール] ペインの [ロール名] で、関数の実行ロールへのリンクを選択します。このリンクを選択すると、IAM コンソールでそのロールのページが開きます。
4. [アクセス許可ポリシー] ペインで、[アクセス許可を追加] を選択し、[ポリシーをアタッチ] を続けて選択します。
5. [検索] フィールドに **AWSLambdaKinesisExecutionRole** を入力します。
6. ポリシーの名前の横にあるチェックボックスを選択し、[アクセス許可を追加] を選択します。

AWS CLI

関数に Kinesis アクセス許可を追加するには

- 次の CLI コマンドを実行して、`AWSLambdaKinesisExecutionRole` ポリシーを関数の実行ロールに追加します。

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

関数に Kinesis アクセス許可を追加するには

- 関数の定義で、次の例に示すように Policies プロパティを追加します。

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

必要なアクセス許可を設定した後、イベントソースマッピングを作成します。

AWS Management Console

Kinesis イベントソースマッピングを作成するには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [関数の概要] ペインで、[トリガーを追加] を選択します。
3. [トリガー設定] で、ソースとして [Kinesis] を選択します。
4. イベントソースマッピングを作成する Kinesis ストリームを選択し、オプションでストリームのコンシューマーを選択します。
5. (オプション) イベントソースマッピングのバッチサイズ 、開始位置、バッチウィンドウを編集します。
6. 追加 を選択します。

コンソールからイベントソースマッピングを作成する場合は、IAM ロールには kinesis:ListStreams 権限と kinesis:ListStreamConsumers 権限が必要です。

AWS CLI

Kinesis イベントソースマッピングを作成するには

- 次の CLI コマンドを実行して、Kinesis イベントソースマッピングを作成します。ユースケースに応じて、独自のバッチサイズと開始位置を選択します。

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

バッチ処理ウィンドウを指定するには、--maximum-batching-window-in-seconds オプションを追加します。このパラメータおよびその他のパラメータの使用の詳細については、「AWS CLI コマンドリファレンス」の「create-event-source-mapping」を参照してください。

AWS SAM

Kinesis イベントソースマッピングを作成するには

- 関数の定義で、次の例に示すように KinesisEvent プロパティを追加します。

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100
```

```
MyKinesisStream:
  Type: AWS::Kinesis::Stream
  Properties:
    ShardCount: 1
```

AWS SAM で Kinesis Data Streams のイベントソースマッピングを作成する方法の詳細については、「AWS Serverless Application Model デベロッパーガイド」の「Kinesis」を参照してください。

ポーリングとストリームの開始位置

イベントソースマッピングの作成時および更新時のストリームのポーリングは、最終的に一貫性があることに注意してください。

- イベントソースマッピングの作成時、ストリームからのイベントのポーリングが開始されるまでに数分かかる場合があります。
- イベントソースマッピングの更新時、ストリームからのイベントのポーリングが停止および再開されるまでに数分かかる場合があります。

つまり、LATEST をストリームの開始位置として指定すると、イベントソースマッピングの作成または更新中にイベントを見逃す可能性があります。イベントを見逃さないようにするには、ストリームの開始位置を TRIM_HORIZON または AT_TIMESTAMP として指定します。

クロスアカウントのイベントソースマッピングの作成

Amazon Kinesis Data Streams は、リソースベースのポリシーをサポートします。このため、別のアカウントの Lambda 関数を使用して AWS アカウント のストリームに取り込まれたデータを処理できます。

別の AWS アカウント の Kinesis ストリームを使用して Lambda 関数のイベントソースマッピングを作成するには、リソースベースのポリシーを使用してストリームを設定し、Lambda 関数に項目を読み取るアクセス許可を付与する必要があります。クロスアカウントアクセスを許可するようにストリームを設定する方法については、「Amazon Kinesis Streams Developer guide」の「Sharing access with cross-account AWS Lambda functions」を参照してください。

Lambda 関数に必要なアクセス許可を付与するリソースベースのポリシーでストリームを設定したら、前のセクションで説明した方法のいずれかを使用してイベントソースマッピングを作成します。

Lambda コンソールでイベントソースマッピングを作成する場合は、ストリームの ARN を入力フィールドに直接貼り付けます。ストリームにコンシューマーを指定する場合、コンシューマーの ARN を貼り付けると、ストリームフィールドが自動的に入力されます。

Kinesis Data Streams と Lambda を使用した部分的なバッチレスポンスの設定

イベントソースからストリーミングデータを使用および処理する場合、デフォルトでは、バッチが完全に成功した場合にのみ、バッチの最大シーケンス番号に Lambda チェックポイントが設定されます。Lambda は、他のすべての結果を完全な失敗として扱い、再試行の上限までバッチの処理を再試行します。ストリームからのバッチの処理中に部分的な成功を許可するには、`ReportBatchItemFailures`をオンにします 。部分的な成功を許可すると、レコードの再試行回数を減らすことができますが、成功したレコードの再試行の可能性を完全に妨げるわけではありません。

`ReportBatchItemFailures` をオンにするには、列挙値 **`ReportBatchItemFailures`** を FunctionResponseTypes リストに含めます。このリストは、関数で有効になっているレスポンスタイプを示します。このリストは、イベントソースマッピングを作成または更新するときに設定できます。

レポートの構文

バッチアイテムの失敗に関するレポートを設定する場合、`StreamsEventResponse` クラスはバッチアイテムの失敗のリストとともに返されます。`StreamsEventResponse`オブジェクトを使用して、バッチ処理で最初に失敗したレコードのシーケンス番号を返すことができます。また、正しいレスポンスシンタックスを使用して、独自のカスタムクラスを作成することもできます。次の JSON 構造体は、必要な応答構文を示しています。

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> batchItemFailures 配列に複数の項目が含まれている場合、Lambda はシーケンス番号が最も小さいレコードをチェックポイントとして使用します。その後、Lambda はそのチェックポイントからすべてのレコードを再試行します。

成功条件と失敗の条件

次のいずれかを返すと、Lambda はバッチを完全な成功として処理します:

- 空のbatchItemFailureリストです。
- null の batchItemFailure リスト
- 空の EventResponse
- ヌル EventResponse

次のいずれかを返すと、Lambda はバッチを完全な失敗として処理します:

- 空の文字列itemIdentifier
- ヌル itemIdentifier
- itemIdentifier間違えているキー名

Lambda は、再試行戦略に基づいて失敗を再試行します。

バッチを 2 分割します

呼び出しが失敗し、BisectBatchOnFunctionError オンになっている場合、バッチはReportBatchItemFailures設定に関係なく 2 分割されます。

部分的なバッチ成功レスポンスを受信し、BisectBatchOnFunctionError と ReportBatchItemFailures の両方がオンになっている場合、バッチは返されたシーケンス番号で 2 分割され、Lambda は残りのレコードのみを再試行します。

バッチで失敗したメッセージ ID のリストを返す関数コードの例を次に示します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
```

```
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
```

```
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }
```

```
  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
```

```
public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Javascript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};
```

```
async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }
```

```
    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で Kinesis バッチアイテム失敗のレポートをします。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
```

```
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();
```

```
    run(service_fn(function_handler)).await
}
```

Lambda で Kinesis Data Streams イベントソースの破棄されたバッチレコードを保持する

Kinesis イベントソースマッピングのエラー処理は、エラーが関数の呼び出し前に発生するか、関数の呼び出し中に発生するかによって異なります。

- 呼び出し前: スロットリングまたはその他の問題によって Lambda イベントソースマッピングが関数を呼び出すことができない場合、レコードの有効期限が切れるか、イベントソースマッピングで設定された最大有効期間 (MaximumRecordAgeInSeconds) を超えるまで再試行します。
- 呼び出し中: 関数は呼び出されたがエラーが返された場合、Lambda はレコードの有効期限が切れるか、最大有効期間 (MaximumRecordAgeInSeconds) を超えるか、設定された再試行クォータ (MaximumRetryAttempts) に達するまで再試行します。関数エラーの場合、BisectBatchOnFunctionError を設定することもできます。これは、失敗したバッチを 2 つの小さなバッチに分割し、不良レコードを分離してタイムアウトを回避します。バッチを分割しても、再試行クォータは消費されません。

エラー処理の対策に失敗すると、Lambda はレコードを破棄し、ストリームからのバッチ処理を継続します。デフォルト設定では、不良レコードによって、影響を受けるシャードでの処理が最大 1 週間ブロックされる可能性があります。これを回避するには、関数のイベントソースマッピングを、適切な再試行回数と、ユースケースに適合する最大レコード経過時間で設定します。

失敗した呼び出しの送信先の設定

失敗したイベントソースマッピング呼び出しの記録を保持するには、関数のイベントソースマッピングに送信先を追加します。送信先に送られる各レコードは、失敗した呼び出しに関するメタデータを含む JSON ドキュメントです。任意の Amazon SNS トピックまたは Amazon SQS キューを送信先として設定できます。実行ロールには、送信先に対するアクセス許可が必要です。

- SQS 送信先の場合: sqs:SendMessage
- SNS 送信先の場合: sns:Publish

障害発生時の送信先をコンソールを使用して設定するには、以下の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数を選択します。
3. [機能の概要] で、[送信先を追加] を選択します。
4. [ソース] には、[イベントソースマッピング呼び出し] を選択します。
5. [イベントソースマッピング] では、この関数用に設定されているイベントソースを選択します。
6. [条件] には [失敗時] を選択します。イベントソースマッピング呼び出しでは、これが唯一受け入れられる条件です。
7. [送信先タイプ] では、Lambda が呼び出しレコードを送信する送信先タイプを選択します。
8. [送信先] で、リソースを選択します。
9. [Save] を選択します。

AWS Command Line Interface (AWS CLI) を使用して障害発生時の送信先を設定することもできます。例えば、次の create-event-source-mapping コマンドは、SQS を障害発生時の送信先として持つイベントソースマッピングを `MyFunction` に追加します。

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

次の update-event-source-mapping コマンドは、2 回の再試行後、またはレコードが 1 時間以上経過した場合に失敗した呼び出しレコードを SNS 送信先に送信するように、イベントソースマッピングを更新します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

更新された設定は非同期に適用され、プロセスが完了するまで出力に反映されません。現在のステータスを表示するには、get-event-source-mapping コマンドを使用します。

送信先を削除するには、`destination-config` パラメータの引数として空の文字列を指定します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

以下の例は、Kinesis イベントソース呼び出しが失敗した場合に Lambda が SQS キューまたは SNS トピックに送信する内容を示しています。Lambda はこれらの送信先タイプにメタデータのみを送信するため、元のレコード全体を取得するには、streamArn、shardId、startSequenceNumber、endSequenceNumber の各フィールドを使用します。

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

この情報は、トラブルシューティングのためにストリームから影響を受けるレコードを取得する際に使用できます。実際のレコードは含まれていないので、有効期限が切れて失われる前に、このレコードを処理し、ストリームから取得する必要があります。

Lambda でのステートフル Kinesis Data Streams 処理の実装

Lambda 関数は、連続ストリーム処理アプリケーションを実行できます。ストリームは、アプリケーションを継続的に流れる無限のデータを表します。この継続的に更新される入力からの情報を分析するために、時間に関して定義されたウィンドウを使用して、含まれるレコードをバインドできます。

タンブリングウィンドウは、一定の間隔で開閉する別個のタイムウィンドウです。ディフォルトでは、Lambda 呼び出しはステートレス — 外部データベースがない場合、複数の連続した呼び出しでデータを処理するために使用することはできません。ただし、タンブリングウィンドウを使用して、呼び出し間で状態を維持できます。この状態は、現在のウィンドウに対して以前に処理されたメッセージの集計結果が含まれます。状態は、シャードごとに最大 1 MB にすることができます。このサイズを超えると、Lambda はウィンドウを早期に終了します。

ストリームの各レコードは、特定のウィンドウに属しています。Lambda は各レコードを少なくとも 1 回処理しますが、各レコードが 1 回だけ処理される保証はありません。エラー処理などのまれなケースでは、一部のレコードが複数回処理されることがあります。レコードは常に最初から順番に処理されます。レコードが複数回処理される場合、順不同で処理されます。

集約と処理

ユーザー管理関数は、集約と、その集約の最終結果を処理するために呼び出されます。Lambda は、ウィンドウで受信したすべてのレコードを集約します。これらのレコードは、個別の呼び出しとして複数のバッチで受け取ることができます。各呼び出しは状態を受け取ります。したがって、タンブリングウィンドウを使用する場合、Lambda 関数の応答に `state` プロパティが含まれている必要があります。応答に `state` プロパティが含まれてないと、Lambda はこれを失敗した呼び出しと見なします。この条件を満たすために、関数は次の JSON 形式の `TimeWindowEventResponse` オブジェクトを返すことができます。

Example **`TimeWindowEventResponse`値**

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Java 関数の場合は、Map<String, String>を使用して状態を表すことをお勧めします。

ウィンドウの最後で、フラグisFinalInvokeForWindowがtrueに設定され、これが最終状態であり、処理の準備ができていることが示されます。処理が完了すると、ウィンドウが完了し、最終的な呼び出しが完了し、状態は削除されます。

ウィンドウの最後に、Lambda は集計結果に対するアクションの最終処理を使用します。最終処理が同期的に呼び出されます。呼び出しが成功すると、関数はシーケンス番号をチェックポイントし、ストリーム処理が続行されます。呼び出しが失敗した場合、Lambda 関数は呼び出しが成功するまで処理を一時停止します。

Example kinesisTimeWindowEvent

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
```

```
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

構成

イベントソースマッピングを作成または更新するときに 、タンブリングウィンドウを設定できます。タンブリングウィンドウを設定するには、ウィンドウを秒単位で指定します(TumblingWindowInSeconds)。次の例のAWS Command Line Interface (AWS CLI)コマンドは、タンブルウィンドウが120秒に設定されたストリーミングイベントソースマッピングを作成します。集約と処理のために Lambda 関数が定義した関数の名前は `tumbling-window-example-function` です。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambdaは、レコードがストリームに挿入された時間に基づいて、タンブルするウィンドウ境界を決定します。すべてのレコードには、Lambda が境界の決定に使用するおおよそのタイムスタンプがあります。

ウィンドウの集合をタンブルしても、再共有はサポートされません。シャードが終了すると、Lambda はウィンドウが閉じられると見なし、子シャードは新しい状態で独自のウィンドウを開始します。現在のウィンドウに新しいレコードが追加されていない場合、Lambda は最大で 2 分間待機してから、ウィンドウが終了したと見なします。これにより、レコードが断続的に追加された場合でも、関数は現在のウィンドウ内のすべてのレコードを読み取ることができます。

タンブルウィンドウは、既存の再試行ポリシー`maxRetryAttempts`および`maxRecordAge`を完全にサポートします。

Example Handler.py - 集約と処理

次の Python 関数は、最終状態を集約して処理する方法を示しています。

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Amazon Kinesis Data Streams イベントソースマッピングの Lambda パラメータ

すべての Lambda イベントソースマッピングで、同じ CreateEventSourceMapping および UpdateEventSourceMapping API オペレーションが共有されます。ただし、Kinesis に適用されるのは一部のパラメータのみです。

Kinesis に適用されるイベントソースパラメータ

[Parameter] (パラメータ)	必須	デフォルト	メモ
BatchSize	N	100	最大: 10,000

[Parameter] (パラメータ)	必須	デフォルト	メモ
BisectBatchOnFunctionError	N	false	
DestinationConfig	N		破棄されたレコードの Amazon SQS キューまたは Amazon SNS トピックの送信先。詳細については、「失敗した呼び出しの送信先の設定」を参照してください。
[Enabled] (有効)	N	true	
EventSourceArn	Y		データストリームまたはストリームコンシューマーの ARN。
FunctionName	Y		
FunctionResponseTypes	N		関数がバッチ内の特定の失敗を報告できるようにするには、`FunctionResponseTypes` に値 `ReportBatchItemFailures` を含めます。詳細については、「Kinesis Data Streams と Lambda を使用した部分的なバッチレスポンスの設定」を参照してください。

[Parameter] (パラメータ)	必須	デフォルト	メモ
MaximumBatchingWindowInSeconds	N	0	
MaximumRecordAgeInSeconds	N	-1	-1 は無制限を意味します: Lambda はレコードを破棄しません (Kinesis Data Streams データ保持設定は引き続き適用されます) 最小: -1 最大: 604,800
MaximumRetryAttempts	N	-1	-1 に設定すると無制限になり、失敗したレコードはレコードの有効期限が切れるまで再試行されます。 最小: -1 最大: 10,000
ParallelizationFactor	N	1	最大: 10
StartingPosition	Y		AT_TIMESTAMP、TRIM_HORIZON、または LATEST

[Parameter] (パラメータ)	必須	デフォルト	メモ
StartingPositionTimestamp	N		StartingPosition が AT_TIMESTAMP に設定されている場合にのみ有効です。Unix タイム秒単位で読み取りをスタートする時間
TumblingWindowInSeconds	N		最小: 0 最大: 900

チュートリアル: Lambda を Kinesis Data Streams で使用する

このチュートリアルでは、Amazon Kinesis データストリームのイベントを処理する Lambda 関数を作成します。

1. カスタムアプリケーションがストリームにレコードを書き込みます。
2. AWS Lambda はストリームをポーリングし、ストリームで新しいレコードを検出すると Lambda 関数を呼び出します。
3. AWS Lambda は、Lambda 関数の作成時に指定した実行ロールを引き受けることにより、Lambda 関数を実行します。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する

AWS リソースにアクセスするためのアクセス権限を関数に付与する実行ロールを作成します。

実行ロールを作成するには

1. IAM コンソールの [ロールページ] を開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。
   - 信頼されたエンティティ - AWS Lambda
   - アクセス許可 - AWSLambdaKinesisExecutionRole。
   - Role name – **`lambda-kinesis-role`**。

AWSLambdaKinesisExecutionRole ポリシーには、Kinesis から項目を読み取り、CloudWatch Logs にログを書き込むために関数が必要とするアクセス許可があります。

関数を作成する

Kinesis メッセージを処理する Lambda 関数を作成します。この関数コードは、Kinesis レコードのイベント ID とイベントデータを CloudWatch Logs にログ記録します。

このチュートリアルでは Node.js 18.x ランタイムを使用しますが、他のランタイム言語のサンプルコードも提供しています。次のボックスでタブを選択すると、関心のあるランタイムのコードが表示されます。このステップで使用する JavaScript コードは、[JavaScript] タブに表示されている最初のサンプルにあります。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
```

```
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
```

```
        return null;
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
```

```
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
```

```
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
```

```
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
```

```
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

関数を作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. サンプル JavaScript コードを `index.js` という名前の新しいファイルにコピーします。
3. デプロイパッケージを作成します。

```
zip function.zip index.js
```

4. `create-function` コマンドを使用して Lambda 関数を作成します。

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

Lambda 関数をテストする

invokeAWS Lambda CLI コマンドおよびサンプルの Kinesis イベントを使用して、手動で Lambda 関数を呼び出します。

Lambda 関数をテストするには

1. 以下の JSON をファイルにコピーし、input.txt という名前で保存します。

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. invoke コマンドを使用して、関数にイベントを送信します。

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-

out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

レスポンスは out.txt に保存されます。

Kinesis Stream を作成する

create-stream コマンドを使用して、スキーマを作成します。

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

次の describe-stream コマンドを実行して、ストリーム ARN を取得します。

```
aws kinesis describe-stream --stream-name lambda-stream
```

次のような出力が表示されます。

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
```

```
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

次のステップで Lambda 関数にストリームを関連付けるために、ストリーム ARN を使用します。

AWS Lambda でイベントソースを追加する

次の AWS CLI `add-event-source` コマンドを実行します。

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

後で使用するために、マッピング ID をメモしておきます。`list-event-source-mappings` コマンドを実行して、イベントソースマッピングのリストを取得できます。

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

レスポンスでは、ステータス値が `enabled` であることを確認できます。イベントソースマッピングを無効にすると、レコードを失うことなくポーリングを一時停止できます。

セットアップをテストする

イベントソースマッピングをテストするには、イベントレコードを Kinesis ストリームに追加します。`--data` 値は、文字列を Kinesis に送信する前に CLI で base64 にエンコードされる文字列です。同じコマンドを複数回実行して、複数のレコードをストリームに追加することができます。

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda は実行ロールを使用して、ストリームからレコードを読み取ります。次に、Lambda 関数を呼び出し、レコードのバッチを渡します。この関数は、各レコードからデータをデコードしてログ記録し、出力を CloudWatch Logs に送信します。CloudWatch コンソールでログを表示する

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[Delete] (削除) を選択します。

Kinesis ストリームを削除するには

1. AWS Management Console にサインインし、Kinesis コンソール (https://console.aws.amazon.com/kinesis) を開きます。
2. 作成したストリームを選択します。
3. [Actions] で、[Delete] を選択します。
4. テキスト入力フィールドに **delete** を入力します。
5. [削除] を選択します。

Amazon MQ で Lambda を使用する

> **Note**
>
> Lambda 関数以外のターゲットにデータを送信したい、または送信する前にデータをエンリッチしたいという場合は、「Amazon EventBridge Pipes」を参照してください。

Amazon MQ は、Apache ActiveMQ および RabbitMQ 用のマネージドメッセージブローカーサービスです。メッセージブローカーを使用すると、ソフトウェアアプリケーションおよびコンポーネントは、さまざまなプログラミング言語、オペレーティングシステム、および、トピックまたはキューイベント送信先を介した正式なメッセージングプロトコルを使って、通信できるようになります。

また Amazon MQ は、ActiveMQ か RabbitMQ ブローカーをインストールすることにより、もしくは、異なるネットワークトポロジやその他のインフラストラクチャのニーズを提供することにより、ユーザーに代わって Amazon Elastic Compute Cloud (Amazon EC2)インスタンスを管理することもできます。

Lambda 関数を使用することで、Amazon MQ メッセージブローカーからのレコードを処理できます。Lambda は、ブローカーからメッセージを読み取り関数を同期的に呼び出す Lambda リソース、イベントソースマッピングによって関数を呼び出します。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

Amazon MQ イベントソースマッピングには、次の設定制限があります。

- 同時実行数 — Amazon MQ イベントソースマッピングを使用する Lambda 関数には、デフォルトの最大同時実行数設定があります。ActiveMQ の場合、Lambda サービスは同時実行環境の数を 5 つに制限します。RabbitMQ の場合、同時実行環境の数は 1 つに制限されます。関数の予約またはプロビジョニングされる同時実行数設定を変更しても、Lambda サービスはこれ以上実行環境を利用できるようにしません。デフォルトの最大同時実行数の増加をリクエストするには、AWS Support にお問い合わせください。
- クロスアカウント - Lambda はクロスアカウント処理をサポートしていません。Lambda を使用して、別の AWS アカウント にある Amazon MQ メッセージブローカーからのレコードを処理することはできません。
- 認証 - ActiveMQ では、ActiveMQ SimpleAuthenticationPlugin のみサポートされています。RabbitMQ の場合、PLAIN 認証メカニズムのみサポートされています。ユーザーは、資格情報の管理には AWS Secrets Manager を使用します。ActiveMQ 認証の詳細については、Amazon MQ デベロッパーガイドの Integrating ActiveMQ brokers with LDAP を参照してください。

- 接続クォータ - ブローカーは、ワイヤレベルプロトコルごとに最大の接続可能数を持っています。このクォータは、ブローカーインスタンスタイプに基づいています。これらの制限の詳細については、Amazon MQ デベロッパーガイドの Quotas in Amazon MQ の Brokers のセクションを参照してください。
- 接続 - ブローカーをパブリックまたはプライベートの Virtual Private Cloud (VPC) に作成できます。プライベート VPC の場合、Lambda 関数が VPC にアクセスしてメッセージを受信する必要があります。詳細については、このトピックで後述する「the section called “ネットワーク構成”」を参照してください。
- イベント送信先 - キューの送信先のみがサポートされます。ただし、仮想トピックを使用することができます。仮想トピックは、内部的にトピックとして動作し、キューとして Lambda と対話しながら動作します。詳細については、Apache ActiveMQ ウェブサイトの Virtual Destinations および RabbitMQ ウェブサイトの Virtual Hostsを参照してください。
- ネットワークトポロジ - ActiveMQ の場合、イベントソースマッピングごとに、 1 つの単一インスタンスまたはスタンバイブローカーがサポートされます。RabbitMQ の場合、イベントソースマッピングごとに、単一インスタンスブローカーまたはクラスターデプロイメントがサポートされます。単一インスタンスブローカーには、フェイルオーバーエンドポイントが必要です。これらのブローカーデプロイメントモードの詳細については、Amazon MQ デベロッパーガイドの Active MQ Broker Architecture および Rabbit MQ Broker Architecture を参照してください。
- プロトコル — サポートされるプロトコルは、Amazon MQ の統合のタイプによって異なります。
  - ActiveMQ 統合の場合、Lambda は OpenWire/Java Message Service (JMS) プロトコルを使用してメッセージを使用します。その他のプロトコルは、メッセージの使用をサポートしていません。JMS プロトコル内では、`TextMessage` および `BytesMessage` のみがサポートされています。Lambda は、JMS カスタムプロパティもサポートしています。OpenWire プロトコルの詳細については、Apache ActiveMQ ウェブサイトの OpenWire を参照してください。
  - RabbitMQ 統合の場合、Lambda は AMQP 0-9-1 プロトコルを使ってメッセージを使用します。その他のプロトコルは、メッセージの使用をサポートしていません。RabbitMQ による AMQP 0-9-1 プロトコルの実装の詳細については、RabbitMQ ウェブサイトの AMQP 0-9-1 Complete Reference Guide を参照してください。

Lambda は、Amazon MQ がサポートする ActiveMQ および RabbitMQ の最新バージョンを自動的にサポートします。サポートされている最新バージョンについては、Amazon MQ デベロッパーガイドの Amazon MQ リリースノートを参照してください。

> **Note**
>
> デフォルトでは、Amazon MQ には毎週、ブローカー用のメンテナンスウィンドウがあります。その期間中、ブローカーは利用できません。スタンバイのないブローカーの場合、Lambda はそのウィンドウ中にメッセージを処理できません。

セクション

Lambda コンシューマーグループ

Amazon MQ と対話するため、Lambda は、Amazon MQ ブローカーから読み取ることができるコンシューマーグループを作成します。コンシューマーグループは、イベントソースマッピング UUID と同じ ID で作成されます。

Amazon MQ イベントソースの場合、Lambda はレコードをまとめてバッチ処理し、それらを単一のペイロードで関数に送信します。動作を制御するには、バッチ処理ウィンドウとバッチサイズを設定できます。Lambda は、最大 6 MB のペイロードサイズを処理する、バッチ処理ウィンドウの期限が切れる、またはレコード数が完全なバッチサイズに到達するまで、メッセージをプルします。詳細については、「バッチ処理動作」を参照してください。

コンシューマーグループは、メッセージをバイトの BLOB として取得し、それらを base64 でエンコードして単一の JSON ペイロードに変換してから、関数を呼び出します。関数がバッチ内のいずれかのメッセージに対してエラーを返すと、Lambda は、処理が成功するかメッセージが期限切れになるまでメッセージのバッチ全体を再試行します。

> **Note**
>
> Lambda 関数の最大タイムアウト制限は通常 15 分ですが、Amazon MSK、自己管理型 Apache Kafka、Amazon DocumentDB、および ActiveMQ と RabbitMQ 向け Amazon MQ の

イベントソースマッピングでは、最大タイムアウト制限が 14 分の関数のみがサポートされます。この制約により、イベントソースマッピングは関数エラーと再試行を適切に処理できます。

Amazon CloudWatch の ConcurrentExecutions メトリクスを使用して、特定の関数の同時実行使用率を監視できます。同時実行の詳細については、「the section called “予約済同時実行数の設定”」を参照してください。

Example Amazon MQ レコードイベント

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
west-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
```

```
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
west-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
```

```
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> RabbitMQ の例では、`pizzaQueue` は RabbitMQ キューの名前、`/` は仮想ホストの名前です。メッセージを受信すると、イベントソースは `pizzaQueue::/` の下にメッセージを一覧表示します。

実行ロールのアクセス許可

Amazon MQ ブローカーからレコードを読み取るには、Lambda 関数は次のアクセス許可を実行ロールに追加する必要があります。

- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

> **Note**
>
> 暗号化されたカスタマー管理キーを使用する場合は、`kms:Decrypt` 権限も追加します。

ネットワーク構成

イベントソースマッピングを通じて Lambda にブローカーへのフルアクセスを許可するには、ブローカーがパブリックエンドポイント (パブリック IP アドレス) を使用するか、ブローカーを作成した Amazon VPC へのアクセスを提供する必要があります。

デフォルトでは、Amazon MQ ブローカーを作成すると、PubliclyAccessible フラグは false に設定されます。ブローカーがパブリック IP アドレスを受け取るには、PubliclyAccessible フラグを true にする必要があります。

Lambda で Amazon MQ を使用するときのベストプラクティスは、「VPC エンドポイント」を使用し、Lambda 関数にブローカーの VPC へのアクセスを付与することです。Lambda にエンドポイントをデプロイし、ActiveMQ のみの場合は AWS Security Token Service (AWS STS) のエンドポイントをデプロイします。ブローカーが認証を使用する場合、AWS Secrets Manager にもエンドポイントをデプロイします。詳細については、「the section called “VPCエンドポイントの使用”」を参照してください。

または、Amazon MQ ブローカーを含む VPC の各パブリックサブネットに NAT ゲートウェイを設定します。詳細については、「the section called “VPC 関数のインターネットアクセス”」を参照してください。

Amazon MQ ブローカーのイベントソースマッピングを作成すると、Lambda はブローカーの VPC のサブネットおよびセキュリティグループに Elastic Network Interface (ENI) が既に存在するかどうかを確認します。Lambda が既存の ENI を検出した場合、再利用しようとします。それ以外の場合、Lambda は新しい ENI を作成し、イベントソースに接続して関数を呼び出します。

> **Note**
>
> Lambda 関数は、Lambda サービスが所有する VPC 内で常に実行されます。これらの VPC はサービスによって自動的に管理され、顧客には表示されません。関数を Amazon VPC に接続することもできます。いずれの場合、関数の VPC 設定はイベントソースマッピングに影響しません。Lambda がイベントソースに接続する方法を判定するのは、イベントソースの VPC の設定のみです。

VPC セキュリティグループのルール

次のルール (最低限) に従ってクラスターを含む Amazon VPC のセキュリティグループを設定してください。

- インバウンドルール – イベントソースに指定されたセキュリティグループの、独自のセキュリティグループ内からのブローカーポート上のすべてのトラフィックを許可します。ActiveMQ はデフォルトでポート 61617 を使用し、RabbitMQ はデフォルトでポート 5671 を使用します。
- アウトバウンドルール – すべての送信先に対して、ポート 443 上のすべてのトラフィックを許可します。ブローカーポートのすべてのトラフィックを独自のセキュリティグループ内で許可しま

す。ActiveMQ はデフォルトでポート 61617 を使用し、RabbitMQ はデフォルトでポート 5671 を使用します。

- NAT ゲートウェイの代わりに VPC エンドポイントを使用する場合は、その VPC エンドポイントに関連付けられたセキュリティグループが、イベントソースのセキュリティグループからのポート 443 上のすべてのインバウンドトラフィックを許可する必要があります。

VPCエンドポイントの使用

VPC エンドポイントを使用するとき、ENI を使用し、関数を呼び出す API コールはこれらのエンドポイントを経由してルーティングされます。Lambda サービスプリンシパルは、これらの ENI を使用するすべての関数に `lambda:InvokeFunction` を呼び出す必要があります。さらに、ActiveMQ の場合、Lambda サービスプリンシパルは ENI を使用するロールに `sts:AssumeRole` を呼び出す必要があります。

デフォルトでは、VPC エンドポイントはオープンな IAM ポリシーがあります。特定のプリンシパルのみがそのエンドポイントを使用して必要なアクションを実行するため、これらのポリシーを制限することがベストプラクティスです。イベントソースマッピングが Lambda 関数を呼び出せるようにするには、VPC エンドポイントポリシーは Lambda サービスプリンシパルが `lambda:InvokeFunction` を呼び出せるようにする必要があります。ActiveMQ の場合は `sts:AssumeRole` です。組織内で発生する API コールのみを許可するように VPC エンドポイントポリシーを制限すると、イベントソースマッピングが正しく機能しなくなります。

次の VPC エンドポイントポリシーの例では、AWS STS および Lambda エンドポイントに必要なアクセスを付与する方法を示しています。

Example VPC エンドポイントポリシー - AWS STS エンドポイント (ActiveMQ のみ)

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
```

```
    ]
}
```

Example VPC エンドポイントポリシー – Lambda エンドポイント

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Amazon MQ ブローカーが認証を使用する場合、Secrets Manager エンドポイントの VPC エンドポイントポリシーを制限することもできます。Secrets Manager API を呼び出す場合、Lambda は Lambda サービスプリンシパルではなく、関数ロールを使用します。次の例では、Secrets Manager エンドポイントポリシーを示します。

Example VPC エンドポイントポリシー - Secrets Manager エンドポイント

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

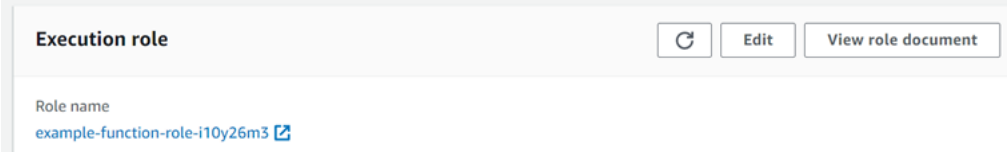
アクセス許可を追加し、イベントソースマッピングを作成するには

イベントソースマッピングを作成し、Amazon MQ ブローカーから Lambda 関数にレコードを送信するよう Lambda に通知します。複数のイベントソースマッピングを作成することで、複数の関数で同じデータを処理したり、単一の関数で複数のソースから項目を処理したりできます。

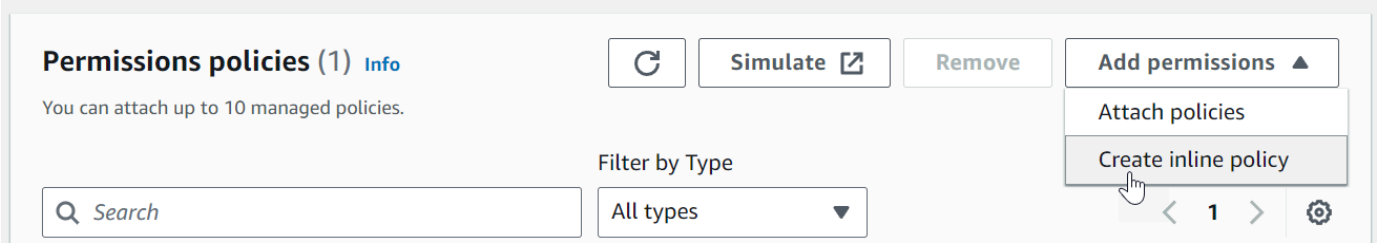
Amazon MQ から読み取るように関数を設定するには、必要なアクセス許可を追加し Lambda コンソールで MQ トリガーを作成します。

アクセス許可を追加してトリガーを作成するには

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Configuration] (設定) タブを開き、次に [Permissions] (アクセス許可) をクリックします。
4. [実行ロール] で、実行ロールのリンクを選択します。このリンクを選択すると、IAM コンソールでロールが開きます。



5. アクセス許可を追加、インラインポリシーを作成の順に選択します。



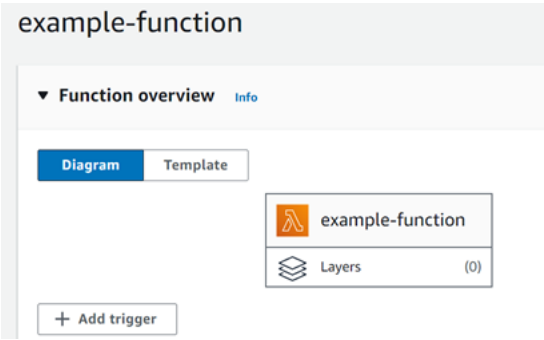
6. [ポリシーエディター] で、[JSON] を選択します。以下のポリシーを入力します。Amazon MQ ブローカーから読み取るには、関数にこれらのアクセス許可が必要です。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "mq:DescribeBroker",
        "secretsmanager:GetSecretValue",
        "ec2:CreateNetworkInterface",
```

```
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> 暗号化されたカスタマー管理キーを使用する場合は、`kms:Decrypt` アクセス許可も追加する必要があります。

7. [Next] を選択します。ポリシー名を入力し、[ポリシーの作成] を選択します。
8. Lambda コンソールの関数に戻ります。[関数の概要] で [トリガーを追加] をクリックします。



9. MQトリガータイプを選択します。
10. 必須のオプションを設定し、[追加] を選択します。

Lambda では、Amazon MQ イベントソースの以下のオプションがサポートされています。

- MQ ブローカー — Amazon MQ ブローカーを選択します。
- バッチサイズ - 単一のバッチで取得するメッセージの最大数を設定します。
- キュー名 - 使用する Amazon MQ キューを入力します。

- ソースアクセス設定 — 仮想ホスト情報とブローカーの認証情報を保存する Secrets Manager のシークレットを入力します。
- トリガーの有効化 - レコードの処理を停止するトリガーを無効にします。

トリガーを有効または無効にする（または削除する）には、 MQ トリガーをデザイナーで選択します。トリガーを再設定するには、イベントソースマッピング API 操作を使用します。

イベントソースマッピングの更新

イベントソースマッピングを更新するには、update-event-source-mapping コマンドを使用します。次のコマンド例では、イベントソースマッピングを更新して、バッチサイズを 2 にします。

```
aws lambda update-event-source-mapping \
--uuid 91eaeb7e-c976-1234-9451-8709db01f137 \
--batch-size 2
```

次のような出力が表示されます。

```
{
    "UUID": "91eaeb7e-c976-1234-9451-8709db01f137",
    "BatchSize": 2,
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-
b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-
Function",
    "LastModified": 1601928393.531,
    "LastProcessingResult": "No records processed",
    "State": "Updating",
    "StateTransitionReason": "USER_INITIATED"
}
```

Lambda は、これらの設定を非同期的に更新します。このプロセスが完了するまで、出力には変更は反映されません。get-event-source-mapping コマンドを使用して、リソースの現在のステータスを表示します。

```
aws lambda get-event-source-mapping \
--uuid 91eaeb7e-c976-4939-9451-8709db01f137
```

次のような出力が表示されます。

```
{
    "UUID": "91eaeb7e-c976-4939-9451-8709db01f137",
    "BatchSize": 2,
    "EventSourceArn": "arn:aws:mq:us-east-1:123456789012:broker:ExampleMQBroker:b-
b4d492ef-bdc3-45e3-a781-cd1a3102ecca",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MQ-Example-
Function",
    "LastModified": 1601928393.531,
    "LastProcessingResult": "No records processed",
    "State": "Enabled",
    "StateTransitionReason": "USER_INITIATED"
}
```

イベントソースマッピングエラー

Lambda 関数で回復不可能なエラーが発生すると、Amazon MQ コンシューマーはレコードの処理を停止します。他のコンシューマーは、同じエラーが発生しない限り、処理を続行できます。コンシューマーが停止した原因を特定するには、`StateTransitionReason` から返された詳細の `EventSourceMapping` フィールドをチェックし、以下のいずれかのコードを探します。

`ESM_CONFIG_NOT_VALID`

イベントソースマッピングの設定が無効です。

`EVENT_SOURCE_AUTHN_ERROR`

Lambda がイベントソースの認証に失敗しました。

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda にイベントソースへのアクセス許可がありません。

`FUNCTION_CONFIG_NOT_VALID`

関数の設定が無効です。

レコードは、サイズが原因で Lambda が削除した場合も、未処理になります。Lambda レコードのサイズ上限は 6 MB です。関数エラー時にメッセージを再配信するには、デッドレターキュー（DLQ）を使用します。詳細については、Apache ActiveMQ ウェブサイトの Message Redelivery and DLQ Handling および RabbitMQ ウェブサイトの Reliability Guide を参照してください。

> **Note**
>
> Lambda はカスタム再配信ポリシーをサポートしていません。代わりに Lambda は、Apache ActiveMQ ウェブサイトの Redelivery Policy ページのデフォルト値のポリシーを使用し、`maximumRedeliveries` は 6 に設定します。

Amazon MQ と RabbitMQ の設定パラメータ

すべての Lambda イベントソースタイプは、同じCreateEventSourceMapping および UpdateEventSourceMapping API オペレーションを共有しています。ただし、Amazon MQ と RabbitMQ に適用されるのは一部のパラメータのみです。

Amazon MQ と RabbitMQ に適用されるイベントソースパラメータ

[Parameter] (パラメータ)	必須	デフォルト	メモ
BatchSize	N	100	最大: 10,000
有効	N	true	
FunctionName	Y		
FilterCriteria	N		Lambda のイベントフィルタリング
MaximumBatchingWindowInSeconds	N	500 ミリ秒	バッチ処理動作
キュー	N		消費する Amazon MQ ブローカーの送信先キューの名前。
SourceAccessConfigurations	N		ActiveMQ の場合、BASIC_AUTH 認証情報です。RabbitMQ の場合、BASIC_AUTH 認証情報と VIRTUAL_H

[Parameter] (パラメータ)	必須	デフォルト	メモ
			OST 情報の両方を含めることができます。

Amazon MSK で Lambda を使用する

> **Note**
>
> Lambda 関数以外のターゲットにデータを送信したい、または送信する前にデータをエンリッチしたいという場合は、「Amazon EventBridge Pipes」を参照してください。

Amazon Managed Streaming for Apache Kafka (Amazon MSK) は、Apache Kafka を使ってストリーミングデータを処理するアプリケーションを、構築および実行することを可能にするフルマネージドサービスです。Amazon MSK は、Kafka を実行するクラスターのセットアップ、スケーリング、管理を簡素化します。また、Amazon MSK を使用すると、AWS Identity and Access Management (IAM) を使って複数のアベイラビリティーゾーンやセキュリティ向けにより簡単にアプリケーションを設定することができます。Amazon MSK は、Kafka の複数のオープンソースバージョンをサポートします。

Amazon MSK は、イベントソースとして Amazon Simple Queue Service (Amazon SQS) または Amazon Kinesis を使用する場合と同様に動作します。Lambda は、イベントソースからの新しいメッセージを内部的にポーリングした後、ターゲットの Lambda 関数を同期的に呼び出します。Lambda はメッセージをバッチで読み込み、それらをイベントペイロードとして関数に提供します。最大バッチサイズは設定可能です (デフォルトでは 100 メッセージ)。詳細については、「バッチ処理動作」を参照してください。

> **Note**
>
> Lambda 関数の最大タイムアウト制限は通常 15 分ですが、Amazon MSK、自己管理型 Apache Kafka、Amazon DocumentDB、および ActiveMQ と RabbitMQ 向け Amazon MQ のイベントソースマッピングでは、最大タイムアウト制限が 14 分の関数のみがサポートされます。この制約により、イベントソースマッピングは関数エラーと再試行を適切に処理できます。

Lambda は、パーティションごとにメッセージを順番に読み込みます。1 つの Lambda ペイロードに、複数のパーティションからのメッセージを含めることができます。Lambda は各バッチを処理した後、そのバッチ内のメッセージのオフセットをコミットします。関数がバッチ内のいずれかのメッセージに対してエラーを返すと、Lambda は、処理が成功するかメッセージが期限切れになるまでメッセージのバッチ全体を再試行します。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

Amazon MSK をイベントソースとして設定する方法の例については、AWS Compute Blog の Using Amazon MSK as an event source for AWS Lambdaを参照してください。完全なチュートリアルについては、「Amazon MSK Labs」(Amazon MSK ラボ) の「Amazon MSK Lambda Integration」(Amazon MSK の Lambda 統合) を参照してください。

トピック

- チュートリアル: Amazon MSK イベントソースマッピングを使用して Lambda 関数を呼び出す
- イベントの例
- MSK クラスター認証
- API アクセスと許可の管理
- 認証と認可のエラー
- ネットワーク構成
- Amazon MSK をイベントソースとして追加
- クロスアカウントのイベントソースマッピングの作成
- 障害発生時の送信先
- Amazon MSK イベントソースの Auto Scaling
- ポーリングとストリームの開始位置
- Amazon CloudWatch メトリクス
- Amazon MSK 設定パラメータ

チュートリアル: Amazon MSK イベントソースマッピングを使用して Lambda 関数を呼び出す

本チュートリアルでは、次の手順を実行します。

- 既存の Amazon MSK クラスターと同じ AWS アカウントに Lambda 関数を作成します。
- Amazon MSK と通信するように Lambda のネットワークと認証を設定します。
- Lambda Amazon MSK イベントソースマッピングを設定します。これにより、イベントがトピックに出現したときに Lambda 関数が実行されます。

これらのステップを完了したら、イベントが Amazon MSK に送信されたときに、独自のカスタム Lambda コードを使用してそれらのイベントを自動的に処理するための Lambda 関数を設定できるようになります。

この機能で何ができますか?

ソリューションの例: MSK イベントソースマッピングを使用して、ライブスコアを顧客に配信します。

次のシナリオを考えてみましょう。あなたの会社は、顧客がスポーツの試合などのライブイベントに関する情報を表示できるウェブアプリケーションをホストしています。試合の情報更新は、Amazon MSK の Kafka トピックを通じてチームに提供されます。MSK トピックから取得した更新情報を使用して、開発中のアプリケーション内で顧客にライブイベントの更新ビューを提供するソリューションを設計する必要があります。次の設計アプローチを決定しました。クライアントアプリケーションは、AWS でホストされているサーバーレスバックエンドと通信します。クライアントは、Amazon API Gateway WebSocket API を使用して WebSocket セッション経由で接続します。

このソリューションでは、MSK イベントを読み取り、いくつかのカスタムロジックを実行してアプリケーションレイヤーのイベントを準備し、その情報を API Gateway API に転送するコンポーネントが必要です。このコンポーネントは、Lambda 関数でカスタムロジックを指定し、それを AWS Lambda Amazon MSK イベントソースマッピングで呼び出すことで、AWS Lambda を使用して実装できます。

Amazon API Gateway WebSocket API を使用したソリューションの実装の詳細については、API Gateway ドキュメントの WebSocket API のチュートリアルを参照してください。

前提条件

以下の事前設定されたリソースを持つ AWS アカウント。

これらの前提条件を満たすには、Amazon MSK ドキュメントの「Amazon MSK の使用を開始する」を参照してください。

- Amazon MSK クラスター 「Amazon MSK の使用を開始する」の「Amazon MSK クラスターを作成する」を参照してください。
- 以下の設定を行います。
  - クラスターのセキュリティ設定で [IAM ロールベースの認証] が [有効] になっていることを確認します。これにより、必要な Amazon MSK リソースにのみアクセスするように Lambda 関数を制限することで、セキュリティが向上します。これは、新しい Amazon MSK クラスターではデフォルトで有効になっています。
  - クラスターネットワーク設定で [パブリックアクセス] がオフになっていることを確認します。Amazon MSK クラスターのインターネットへのアクセスを制限すると、データを処理する仲介者の数が制限されることでセキュリティが向上します。これは、新しい Amazon MSK クラスターではデフォルトで有効になっています。
- このソリューションに使用する Amazon MSK クラスターの Kafka トピック。「Amazon MSK の使用を開始する」の「トピックの作成」を参照してください。
- Kafka クラスターから情報を取得し、テスト用に Kafka イベントをトピックに送信するように設定された Kafka 管理ホスト。例えば、Kafka 管理 CLI と Amazon MSK IAM ライブラリがインストールされた Amazon EC2 インスタンスなどです。「Amazon MSK の使用を開始する」の「クライアントマシンを作成する」を参照してください。

これらのリソースを設定したら、AWS アカウントから次の情報を収集して、続行する準備ができていることを確認します。

- Amazon MSK クラスターの名前。この情報は、Amazon MSK コンソールで確認できます。
- クラスター UUID。Amazon MSK クラスターの ARN の一部であり、Amazon MSK コンソールで確認できます。この情報を確認するには、Amazon MSK ドキュメントの「クラスターの一覧表示」の手順に従います。
- Amazon MSK クラスターに関連付けられているセキュリティグループ。この情報は、Amazon MSK コンソールで確認できます。次のステップでは、これらを *clusterSecurityGroups* と呼びます。
- Amazon MSK クラスターを含む Amazon VPC の ID。この情報を見つけるには、Amazon MSK コンソールで Amazon MSK クラスターに関連付けられたサブネットを特定し、Amazon VPC コンソールでそのサブネットに関連付けられた Amazon VPC を特定します。

- ソリューションで使用される Kafka トピックの名前。この情報を確認するには、Kafka 管理ホストから Kafka `topics` CLI を使用して Amazon MSK クラスターを呼び出します。トピック CLI の詳細については、Kafka ドキュメントの「Adding and removing topics」を参照してください。
- Kafka トピックのコンシューマーグループの名前。Lambda 関数での使用に適しています。このグループは Lambda によって自動的に作成できるため、Kafka CLI で作成する必要はありません。コンシューマーグループを管理する必要がある場合、コンシューマーグループ CLI の詳細については、Kafka ドキュメントの「Managing Consumer Groups」を参照してください。

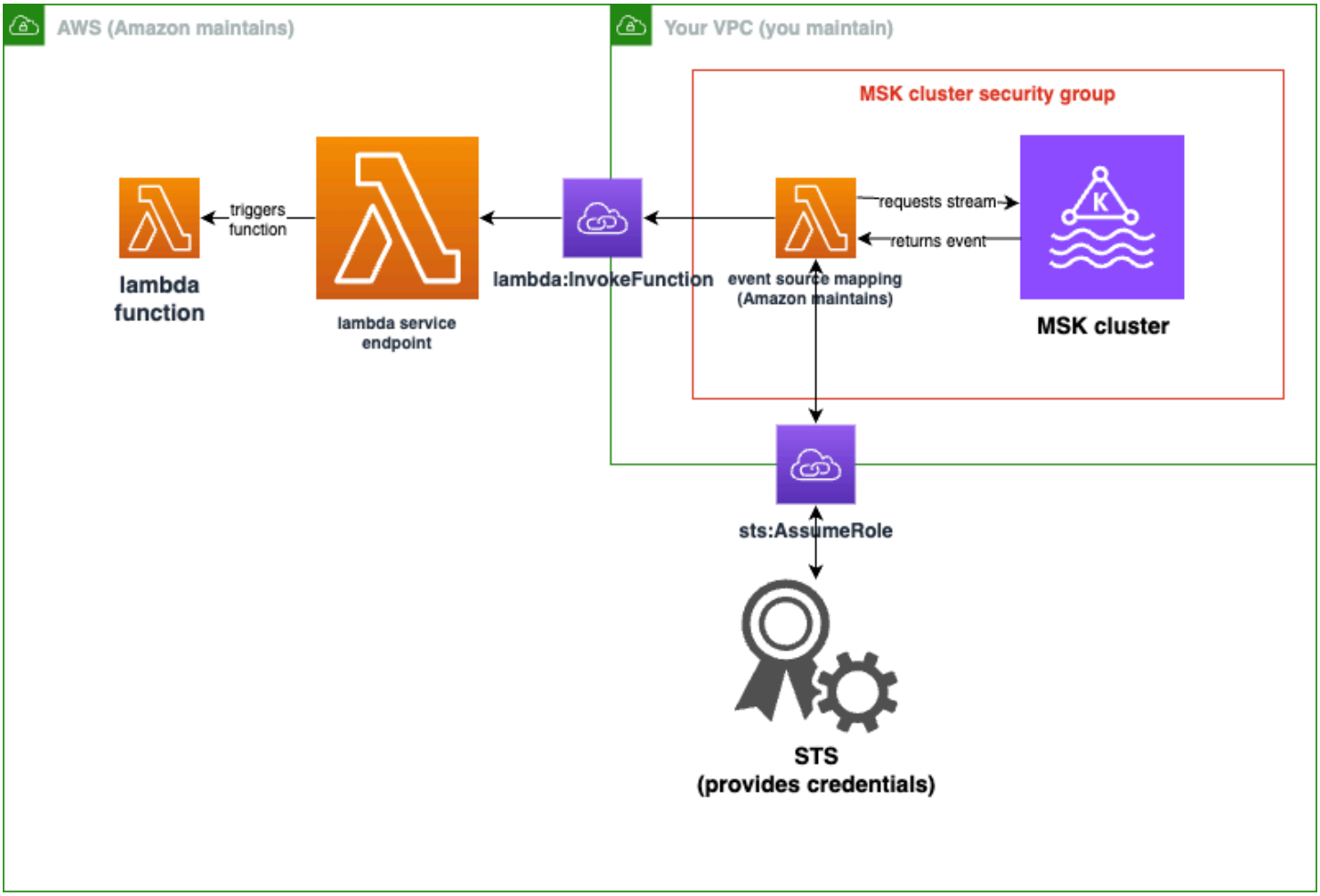
AWS アカウントに次のアクセス許可が必要です。

- Lambda 関数を作成および管理するためのアクセス許可。
- IAM ポリシーを作成し、それを Lambda 関数に関連付けるためのアクセス許可。
- Amazon MSK クラスターをホストする Amazon VPC で Amazon VPC エンドポイントを作成し、ネットワーク設定を変更するためのアクセス許可。

Amazon MSK と通信するように Lambda のネットワーク接続を設定する

AWS PrivateLink を使用して Lambda と Amazon MSK を接続します。これを行うには、Amazon VPC コンソールでインターフェイス Amazon VPC エンドポイントを作成します。ネットワーク設定の詳細については、「the section called “ネットワーク構成”」を参照してください。

Amazon MSK イベントソースマッピングが Lambda 関数に代わって実行される際には、Lambda 関数の実行ロールを引き受けます。この IAM ロールは、マッピングに対して、IAM で保護されたリソース (Amazon MSK クラスターなど) へのアクセスを許可します。コンポーネントは実行ロールを共有しますが、次の図に示すように、Amazon MSK マッピングと Lambda 関数には、それぞれのタスクに対して個別の接続要件があります。



イベントソースマッピングは、Amazon MSK クラスターセキュリティグループに属します。このネットワークステップでは、Amazon MSK クラスター VPC から Amazon VPC エンドポイントを作成して、イベントソースマッピングを Lambda および STS サービスに接続します。Amazon MSK クラスターセキュリティグループからのトラフィックを受け入れるように、これらのエンドポイントを保護します。次に、Amazon MSK クラスターのセキュリティグループを調整して、イベントソースマッピングが Amazon MSK クラスターと通信できるようにします。

AWS Management Consoleを使用して、次の手順を設定できます。

Lambda と Amazon MSK を接続するようにインターフェイス Amazon VPC エンドポイントを設定するには

1. インターフェイス Amazon VPC エンドポイントのセキュリティグループ *endpointSecurityGroup* を作成します。これにより、*clusterSecurityGroups* からのポート 443 でのインバウンド TCP トラフィックが許可されます。Amazon EC2 ドキュメントの「セキュリティグループの作成」の手順に従って、セキュリティグループを作成します。次

に、Amazon EC2 ドキュメントの「セキュリティグループへのルールの追加」の手順に従って、適切なルールを追加します。

次の情報を使用してセキュリティグループを作成します。

インバウンドルールを追加する際に、*clusterSecurityGroups* 内のセキュリティグループごとにルールを作成します。各ルールは、次のように作成します。

- [タイプ] で [HTTPS] を選択します。
- [ソース] で *clusterSecurityGroups* のいずれかを選択します。

2. Lambda サービスを Amazon MSK クラスターを含む Amazon VPC に接続するエンドポイントを作成します。「インターフェイスエンドポイントの作成」の手順に従います。

   次の情報を使用してインターフェイスエンドポイントを作成します。

   - [サービス名] で com.amazonaws.*regionName*.lambda を選択します。ここで、*regionName* が Lambda 関数をホストします。
   - [VPC] で、Amazon MSK クラスターを含む Amazon VPC を選択します。
   - [セキュリティグループ] で、前に作成した *endpointSecurityGroup* を選択します。
   - [サブネット] で、Amazon MSK クラスターをホストするサブネットを選択します。
   - [ポリシー] で、次のポリシードキュメントを指定します。これにより、Lambda サービスプリンシパルが lambda:InvokeFunction アクションに使用するためにエンドポイントを保護します。

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

   - [DNS 名を有効化] が設定されたままであることを確認します。

3. AWS STS サービスを Amazon MSK クラスターを含む Amazon VPC に接続するエンドポイントを作成します。「インターフェイスエンドポイントの作成」の手順に従います。

   次の情報を使用してインターフェイスエンドポイントを作成します。

   - [サービス名] で AWS STS を選択します。
   - [VPC] で、Amazon MSK クラスターを含む Amazon VPC を選択します。
   - [セキュリティグループ] で *endpointSecurityGroup* を選択します。
   - [サブネット] で、Amazon MSK クラスターをホストするサブネットを選択します。
   - [ポリシー] で、次のポリシードキュメントを指定します。これにより、Lambda サービスプリンシパルが `sts:AssumeRole` アクションに使用するためにエンドポイントを保護します。

   ```
   {
       "Statement": [
           {
               "Action": "sts:AssumeRole",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - [DNS 名を有効化] が設定されたままであることを確認します。

4. Amazon MSK クラスターに関連付けられている各セキュリティグループに対して、つまり *clusterSecurityGroups* で、次のことを許可します。

   - すべての *clusterSecurityGroups* に対して、ポート 9098 でのすべてのインバウンドおよびアウトバウンド TCP トラフィックを許可します。これには、clusterSecurityGroups 内でのトラフィックも含まれます。
   - ポート 443 でのすべてのアウトバウンド TCP トラフィックを許可します。

   このトラフィックの一部は、デフォルトのセキュリティグループルールで許可されているため、クラスターが単一のセキュリティグループにアタッチされており、そのグループにデフォルトのルールがある場合は、追加のルールは必要ありません。セキュリティグループルールを調整する

には、Amazon EC2 ドキュメントの「セキュリティグループへのルールの追加」の手順に従います。

次の情報を使用して、セキュリティグループにルールを追加します。

- ポート 9098 のインバウンドルールまたはアウトバウンドルールごとに、以下を指定します。
  - [タイプ] で、[カスタム TCP] を選択します。
  - [ポート範囲] で 9098 を指定します。
  - [ソース] で *clusterSecurityGroups* のいずれかを指定します。
- ポート 443 のインバウンドルールごとに、[タイプ] で [HTTPS] を選択します。

Lambda が Amazon MSK トピックから読み取るための IAM ロールを作成する

Lambda が Amazon MSK トピックから読み取るための認証要件を特定し、ポリシーで定義します。ロール *lambdaAuthRole* を作成します。このロールは、Lambda がこれらのアクセス許可を使用することを承認します。kafka-cluster IAM アクションを使用して Amazon MSK クラスターに対するアクションを承認します。次に、Lambda が Amazon MSK クラスターを検出して接続するために必要な Amazon MSK の kafka アクションと Amazon EC2 アクションを実行すること、および Lambda が実行した内容をログに記録できるように CloudWatch アクションを実行することを承認します。

Lambda が Amazon MSK から読み取るための認証要件を記述するには

1. IAM ポリシードキュメント (JSON ドキュメント) である *clusterAuthPolicy* を作成します。これにより、Lambda は Kafka コンシューマーグループを使用して、Amazon MSK クラスター内の Kafka トピックからデータを読み取ることができるようになります。Lambda では、読み取り時に Kafka コンシューマーグループを設定する必要があります。

   前提条件に合わせて次のテンプレートを変更します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
```

```
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/mskClusterName/cluster-
uuid",
                "arn:aws:kafka:region:account-id:topic/mskClusterName/cluster-
uuid/mskTopicName",
                "arn:aws:kafka:region:account-id:group/mskClusterName/cluster-
uuid/mskGroupName"
            ]
        }
    ]
}
```

詳細については、「the section called “IAM ロールベースの認証”」を参照してください。ポリシーは、次のように作成します。

- *region* および *account-id* には、Amazon MSK クラスターをホストするものを指定します。
- *mskClusterName* には、Amazon MSK クラスターの名前を指定します。
- *cluster-uuid* には、Amazon MSK クラスターの ARN 内の UUID を指定します。
- *mskTopicName* には、Kafka トピックの名前を指定します。
- *mskGroupName* には、Kafka コンシューマーグループの名前を指定します。

2. Lambda が Amazon MSK クラスターを検出して接続し、それらのイベントを口グに記録するために必要となる、Amazon MSK、Amazon EC2、および CloudWatch のアクセス許可を特定します。

   AWSLambdaMSKExecutionRole マネージドポリシーは、必要なアクセス許可を許容的に定義します。これは、次の手順で使用します。

   本番環境で AWSLambdaMSKExecutionRole を評価し、最小特権の原則に基づいて実行ロールポリシーを制限します。その後、ロールに対して、このマネージドポリシーを置き換えるポリシーを作成します。

IAM ポリシー言語の詳細については、IAM ドキュメントを参照してください。

ポリシードキュメントを作成したので、IAM ポリシーを作成してロールにアタッチすることができます。これを行うには、コンソールを使用して次の手順を実行します。

ポリシードキュメントから IAM ポリシーを作成するには

1. AWS Management Console にサインインして、IAM コンソール (https://console.aws.amazon.com/iam/) を開きます。
2. 左側のナビゲーションペインで、[ポリシー] を選択します。
3. [Create policy] を選択します。
4. [ポリシーエディタ] セクションで、[JSON] オプションを選択します。
5. *clusterAuthPolicy* を貼り付けます。
6. ポリシーにアクセス権限を追加し終えたら、[次へ] を選択します。
7. [確認と作成] ページで、作成するポリシーの [ポリシー名] と [説明] (オプション) を入力します。[このポリシーで定義されているアクセス許可] を確認して、ポリシーによって付与されたアクセス許可を確認します。
8. [ポリシーの作成] をクリックして、新しいポリシーを保存します。

詳細については、IAM ドキュメントの「IAM ポリシーの作成」を参照してください。

適切な IAM ポリシーを作成したので、ロールを作成してそれらのポリシーをアタッチします。これを行うには、コンソールを使用して次の手順を実行します。

IAM コンソールで実行ロールを作成するには

1. IAM コンソールの [Roles (ロール)] ページを開きます。
2. [ロールの作成] を選択します。
3. [信頼されたエンティティタイプ] から、[AWS サービス] を選択します。
4. [ユースケース] で、Lambda を選択します。
5. [Next] を選択します。
6. 次のポリシーを指定します。
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. [Next] を選択します。
8. [ロール名] に *lambdaAuthRole* と入力し、[ロールの作成] を選択します。

詳細については、「the section called “実行ロール (関数に対する、他のリソースにアクセスするためのアクセス許可)”」を参照してください。

Amazon MSK トピックから読み取る Lambda 関数を作成する

IAM ロールを使用するように設定された Lambda 関数を作成します。コンソールを使用して Lambda 関数を作成できます。

認証設定を使用して Lambda 関数を作成するには

1. Lambda コンソールを開き、ヘッダーから [関数の作成] を選択します。
2. [ゼロから作る] を選択します。
3. [関数名] で、任意の適切な名前を指定します。
4. [ランタイム] で、[最新のサポート対象] バージョンの `Node.js` を選択すると、このチュートリアルで提供されるコードを使用できます。
5. [デフォルトの実行ロールの変更] を選択します。
6. [既存のロールを使用] を選択します。
7. [既存のロール] で、*`lambdaAuthRole`* を選択します。

本番環境では、通常、Lambda 関数が Amazon MSK のイベントを適切に処理できるようにするために、実行ロールにさらにポリシーを追加する必要があります。ロールにポリシーを追加する方法の詳細については、IAM ドキュメントの「ID アクセス許可の追加または削除」を参照してください。

Lambda 関数へのイベントソースマッピングを作成する

Amazon MSK イベントソースマッピングは、該当する Amazon MSK イベントが発生したときに Lambda を呼び出すために必要な情報を Lambda サービスに提供します。コンソールを使用して Amazon MSK マッピングを作成できます。Lambda トリガーを作成すると、イベントソースマッピングは自動的に設定されます。

Lambda トリガー (およびイベントソースマッピング) を作成するには

1. Lambda 関数の概要ページに移動します。
2. 関数の概要セクションで、左下の [トリガーの追加] を選択します。
3. [ソースの選択] ドロップダウンで、[Amazon MSK] を選択します。
4. [認証] を設定しないでください。
5. [MSK クラスター] で、クラスターの名前を選択します。

6. [バッチサイズ] で、1 を入力します。これは、この機能のテストを容易にするためのステップであり、本番環境で理想的な値ではありません。
7. [トピック名] で、Kafka トピックの名前を指定します。
8. [コンシューマーグループ ID] で、Kafka コンシューマーグループの ID を指定します。

ストリーミングデータを読み取るために Lambda 関数を更新する

Lambda は、イベントメソッドパラメータを使用して Kafka イベントに関する情報を提供します。Amazon MSK イベントの構造の例については、「the section called “ イベントの例”」を参照してください。Lambda によって転送された Amazon MSK イベントを解釈する方法を理解したら、これらのイベントによって提供される情報を使用するように Lambda 関数コードを変更できます。

テスト目的で Lambda Amazon MSK イベントの内容をログに記録するには、Lambda 関数に次のコードを指定します。

Node.js

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

コンソールを使用して Lambda に関数コードを指定できます。

Lambda 関数コードを更新するには

1. Lambda 関数の概要ページに移動します。
2. [コード] タブを選択します。
3. 指定されたコードを [コードソース] IDE に入力します。
4. [コードソース] ナビゲーションバーで、[デプロイ] を選択します。

Lambda 関数をテストして、Amazon MSK トピックに接続されていることを確認します。

CloudWatch イベントログを調べることで、Lambda がイベントソースによって呼び出されているかどうかを確認できるようになりました。

Lambda 関数が呼び出されているかどうかを確認するには

1. Kafka 管理ホストを使用し、`kafka-console-producer` CLI を使用して Kafka イベントを生成します。詳細については、Kafka ドキュメントの「Write some events into the topic」を参照してください。前のステップで定義したイベントソースマッピングのバッチサイズで定義されたバッチを埋めるのに十分なイベントを送信してください。そうしないと、Lambda は追加の情報が呼び出されるまで待機します。
2. 関数が実行されると、Lambda はその結果を CloudWatch に書き込みます。コンソールで、Lambda 関数の詳細ページに移動します。
3. [Configuration (設定)] タブを選択します。
4. サイドバーから、[モニタリングおよび運用ツール] を選択します。
5. [ロギング設定] で [CloudWatch ロググループ] を特定します。ロググループは `/aws/lambda` で始まります。ロググループへのリンクを選択します。
6. CloudWatch コンソールの [ログイベント] で、Lambda がログストリームに送信したログイベントがないかを調べます。次の図のように、Kafka イベントからのメッセージを含むログイベントがあるかどうかを確認します。存在する場合は、Lambda イベントソースマッピングを使用して Lambda 関数を Amazon MSK に正常に接続できています。

```
2020-08-06T15:06:18.861-04:00	START RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a Version: $LATEST
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Key: mytopic-0
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z	88ebae59-be0c-4e22-9db7-4154b437e43a	INFO	Record: {
  topic: 'mytopic',
  partition: 0,
  offset: 38,
  timestamp: 1596740777633,
  timestampType: 'CREATE_TIME',
  value: 'TWVzc2FnZSAjMQ=='
}
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Message: Message #1
2020-08-06T15:06:18.890-04:00	END RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a
```

イベントの例

Lambda は、関数を呼び出すとき、イベントパラメータ内のメッセージのバッチを送信します。イベントペイロードにはメッセージの配列が含まれています。各配列項目には、Amazon MSK トピックとパーティション識別子の詳細が、タイムスタンプおよび base64 でエンコードされたメッセージとともに含まれています。

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:sa-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

MSK クラスター認証

Lambda には、Amazon MSK クラスターにアクセスする、レコードを取得する、およびその他タスクを実行するための許可が必要です。Amazon MSK は、MSK クラスターへのクライアントアクセスを制御するためのいくつかのオプションをサポートしています。

クラスターアクセスオプション

- 非認証アクセス
- SASL/SCRAM 認証
- IAM ロールベースの認証
- 相互 TLS 認証
- mTLS シークレットの設定
- Lambda でのブートストラップブローカーの選択方法

非認証アクセス

インターネット経由でクラスターにアクセスするクライアントがない場合は、非認証アクセスを使用できます。

SASL/SCRAM 認証

Amazon MSK は、Transport Layer Security (TLS) 暗号化を使用した Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 認証をサポートしています。Lambda がクラスターに接続できるようにするには、認証情報 (ユーザー名とパスワード) を AWS Secrets Manager シークレットに保存します。

Secrets Manager の使用に関する詳細については、「Amazon Managed Streaming for Apache Kafka デベロッパーガイド」の「AWS Secrets Manager を使用したユーザーネームとパスワードの認証」を参照してください。

Amazon MSK は SASL/PLAIN 認証をサポートしません。

IAM ロールベースの認証

IAM を使用して、MSK クラスターに接続するクライアントのアイデンティを認証することができます。MSK クラスターで IAM 認証がアクティブ化されており、認証用のシークレットを指定しない場合、Lambda はデフォルトで自動的に IAM 認証を使用します。ユーザーまたはロールベースのポリ

シーを作成してデプロイするには、IAM コンソール、または API を使用します。詳細については、「Amazon Managed Streaming for Apache Kafka Developer Guide」(Amazon Managed Streaming for Apache Kafka デベロッパーガイド) の「IAM access control」(IAM アクセスコントロール) を参照してください。

Lambda が MSK クラスターに接続し、レコードを読み取り、その他の必要なアクションを実行できるようにするには、関数の実行ロールに以下の許可を追加します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:region:account-id:cluster/cluster-name/cluster-uuid",
                "arn:aws:kafka:region:account-id:topic/cluster-name/cluster-uuid/topic-name",
                "arn:aws:kafka:region:account-id:group/cluster-name/cluster-uuid/consumer-group-id"
            ]
        }
    ]
}
```

これらの許可は、特定のクラスター、トピック、およびグループにスコープできます。詳細については、「Amazon Managed Streaming for Apache Kafka Developer Guide」(Amazon Managed Streaming for Apache Kafka デベロッパーガイド) の「Amazon MSK Kafka actions」(Amazon MSK Kafka アクション) を参照してください。

相互 TLS 認証

相互 TLS (mTLS) は、クライアントとサーバー間の双方向認証を提供します。クライアントは、サーバーによるクライアントの検証のためにサーバーに証明書を送信し、サーバーは、クライアントによるサーバーの検証のためにクライアントに証明書を送信します。

Amazon MSK の場合、Lambda がクライアントとして機能します。MSK クラスターのブローカーで Lambda を認証するように、クライアント証明書を (Secrets Manager のシークレットとして) 設定します。クライアント証明書は、サーバーのトラストストア内の CA によって署名される必要があります。MSK クラスターは、Lambda でブローカーを認証するために Lambda にサーバー証明書を送信します。サーバー証明書は、AWS トラストストア内の認証局 (CA) によって署名される必要があります。

クライアント証明書を生成する方法の手順については、「Introducing mutual TLS authentication for Amazon MSK as an event source」(イベントソースとしての Amazon MSK のための相互 TLS 認証の紹介) を参照してください。

Amazon MSK は自己署名のサーバー証明書をサポートしません。これは、Amazon MSK のすべてのブローカーが、Lambda がデフォルトで信頼する Amazon Trust Services CA によって署名されたパブリック証明書を使用するためです。

Amazon MSK のための mTLS に関する詳細については、「Amazon Managed Streaming for Apache Kafka Developer Guide」(Amazon Managed Streaming for Apache Kafka デベロッパーガイド) の「Mutual TLS Authentication」(相互 TLS 認証) を参照してください。

mTLS シークレットの設定

CLIENT_CERTICATE_TLS_AUTH シークレットは、証明書フィールドとプライベートキーフィールドを必要とします。暗号化されたプライベートキーの場合、シークレットはプライベートキーのパスワードを必要とします。証明書とプライベートキーは、どちらも PEM 形式である必要があります。

> **Note**
>
> Lambda は、PBES1 (PBES2 ではありません) プライベートキー暗号化アルゴリズムをサポートします。

証明書フィールドには、クライアント証明書で始まり、その後に中間証明書が続き、ルート証明書で終わる証明書のリストが含まれている必要があります。各証明書は、以下の構造を使用した新しい行で始める必要があります。

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager は最大 65,536 バイトのシークレットをサポートします。これは、長い証明書チェーンにも十分な領域です。

プライベートキーは、以下の構造を使用した PKCS #8 形式にする必要があります。

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

暗号化されたプライベートキーには、以下の構造を使用します。

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

以下は、暗号化されたプライベートキーを使用する mTLS 認証のシークレットの内容を示す例です。暗号化されたプライベートキーの場合は、シークレットにプライベートキーのパスワードを含めます。

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Lambda でのブートストラップブローカーの選択方法

Lambda は、クラスターで使用可能な認証方法、および認証用のシークレットが提供されているかどうかに基づき、ブートストラップブローカーを選択します。mTLS または SASL/SCRAM のシークレットを指定すると、Lambda は自動的にその認証方法を選択します。シークレットを指定しない場合、Lambda は、クラスターでアクティブ化されている中で、最も強力な認証方法を選択します。以下は、Lambda によるブローカー選択の優先度を、最も強力な認証から弱い認証の順に示したものです。

- mTLS (mTLS 用のシークレットを提供)
- SASL/SCRAM (SASL/SCRAM 用のシークレットを提供)
- SASL IAM (シークレットが提供されておらず、IAM 認証がアクティブ)
- 非認証の TLS (シークレットが提供されておらず、IAM 認証も非アクティブ)
- プレーンテキスト (シークレットが提供されておらず、IAM 認証と非認証 TLS の両方が非アクティブ)

> **Note**
>
> Lambda から最も安全なブローカータイプへの接続ができない場合でも、Lambda は別の (安全性の低い) ブローカータイプへの接続を試行しません。安全性の低いブローカータイプを Lambda に選択させたい場合は、クラスターが使用している、より強力な認証方法をすべて無効にします。

API アクセスと許可の管理

Amazon MSK クラスターへのアクセスに加えて、関数にはさまざまな Amazon MSK API アクションを実行するための許可が必要です。これらの許可は、関数の実行ロールに追加します。ユーザーが Amazon MSK API アクションのいずれかにアクセスする必要がある場合は、ユーザーまたはロールのアイデンティティポリシーに必要な許可を追加します。

次の各許可を実行ロールに手動で追加できます。または、AWS マネージドポリシー AWSLambdaMSKExecutionRole を実行ロールにアタッチすることもできます。`AWSLambdaMSKExecutionRole` ポリシーには、以下にリストされているすべての必要な API アクションと VPC 許可が含まれています。

Lambda 関数の実行ロールに必要な許可

Amazon CloudWatch Logs のロググループでログを作成して保存するには、Lambda 関数の実行ロールに以下の許可が必要です。

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Lambda がユーザーに代わって Amazon MSK クラスターにアクセスするには、Lambda 関数の実行ロールに次の許可が必要です。

- kafka:DescribeCluster
- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers
- kafka:DescribeVpcConnection: クロスアカウントのイベントソースマッピングにのみ必要です。
- kafka:ListVpcConnections: 実行ロールでは必要ありませんが、クロスアカウントのイベントソースマッピングを作成する IAM プリンシパルには必要です。

必要なのは、`kafka:DescribeCluster` または `kafka:DescribeClusterV2` のいずれかを追加することだけです。プロビジョンド MSK クラスターの場合、どちらの許可も機能します。サーバーレス MSK クラスターの場合は、`kafka:DescribeClusterV2` を使用する必要があります。

> **Note**
>
> Lambda は関連付けられている `AWSLambdaMSKExecutionRole` マネージドポリシーから `kafka:DescribeCluster` の許可を最終的に削除する予定です。このポリシーを使用する場合、`kafka:DescribeCluster` を使用しているすべてのアプリケーションは、代わりに `kafka:DescribeClusterV2` を使用するように移行する必要があります。

VPC アクセス許可

Amazon MSK クラスターにアクセスできるのが VPC 内のユーザーのみである場合、Lambda 関数には Amazon VPC リソースにアクセスするための許可が必要です。これらのリソースには、VPC、サブネット、セキュリティグループ、ネットワークインターフェイスが含まれます。それらのリソー

スにアクセスするには、関数の実行ロールに次のアクセス許可が必要です。これらのアクセス許可は、AWS マネージドポリシー AWSLambdaMSKExecutionRole に含まれています。

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

Lambda 関数のオプションのアクセス許可

Lambda 関数には、以下を実行する許可も必要になる場合があります。

- SASL/SCRAM 認証を使用している場合は、SCRAM シークレットにアクセスします。
- Secrets Manager シークレットを記述する。
- AWS Key Management Service (AWS KMS) カスタマー管理のキーにアクセスする。
- 失敗した呼び出しのレコードを送信先に送信します。

Secrets Manager と AWS KMS 許可

Amazon MSK ブローカーに設定しているアクセスコントロールのタイプに応じて、Lambda 関数には SCRAM シークレットにアクセスするための許可 (SASL/SCRAM 認証を使用する場合)、または AWS KMS カスタマーマネージドキーを復号するための Secrets Manager シークレットが必要になる場合があります。それらのリソースにアクセスするには、関数の実行ロールに次のアクセス許可が必要です。

- kafka:ListScramSecrets
- secretsmanager:GetSecretValue
- kms:Decrypt

実行ロールへのアクセス許可の追加

IAM コンソールを使用して実行ロールに AWS マネージドポリシー AWSLambdaMSKExecutionRole を追加するには、次の手順を実行します。

AWS 管理ポリシーを追加するには

1. IAM コンソールの [Policies (ポリシー)] ページを開きます。
2. 検索ボックスに、ポリシー名 (AWSLambdaMSKExecutionRole) を入力します。
3. リストからポリシーを選択して、[ポリシーアクション] の [アタッチ] を選択します。
4. 添付ポリシーページで、リストから実行ロールを選択し、[Attach policy](ポリシーの添付) を選択します。

IAM ポリシーを使用したユーザーアクセスの許可

デフォルトでは、ユーザーとロールには Amazon MSK API 操作を実行する許可がありません。組織またはアカウント内のユーザーにアクセス権を付与するには、アイデンティティベースのポリシーを追加または更新することができます。詳細については、Amazon Managed Streaming for Apache Kafka デベロッパーガイドの Amazon MSK Identity-Based Policy Examples を参照してください。

認証と認可のエラー

Amazon MSK クラスターからのデータを消費するために必要な許可のいずれかが欠落している場合、Lambda は [LastProcessingResult] のイベントソースマッピングに以下のエラーメッセージのいずれかを表示します。

エラーメッセージ

- クラスターが Lambda の認可に失敗した
- SASL 認証に失敗した
- Server failed to authenticate Lambda (サーバーが Lambda の認証に失敗しました)
- Provided certificate or private key is invalid (提供された証明書またはプライベートキーが無効です)

クラスターが Lambda の認可に失敗した

SALS/SCRAM または mTLS の場合、このエラーは、指定されたユーザーが以下の必要とされる Kafka アクセスコントロールリスト (ACL) 許可のすべてを持っていないことを示します。

- DescribeConfigs クラスター
- グループを記述する
- グループを読み取る
- トピックを記述する

- トピックを読み取る

IAM アクセスコントロールの場合、関数の実行ロールにグループまたはトピックへのアクセスに必要な許可が 1 つ、または複数不足しています。「the section called “IAM ロールベースの認証”」で、必要な許可のリストを確認してください。

必要な Kafka クラスター許可を使用して Kafka ACL または IAM ポリシーのいずれかを作成するときは、リソースとしてトピックとグループを指定します。トピック名は、イベントソースマッピングのトピックと一致する必要があります。グループ名は、イベントソースマッピングの UUID と一致する必要があります。

必要な許可を実行ロールに追加した後は、変更が有効になるまで数分間かかる場合があります。

SASL 認証に失敗した

SASL/SCRAM の場合、このエラーは指定されたユーザー名とパスワードが無効であることを示します。

IAM アクセスコントロールの場合、実行ロールに MSK クラスターに対する `kafka-cluster:Connect` 許可がありません。この許可をロールに追加して、クラスターの Amazon リソースネーム (ARN) をリソースとして指定します。

このエラーは断続的に発生する場合があります。クラスターは、TCP 接続の数が Amazon MSK サービスクォータを超過すると、接続を拒否します。Lambda は接続に成功するまでバックオフし、再試行します。Lambda がクラスターに接続してレコードをポーリングすると、最後の処理結果が OK に変わります。

Server failed to authenticate Lambda (サーバーが Lambda の認証に失敗しました)

このエラーは、Amazon MSK Kafka ブローカーが Lambda の認証に失敗したことを示します。このエラーは、以下が原因で発生する可能性があります。

- mTLS 認証用のクライアント証明書を提供していない。
- クライアント証明書を提供したが、ブローカーが mTLS を使用するように設定されていない。
- クライアント証明書がブローカーに信頼されていない。

Provided certificate or private key is invalid (提供された証明書またはプライベートキーが無効です)

このエラーは、Amazon MSK コンシューマーが提供された証明書またはプライベートキーを使用できなかったことを示します。証明書とキーが PEM 形式を使用しており、プライベートキーの暗号化が PBES1 アルゴリズムを使用していることを確認してください。

ネットワーク構成

Lambda が Kafka クラスターをイベントソースとして使用するには、クラスターが存在する Amazon VPC にアクセスする必要があります。VCP にアクセスするには、Lambda に AWS PrivateLink 「VPC エンドポイント」をデプロイすることをお勧めします。Lambda および AWS Security Token Service (AWS STS) にエンドポイントをデプロイします。ブローカーが認証を使用する場合は、Secrets Manager 用の VPC エンドポイントもデプロイします。失敗時の送信先を設定した場合は、送信先サービスの VPC エンドポイントもデプロイします。

または、Kafka クラスターに関連付けられた VPC に、パブリックサブネットごとに 1 つの NAT ゲートウェイが含まれていることを確認します。詳細については、「the section called “VPC 関数のインターネットアクセス”」を参照してください。

VPC エンドポイントを使用する場合は、プライベート DNS 名を有効にするように設定する必要もあります。

MSK クラスターのイベントソースマッピングを作成すると、Lambda はクラスターの VPC のサブネットおよびセキュリティグループに Elastic Network Interface (ENI) が既に存在するかどうかを確認します。Lambda が既存の ENI を検出した場合、再利用しようとします。それ以外の場合、Lambda は新しい ENI を作成し、イベントソースに接続して関数を呼び出します。

> **Note**
>
> Lambda 関数は、Lambda サービスが所有する VPC 内で常に実行されます。これらの VPC はサービスによって自動的に管理され、顧客には表示されません。関数を Amazon VPC に接続することもできます。いずれの場合、関数の VPC 設定はイベントソースマッピングに影響しません。Lambda がイベントソースに接続する方法を判定するのは、イベントソースの VPC の設定のみです。

Amazon VPC の設定は、Amazon MSK API を使用して検出できます。create-event-source-mapping コマンドを使用してセットアップ中に設定する必要はありません。

ネットワークの設定方法の詳細については、AWS Compute Blog の Setting up AWS Lambda with an Apache Kafka cluster within a VPC を参照してください。

VPC セキュリティグループのルール

次のルール (最低限) に従ってクラスターを含む Amazon VPC のセキュリティグループを設定してください。

- インバウンドルール – イベントソースに指定されたセキュリティグループに対して、Amazon MSK ブローカーポート (プレーンテキストの場合は 9092、TLS の場合は 9094、SASL の場合は 9096、IAM の場合は 9098) 上のすべてのトラフィックを許可します。
- アウトバウンドルール – すべての送信先に対して、ポート 443 上のすべてのトラフィックを許可します。イベントソースに指定されたセキュリティグループに対して、Amazon MSK ブローカーポート (プレーンテキストの場合は 9092、TLS の場合は 9094、SASL の場合は 9096、IAM の場合は 9098) 上のすべてのトラフィックを許可します。
- NAT ゲートウェイの代わりに VPC エンドポイントを使用している場合は、その VPC エンドポイントに関連付けられたセキュリティグループが、イベントソースのセキュリティグループからのポート 443 上のすべてのインバウンドトラフィックを許可する必要があります。

VPCエンドポイントの使用

VPC エンドポイントを使用するとき、ENI を使用し、関数を呼び出す API コールはこれらのエンドポイントを経由してルーティングされます。Lambda サービスプリンシパルは、これらの ENI を使用するすべてのロールおよび関数に対し、`sts:AssumeRole` および `lambda:InvokeFunction` を呼び出す必要があります。

デフォルトでは、VPC エンドポイントはオープンな IAM ポリシーがあります。特定のプリンシパルのみがそのエンドポイントを使用して必要なアクションを実行するため、これらのポリシーを制限することがベストプラクティスです。イベントソースマッピングが Lambda 関数を呼び出せるようにするには、VPC エンドポイントポリシーは Lambda サービスプリンシパルが `sts:AssumeRole` および `lambda:InvokeFunction` を呼び出せるようにする必要があります。組織内で発生する API コールのみを許可するように VPC エンドポイントポリシーを制限すると、イベントソースマッピングが正しく機能しなくなります。

次の VPC エンドポイントポリシーの例では、AWS STS および Lambda エンドポイントに Lambda サービスプリンシパルの必要なアクセスを付与する方法について示しています。

Example VPC エンドポイントポリシー - AWS STS エンドポイント

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC エンドポイントポリシー – Lambda エンドポイント

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Kafka ブローカーが認証を使用する場合、Secrets Manager エンドポイントの VPC エンドポイントポリシーを制限することもできます。Secrets Manager API を呼び出す場合、Lambda は Lambda サービスプリンシパルではなく、関数ロールを使用します。次の例では、Secrets Manager エンドポイントポリシーを示します。

Example VPC エンドポイントポリシー - Secrets Manager エンドポイント

```
{
```

```
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

障害が発生している宛先が設定されている場合、Lambda は関数のロールを使用し、Lambda が管理する ENI で `s3:PutObject`、`sns:Publish`、`sqs:sendMessage` のいずれかを呼び出します。

Amazon MSK をイベントソースとして追加

イベントソースマッピングを作成するには、Amazon MSK を、Lambda コンソール、AWSSDK、またはAWS Command Line Interface(AWS CLI) を使用して Lambda 関数のトリガーとして追加します。Amazon MSK をトリガーとして追加すると、Lambda は Lambda 関数用の VPC 設定ではなく、Amazon MSK クラスター用の VPC 設定を引き受けることに注意してください。

このセクションでは、Lambda コンソールと AWS CLI を使用してイベントソースマッピングを作成する方法について説明します。

前提条件

- Amazon MSK クラスターと Kafka トピック 詳細については、Amazon Managed Streaming for Apache Kafka デベロッパーガイドの Getting Started Using Amazon MSK を参照してください。
- MSK クラスターが使用する AWS リソースにアクセスするための許可を持つ実行ロール。

カスタマイズ可能なコンシューマーグループ ID

Kafkaをイベントソースとして設定する場合、コンシューマーグループIDを指定できます。このコンシューマーグループ ID は、Lambda 関数を結合したい Kafka コンシューマーグループの既存の識別子です。この機能を使用すると、実行中の Kafka レコード処理設定を他のコンシューマーから Lambda にシームレスに移行できます。

コンシューマーグループ ID を指定し、そのコンシューマーグループ内に他のアクティブなポーラーが存在する場合、Kafka はすべてのコンシューマーにメッセージを配信します。言い換えると、Lambda は Kafka トピックのメッセージをすべて受け取るわけではありません。Lambda にトピック内のすべてのメッセージを処理させたい場合は、そのコンシューマーグループの他のポーラーをすべてオフにします。

さらに、コンシューマーグループ ID を指定し、Kafka が同じ ID を持つ有効な既存のコンシューマーグループを見つけた場合、Lambda は、イベントソースマッピングの `StartingPosition` パラメーターを無視します。代わりに、Lambda はコンシューマーグループのコミットされたオフセットに従ってレコードの処理を開始します。コンシューマーグループ ID を指定しても、Kafka が既存のコンシューマーグループを見つけられない場合、Lambda は指定された `StartingPosition` を使用してイベントソースを設定します。

指定するコンシューマーグループ ID は、すべての Kafka イベントソースの中で一意でなければなりません。コンシューマーグループ ID を指定して Kafka イベントソースマッピングを作成した後は、この値を更新することはできません。

Amazon MSK トリガーの追加 (コンソール)

Amazon MSK クラスターと Kafka トピックを Lambda 関数のトリガーとして追加するには、次の手順を実行します。

Amazon MSK トリガーをLambda 関数 (コンソール) に追加するには

1. Lambda コンソールの [Functions (関数)] ページを開きます。
2. Lambda 関数の名前を選択します。
3. [機能の概要] で、[トリガーを追加] を選択します。
4. [Trigger configuration] (トリガー設定) で次の操作を実行します。
    a. MSKトリガータイプを選択します。
    b. [MSK cluster] (MSK クラスター) で、クラスターを選択します。
    c. [Batch size] (バッチサイズ) で、単一バッチで取得されるメッセージの最大数を設定します。
    d. バッチウィンドウ では、Lambda が関数を呼び出すまで費やすレコード収集の最大時間 (秒) を入力します。
    e. [Topic name] (トピック名)に、Kafka トピックの名前を入力します。
    f. (オプション) コンシューマーグループ ID で、参加する Kafka コンシューマーグループの ID を入力します。

g. (オプション) [開始位置] で、[最新] を選択して最新のレコードからストリームの読み取りを開始するか、[水平トリム] を選択して使用可能な最も以前のレコードから開始するか、または [タイムスタンプ時点] を選択して読み取りを開始するタイムスタンプを指定します。

h. (オプション) [Authentication] (認証) で、MSK クラスターのブローカーで認証するためのシークレットキーを選択します。

i. テスト用に無効状態のトリガーを作成する (推奨) には、[Enable trigger] (トリガーを有効にする) を解除します。または、トリガーをすぐに有効にするには、[Enable trigger] (トリガーを有効にする) を選択します。

5. トリガーを追加するには、[Add] (追加) を選択します。

Amazon MSK トリガーの追加 (AWS CLI)

Lambda 関数の Amazon MSK トリガーを作成および表示するには、次の例の AWS CLI コマンドを使用します。

AWS CLI を使用したトリガーの作成

Example — IAM 認証を使用するクラスターのイベントソースマッピングを作成します。

次の例では、`create-event-source-mapping` AWS CLI コマンドを使用して、Lambda 関数 `my-kafka-function` を、Kafka トピック `AWSKafkaTopic` にマップします。トピックの開始位置は `LATEST` に設定します。クラスターが IAM ロールベースの認証を使用する場合、SourceAccessConfiguration オブジェクトは必要ありません。例：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — SASL/SCRAM 認証を使用するクラスターのイベントソースマッピングを作成します。

クラスターが SASL/SCRAM 認証を使用する場合は、`SASL_SCRAM_512_AUTH` および Secrets Manager のシークレット ARN を指定する SourceAccessConfiguration オブジェクトを含める必要があります。

```
aws lambda create-event-source-mapping \
```

```
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

Example — mTLS 認証を使用するクラスターのイベントソースマッピングを作成します。

クラスターが mTLS 認証を使用する場合は、CLIENT_CERTIFICATE_TLS_AUTH および Secrets Manager のシークレット ARN を指定する SourceAccessConfiguration オブジェクトを含める必要があります。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

詳細については、API リファレンスドキュメント CreateEventSourceMapping を参照してください。

AWS CLI を使用したステータスの表示

次の例では、get-event-source-mapping AWS CLI コマンドを使用して、作成したイベントソースマッピングのステータスを記述します。

```
aws lambda get-event-source-mapping \
  --uuid 6d9bce8e-836b-442c-8070-74e77903c815
```

クロスアカウントのイベントソースマッピングの作成

マルチ VPC プライベート接続を使用して、Lambda 関数を別の AWS アカウントのプロビジョニングされた MSK クラスターに接続できます。マルチ VPC 接続は AWS PrivateLink を使用して、すべてのトラフィックを AWS ネットワーク内に保持します。

> **Note**
>
> サーバーレス MSK クラスターにはクロスアカウントイベントソースマッピングを作成できません。

クロスアカウントイベントソースマッピングを作成するには、まず MSK クラスターのマルチ VPC 接続を設定する必要があります。イベントソースマッピングを作成するときは、以下の例に示すように、クラスター ARN の代わりにマネージド VPC 接続 ARN を使用します。CreateEventSourceMapping オペレーションは、MSK クラスターが使用する認証タイプによっても異なります。

Example — IAM 認証を使用するクラスターのクロスアカウントイベントソースマッピングを作成します。

クラスターが IAM ロールベースの認証を使用する場合、SourceAccessConfiguration オブジェクトは必要ありません。例：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example — SASL/SCRAM 認証を使用するクラスターのクロスアカウントイベントソースマッピングを作成します。

クラスターが SASL/SCRAM 認証を使用する場合は、SASL_SCRAM_512_AUTH および Secrets Manager のシークレット ARN を指定する SourceAccessConfiguration オブジェクトを含める必要があります。

SASL/SCRAM 認証でクロスアカウントの Amazon MSK イベントソースマッピングにシークレットを使用する方法は 2 つあります。

- Lambda 関数アカウントにシークレットを作成し、クラスターシークレットと同期します。2 つのシークレットを同期させるローテーションを作成します。このオプションでは、関数アカウントからシークレットを制御できます。
- MSK クラスターに関連付けられているシークレットを使用してください。このシークレットは、Lambda 関数アカウントへのクロスアカウントアクセスを許可する必要があります。詳細につ

いては、「別のアカウントのユーザーの AWS Secrets Manager シークレットに対するアクセス許可」を参照してください。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example — mTLS 認証を使用するクラスターのクロスアカウントイベントソースマッピングを作成します。

クラスターが mTLS 認証を使用する場合は、CLIENT_CERTIFICATE_TLS_AUTH および Secrets Manager のシークレット ARN を指定する SourceAccessConfiguration オブジェクトを含める必要があります。シークレットは、クラスターアカウントまたは Lambda 関数アカウントに保存できます。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

障害発生時の送信先

失敗したイベントソースマッピング呼び出しの記録を保持するには、関数のイベントソースマッピングに送信先を追加します。送信先に送られる各レコードは、失敗した呼び出しに関するメタデータを含む JSON ドキュメントです。任意の Amazon SNS トピック、Amazon SQS キュー、または S3 バケットを送信先として設定できます。実行ロールには、送信先に対するアクセス許可が必要です。

- SQS 送信先の場合: sqs:SendMessage
- SNS 送信先の場合: sns:Publish
- S3 バケット送信先の場合: s3:PutObject および s3:ListBuckets

さらに、送信先に KMS キーを設定した場合、Lambda には送信先のタイプに応じて以下のアクセス許可が必要です。

- S3 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:GenerateDataKey が必要です。KMS キーと S3 バケットの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、kms:GenerateDataKey を許可するように実行ロールを信頼するように KMS キーを設定します。
- SQS 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:Decrypt および kms:GenerateDataKey が必要です。KMS キーと SQS キューの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、KMS キーが実行ロールを信頼し、kms:Decryp、kms:GenerateDataKey、kms:DescribeKey、および kms:ReEncrypt を許可するように設定します。
- SNS 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:Decrypt と kms:GenerateDataKey が必要です。KMS キーと SNS トピックの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、KMS キーが実行ロールを信頼し、kms:Decryp、kms:GenerateDataKey、kms:DescribeKey、および kms:ReEncrypt を許可するように設定します。

障害発生時の送信先をコンソールを使用して設定するには、以下の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [機能の概要] で、[送信先を追加] を選択します。
4. [ソース] には、[イベントソースマッピング呼び出し] を選択します。
5. [イベントソースマッピング] では、この関数用に設定されているイベントソースを選択します。
6. [条件] には [失敗時] を選択します。イベントソースマッピング呼び出しでは、これが唯一受け入れられる条件です。
7. [送信先タイプ] では、Lambda が呼び出しレコードを送信する送信先タイプを選択します。
8. [送信先] で、リソースを選択します。
9. [Save] を選択します。

AWS CLI を使用して障害発生時の送信先を設定することもできます。例えば、次の create-event-source-mapping コマンドは、SQS を障害発生時の送信先として持つイベントソースマッピングを `MyFunction` に追加します。

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下の update-event-source-mapping コマンドは、S3 の障害発生時の送信先を、入力 uuid に関連付けられたイベントソースに追加します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

送信先を削除するには、destination-config パラメータの引数として空の文字列を指定します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS および SQS の呼び出しレコードの例

以下の例は、Kafka イベントソース呼び出しが失敗した場合に Lambda が SNS トピックまたは SQS キューの送信先に送信する内容を示しています。recordsInfo の各キーには、Kafka トピックとパーティションの両方がハイフンで区切られて含まれています。例えば、キー "Topic-0" の場合、Topic は Kafka トピック、0 はパーティションです。各トピックとパーティションについて、オフセットとタイムスタンプデータを使用して元の呼び出しレコードを検索できます。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
```

```
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 送信先の呼び出しレコードの例

S3 の送信先の場合、Lambda は呼び出しレコード全体をメタデータと共に送信先に送信します。以下の例は、Kafka イベントソース呼び出しが失敗した場合に、Lambda が S3 バケットの送信先に送信することを示しています。SQS と SNS の送信先に関する前例のすべてのフィールドに加えて、payload フィールドには元の呼び出しレコードがエスケープされた JSON 文字列として含まれています。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
```

```
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 送信先バケットで S3 バージョニングを有効にすることをお勧めします。

Amazon MSK イベントソースの Auto Scaling

初めて Amazon MSK イベントソースを作成するときは、Lambda が Kafka トピック内のすべてのパーティションを処理するために 1 つのコンシューマーを割り当てます。各コンシューマーには、増加したワークロードを処理するために同時実行される複数のプロセッサがあります。さらに、Lambda は、ワークロードに基づいてコンシューマーの数を自動的にスケールアップまたはスケールダウンします。各パーティションでメッセージの順序を保つため、コンシューマーの最大数は、トピック内のパーティションあたり 1 つとなっています。

Lambda は、1 分間隔でトピック内のすべてのパーティションのコンシューマーオフセット遅延を評価します。遅延が大きすぎる場合、パーティションは Lambda で処理可能な速度よりも速い速度でメッセージを受信します。必要に応じて、Lambda はトピックにコンシューマーを追加するか、またはトピックからコンシューマーを削除します。コンシューマーを追加または削除するスケーリングプロセスは、評価から 3 分以内に行われます。

ターゲットの Lambda 関数がスロットリングされると、Lambda はコンシューマーの数を減らします。このアクションにより、コンシューマーが取得し関数に送信するメッセージの数が減り、関数への負荷が軽減されます。

Kafka トピックのスループットをモニタリングするには、関数がレコードを処理する間に Lambda が発行するオフセット遅延メトリクスを表示してください。

いくつの関数呼び出しが並行して発生しているかを確認するときは、関数の同時実行メトリクスも監視します。

ポーリングとストリームの開始位置

イベントソースマッピングの作成時および更新時のストリームのポーリングは、最終的に一貫性があることに注意してください。

- イベントソースマッピングの作成時、ストリームからのイベントのポーリングが開始されるまでに数分かかる場合があります。
- イベントソースマッピングの更新時、ストリームからのイベントのポーリングが停止および再開されるまでに数分かかる場合があります。

つまり、LATEST をストリームの開始位置として指定すると、イベントソースマッピングの作成または更新中にイベントを見逃す可能性があります。イベントを見逃さないようにするには、ストリームの開始位置を TRIM_HORIZON または AT_TIMESTAMP として指定します。

Amazon CloudWatch メトリクス

Lambda は、関数がレコードを処理している間に `OffsetLag` メトリクスを発行します。このメトリクスの値は、Kafka イベントソーストピックに書き込まれた最後のレコードと関数のコンシューマグループが処理した最後のレコードの間のオフセットの差分です。レコードが追加されてからコンシューマグループがそれを処理するまでのレイテンシーを見積もるには、`OffsetLag` を使用できます。

`OffsetLag` の増加傾向は、関数のコンシューマグループに問題があることを示している可能性があります。詳細については、「Lambda 関数のメトリクスの使用」を参照してください。

Amazon MSK 設定パラメータ

すべての Lambda イベントソースタイプは、同じCreateEventSourceMapping および UpdateEventSourceMapping API オペレーションを共有しています。ただし、Amazon MSK に適用されるのは一部のパラメータのみです。

Amazon MSK に適用されるイベントソースパラメータ

Parameter	必須	デフォルト	メモ
AmazonManagedKafkaEventSourceConfig	N	ConsumerGroupID フィールドを含み、デフォルトでは一意の値になっています。	作成時にのみ設定可能
BatchSize	N	100	最大: 10,000
有効	N	有効	
EventSourceArn	Y		作成時にのみ設定可能
FunctionName	Y		
FilterCriteria	N		Lambda のイベントフィルタリング
MaximumBatchingWindowInSeconds	N	500 ミリ秒	バッチ処理動作

Parameter	必須	デフォルト	メモ
SourceAccessConfigurations	N	認証情報なし	イベントソース用の、SASL/SCRAM あるいは CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) の認証情報
StartingPosition	Y		AT_TIMESTAMP、TRIM_HORIZON、または LATEST 作成時にのみ設定可能
StartingPositionTimestamp	N		StartingPosition が AT_TIMESTAMP に設定されている場合にのみ必要
トピック	Y		カフカのトピック名 作成時にのみ設定可能

セルフマネージド型の Apache Kafka で Lambda を使用する

> **Note**
>
> Lambda 関数以外のターゲットにデータを送信したい、または送信する前にデータをエンリッチしたいという場合は、「Amazon EventBridge Pipes」を参照してください。

Lambdaは、Apache Kafka をイベントソースソースとしてサポートしています。Apache Kafka は、データパイプラインやストリーミング分析などのワークロードをサポートする、オープンソースのイベントストリーミングプラットフォームです。

ユーザーは、AWS マネージドの Kafka サービス、Amazon Managed Streaming for Apache Kafka (Amazon MSK)、またはセルフマネージドの Kafka クラスターを使用できます。Amazon MSK で Lambda を使用する方法の詳細については、Amazon MSK で Lambda を使用する を参照してください。

このトピックでは、セルフマネージド型の Kafka クラスターで Lambda を使用する方法を説明します。AWS の用語集では、セルフマネージド型クラスターには、非 AWS のホストされた Kafka クラスターが含まれています。たとえば、お使いの Kafka クラスターを、Confluent Cloud などのクラウドプロバイダーでホストすることが可能です。

イベントソースとしての Apache Kafka は、Amazon Simple Queue Service (Amazon SQS) または Amazon Kinesis を使用する場合と同様に動作します。Lambda は、イベントソースからの新しいメッセージを内部的にポーリングした後、ターゲットの Lambda 関数を同期的に呼び出します。Lambda はメッセージをバッチで読み込み、それらをイベントペイロードとして関数に提供します。最大バッチサイズは調整可能です。(デフォルト値は 100 メッセージ)。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

Kafka ベースのイベントソースの場合、Lambda はバッチ処理ウィンドウやバッチサイズなどの制御パラメータの処理をサポートします。詳しくは、「バッチ処理動作」を参照してください。

セルフマネージド型 Kafka をイベントソースとして使用する方法の例については、AWS Compute Blog の Using self-hosted Apache Kafka as an event source for AWS Lambda を参照してください。

トピック

- イベントの例
- Kafka クラスター認証
- API アクセスと許可の管理

- 認証と認可のエラー
- ネットワーク構成
- Kafka クラスターをイベントソースとして追加する
- 障害発生時の送信先
- Kafka クラスターをイベントソースとして使用する
- ポーリングとストリームの開始位置
- Kafka イベントソースのオートスケーリング
- イベントソースマッピングエラー
- Amazon CloudWatch メトリクス
- セルフマネージド Apache Kafka の設定パラメータ

イベントの例

Lambda は、Lambda 関数を呼び出すとき、イベントパラメータ内のメッセージのバッチを送信します。イベントペイロードにはメッセージの配列が含まれています。各配列項目には、Kafka トピックと Kafka パーティション識別子の詳細が、タイムスタンプおよび base64 でエンコードされたメッセージとともに含まれています。

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
```

```
                    101,
                    97,
                    100,
                    101,
                    114,
                    86,
                    97,
                    108,
                    117,
                    101
                ]
            }
        ]
    }
  ]
 }
}
```

Kafka クラスター認証

Lambda は、セルフマネージド型 Apache Kafka クラスターで認証するための方法をいくつかサポートしています。これらのサポートされる認証方法のいずれかを使用するように、Kafka クラスターを設定しておいてください。Kafka セキュリティの詳細については、Kafka ドキュメントの「Security」(セキュリティ) セクションを参照してください。

VPC アクセス

VPC 内の Kafka ユーザーのみが Kafka ブローカーにアクセスする場合は、Amazon Virtual Private Cloud (Amazon VPC) アクセス用に Kafka イベントソースを設定する必要があります。

SASL/SCRAM 認証

Lambda は、Transport Layer Security (TLS) 暗号化 (`SASL_SSL`) を使用した Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 認証をサポートしています。Lambda は、暗号化された認証情報を送信してクラスターで認証します。Lambda は plaintext の SASL/PLAIN (`SASL_PLAINTEXT`) をサポートしません。SASL/SCRAM 認証の詳細については、「RFC 5802」を参照してください。

Lambda は SASL/PLAIN 認証もサポートします。このメカニズムはクリアテキスト認証情報を使用するので、この認証情報が保護されることを確実にするためにも、サーバーへの接続には TLS 暗号化を使用する必要があります。

SASL 認証の場合は、サインイン認証情報をシークレットとして AWS Secrets Manager に保存します。Secrets Manager の使用に関する詳細については、「AWS Secrets Manager ユーザーガイド」の「チュートリアル: シークレットの作成と取得」を参照してください。

> **Important**
>
> 認証に Secrets Manager を使用するには、シークレットを Lambda 関数と同じ AWS リージョンに保存する必要があります。

相互 TLS 認証

相互 TLS (mTLS) は、クライアントとサーバー間の双方向認証を提供します。クライアントは、サーバーによるクライアントの検証のためにサーバーに証明書を送信し、サーバーは、クライアントによるサーバーの検証のためにクライアントに証明書を送信します。

セルフマネージド Apache Kafka では、Lambda がクライアントとして機能します。Kafka ブローカーで Lambda を認証するように、クライアント証明書を (Secrets Manager のシークレットとして) 設定します。クライアント証明書は、サーバーのトラストストア内の CA によって署名される必要があります。

Kafka クラスターは、Lambda で Kafka ブローカーを認証するために Lambda にサーバー証明書を送信します。サーバー証明書は、パブリック CA 証明書またはプライベート CA/自己署名証明書にすることができます。パブリック CA 証明書は、Lambda トラストストア内の認証局 (CA) によって署名される必要があります。プライベート CA/自己署名証明書の場合は、サーバルート CA 証明書を (Secrets Manager のシークレットとして) 設定します。Lambda はルート証明書を使用して Kafka ブローカーを検証します。

mTLS の詳細については、「Introducing mutual TLS authentication for Amazon MSK as an event source」(イベントソースとしての Amazon MSK のための相互 TLS の紹介) を参照してください。

クライアント証明書シークレットの設定

CLIENT_CERTICATE_TLS_AUTH シークレットは、証明書フィールドとプライベートキーフィールドを必要とします。暗号化されたプライベートキーの場合、シークレットはプライベートキーのパスワードを必要とします。証明書とプライベートキーは、どちらも PEM 形式である必要があります。

> **Note**
>
> Lambda は、PBES1 (PBES2 ではありません) プライベートキー暗号化アルゴリズムをサポートします。

証明書フィールドには、クライアント証明書で始まり、その後に中間証明書が続き、ルート証明書で終わる証明書のリストが含まれている必要があります。各証明書は、以下の構造を使用した新しい行で始める必要があります。

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager は最大 65,536 バイトのシークレットをサポートします。これは、長い証明書チェーンにも十分な領域です。

プライベートキーは、以下の構造を使用した PKCS #8 形式にする必要があります。

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

暗号化されたプライベートキーには、以下の構造を使用します。

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

以下は、暗号化されたプライベートキーを使用する mTLS 認証のシークレットの内容を示す例です。暗号化されたプライベートキーの場合は、シークレットにプライベートキーのパスワードを含めます。

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
```

```
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

サーバルート CA 証明書シークレットの設定

このシークレットは、Kafka ブローカーがプライベート CA によって署名された証明書で TLS 暗号化を使用する場合に作成します。TLS 暗号化は、VPC、SASL/SCRAM、SASL/PLAIN、または mTLS 認証に使用できます。

サーバールート CA 証明書シークレットには、PEM 形式の Kafka ブローカーのルート CA 証明書が含まれるフィールドが必要です。以下は、このシークレットの構造を示す例です。

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

API アクセスと許可の管理

セルフマネージド Kafka クラスターへのアクセスに加えて、Lambda 関数にはさまざまな API アクションを実行するための許可が必要です。これらの許可は、関数の実行ロールに追加します。ユーザーが API アクションのいずれかにアクセスする必要がある場合は、AWS Identity and Access Management (IAM) ユーザーまたはロールのアイデンティティポリシーに必要な許可を追加します。

Lambda 関数に必要なアクセス許可

Amazon CloudWatch Logs のロググループでログを作成して保存するには、Lambda 関数の実行ロールに以下の許可が必要です。

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Lambda 関数のオプションのアクセス許可

Lambda 関数には、以下を実行する許可も必要になる場合があります。

- Secrets Manager シークレットを記述する。
- AWS Key Management Service (AWS KMS) カスタマー管理のキーにアクセスする。
- Amazon VPC にアクセスする。
- 失敗した呼び出しのレコードを送信先に送信します。

Secrets Manager と AWS KMS 許可

Kafka ブローカーに設定しているアクセスコントロールのタイプに応じて、Lambda 関数には Secrets Manager シークレットにアクセスするための許可、または AWS KMS カスタマーマネージドキーを復号化するための許可が必要になる場合があります。それらのリソースにアクセスするには、関数の実行ロールに次のアクセス許可が必要です。

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC アクセス許可

セルフマネージド Apache Kafka クラスターにアクセスできるのが VPC 内のユーザーのみである場合、Lambda 関数には Amazon VPC リソースにアクセスするための許可が必要です。これらのリソースには、VPC、サブネット、セキュリティグループ、ネットワークインターフェイスが含まれます。それらのリソースにアクセスするには、関数の実行ロールに次のアクセス許可が必要です。

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets

- ec2:DescribeSecurityGroups

実行ロールへのアクセス許可の追加

セルフマネージド型 Apache Kafka クラスターが使用するその他の AWS サービスにアクセスするために、Lambda は、関数の実行ロールで定義されたアクセス許可ポリシーを使用します。

デフォルトでは、Lambda は、セルフマネージド型 Apache Kafka クラスターに対して、必須のまたはオプションのアクションを実行することはできません。これらのアクションを IAM 信頼ポリシーで作成および定義し、そのポリシーを実行ロールにアタッチする必要があります。この例では、Lambda に Amazon VPC リソースへのアクセスを許可する、ポリシーの作成方法を紹介します。

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
           }
        ]
     }
```

IAM コンソールで JSON ポリシードキュメントを作成する方法については、IAM ユーザーガイド の[JSON] タブでのポリシーの作成を参照してください。

IAM ポリシーを使用したユーザーアクセスの許可

デフォルトでは、ユーザーおよびロールにはイベントソースの API オペレーションを実行するアクセス許可がありません。組織またはアカウント内のユーザーにアクセス権を付与するには、アイデンティティベースのポリシーを作成または更新します。詳細については、IAM ユーザーガイドのポリシーを使用した AWS リソースへのアクセスのコントロールを参照してください。

認証と認可のエラー

Kafka クラスターからのデータを消費するために必要な許可のいずれかが欠落している場合、Lambda は [LastProcessingResult] のイベントソースマッピングに以下のエラーメッセージのいずれかを表示します。

エラーメッセージ

- クラスターが Lambda の認可に失敗した
- SASL 認証に失敗した
- Server failed to authenticate Lambda (サーバーが Lambda の認証に失敗しました)
- Lambda failed to authenticate server (Lambda がサーバーの認証に失敗しました)
- Provided certificate or private key is invalid (提供された証明書またはプライベートキーが無効です)

クラスターが Lambda の認可に失敗した

SALS/SCRAM または mTLS の場合、このエラーは、指定されたユーザーが以下の必要とされる Kafka アクセスコントロールリスト (ACL) 許可のすべてを持っていないことを示します。

- DescribeConfigs クラスター
- グループを記述する
- グループを読み取る
- トピックを記述する
- トピックを読み取る

必要な `kafka-cluster` 許可を使用して Kafka ACL を作成するときは、リソースとしてトピックとグループを指定します。トピック名は、イベントソースマッピングのトピックと一致する必要があります。グループ名は、イベントソースマッピングの UUID と一致する必要があります。

必要な許可を実行ロールに追加した後は、変更が有効になるまで数分間かかる場合があります。

SASL 認証に失敗した

SASL/SCRAM または SASL/PLAIN の場合、このエラーは指定されたサインイン認証情報が無効であることを示します。

Server failed to authenticate Lambda (サーバーが Lambda の認証に失敗しました)

このエラーは、Kafka ブローカーが Lambda の認証に失敗したことを示します。このエラーは、以下が原因で発生する可能性があります。

- mTLS 認証用のクライアント証明書を提供していない。
- クライアント証明書を提供したが、Kafka ブローカーが mTLS 認証を使用するように設定されていない。
- クライアント証明書が Kafka ブローカーに信頼されていない。

Lambda failed to authenticate server (Lambda がサーバーの認証に失敗しました)

このエラーは、Lambda が Kafka ブローカーの認証に失敗したことを示します。このエラーは、以下が原因で発生する可能性があります。

- Kafka ブローカーは自己署名証明書またはプライベート CA を使用するが、サーバールート CA 証明書を提供しなかった。
- サーバールート CA 証明書が、ブローカーの証明書に署名したルート CA と一致しない。
- ブローカーの証明書にサブジェクトの別名としてブローカーの DNS 名または IP アドレスが含まれていないため、ホスト名の検証が失敗した。

Provided certificate or private key is invalid (提供された証明書またはプライベートキーが無効です)

このエラーは、Kafka コンシューマーが提供された証明書またはプライベートキーを使用できなかったことを示します。証明書とキーが PEM 形式を使用しており、プライベートキーの暗号化が PBES1 アルゴリズムを使用していることを確認してください。

ネットワーク構成

Lambda が Kafka クラスターをイベントソースとして使用するには、クラスターが存在する Amazon VPC にアクセスする必要があります。VCP にアクセスするには、Lambda に AWS PrivateLink 「VPC エンドポイント」をデプロイすることをお勧めします。Lambda および AWS Security Token Service (AWS STS) にエンドポイントをデプロイします。ブローカーが認証を使用する場合は、Secrets Manager 用の VPC エンドポイントもデプロイします。失敗時の送信先を設定した場合は、送信先サービスの VPC エンドポイントもデプロイします。

または、Kafka クラスターに関連付けられた VPC に、パブリックサブネットごとに 1 つの NAT ゲートウェイが含まれていることを確認します。詳細については、「the section called “VPC 関数のインターネットアクセス”」を参照してください。

VPC エンドポイントを使用する場合は、プライベート DNS 名を有効にするように設定する必要もあります。

自己管理型の Apache Kafka クラスターのイベントソースマッピングを作成すると、Lambda はクラスターの VPC のサブネットおよびセキュリティグループに Elastic Network Interface (ENI) が既に存在するかどうかを確認します。Lambda が既存の ENI を検出した場合、再利用しようとします。それ以外の場合、Lambda は新しい ENI を作成し、イベントソースに接続して関数を呼び出します。

> **Note**
>
> Lambda 関数は、Lambda サービスが所有する VPC 内で常に実行されます。これらの VPC はサービスによって自動的に管理され、顧客には表示されません。関数を Amazon VPC に接続することもできます。いずれの場合、関数の VPC 設定はイベントソースマッピングに影響しません。Lambda がイベントソースに接続する方法を判定するのは、イベントソースの VPC の設定のみです。

ネットワークの設定方法の詳細については、AWS Compute Blog の Setting up AWS Lambda with an Apache Kafka cluster within a VPC を参照してください。

VPC セキュリティグループのルール

次のルール (最低限) に従ってクラスターを含む Amazon VPC のセキュリティグループを設定してください。

- インバウンドルール – イベントソース用に指定されたセキュリティグループに対して Kafka ブローカーポート上のすべてのトラフィックを許可します。Kafka は、デフォルトでポート 9092 を使用します。
- アウトバウンドルール – すべての送信先に対して、ポート 443 上のすべてのトラフィックを許可します。イベントソース用に指定されたセキュリティグループに対して Kafka ブローカーポート上のすべてのトラフィックを許可します。Kafka は、デフォルトでポート 9092 を使用します。
- NAT ゲートウェイの代わりに VPC エンドポイントを使用している場合は、その VPC エンドポイントに関連付けられたセキュリティグループが、イベントソースのセキュリティグループからのポート 443 上のすべてのインバウンドトラフィックを許可する必要があります。

VPCエンドポイントの使用

VPC エンドポイントを使用するとき、ENI を使用し、関数を呼び出す API コールはこれらのエンドポイントを経由してルーティングされます。Lambda サービスプリンシパルは、これらの ENI を使用するすべてのロールおよび関数に対し、`sts:AssumeRole` および `lambda:InvokeFunction` を呼び出す必要があります。

デフォルトでは、VPC エンドポイントはオープンな IAM ポリシーがあります。特定のプリンシパルのみがそのエンドポイントを使用して必要なアクションを実行するため、これらのポリシーを制限することがベストプラクティスです。イベントソースマッピングが Lambda 関数を呼び出せるようにするには、VPC エンドポイントポリシーは Lambda サービスプリンシパルが `sts:AssumeRole` および `lambda:InvokeFunction` を呼び出せるようにする必要があります。組織内で発生する API コールのみを許可するように VPC エンドポイントポリシーを制限すると、イベントソースマッピングが正しく機能しなくなります。

次の VPC エンドポイントポリシーの例では、AWS STS および Lambda エンドポイントに Lambda サービスプリンシパルの必要なアクセスを付与する方法について示しています。

Example VPC エンドポイントポリシー - AWS STS エンドポイント

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC エンドポイントポリシー – Lambda エンドポイント

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
```

```
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Kafka ブローカーが認証を使用する場合、Secrets Manager エンドポイントの VPC エンドポイントポリシーを制限することもできます。Secrets Manager API を呼び出す場合、Lambda は Lambda サービスプリンシパルではなく、関数ロールを使用します。次の例では、Secrets Manager エンドポイントポリシーを示します。

Example VPC エンドポイントポリシー - Secrets Manager エンドポイント

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

障害が発生している宛先が設定されている場合、Lambda は関数のロールを使用し、Lambda が管理する ENI で `s3:PutObject`、`sns:Publish`、`sqs:sendMessage` のいずれかを呼び出します。

Kafka クラスターをイベントソースとして追加する

イベントソースマッピングを作成するには、Lambda コンソール、AWSSDK、またはAWS Command Line Interface(AWS CLI) を使用して、Kafka クラスターを Lambda 関数のトリガーとして追加します。

このセクションでは、Lambda コンソールと AWS CLI を使用してイベントソースマッピングを作成する方法について説明します。

前提条件

- セルフマネージド型 Apache Kafka クラスター。Lambda は、Apache Kafka バージョン 0.10.1.0 以降をサポートしています。
- セルフマネージド Kafka クラスターが使用する AWS リソースにアクセスするための許可を持つ実行ロール。

カスタマイズ可能なコンシューマーグループ ID

Kafkaをイベントソースとして設定する場合、コンシューマーグループIDを指定できます。このコンシューマーグループ ID は、Lambda 関数を結合したい Kafka コンシューマーグループの既存の識別子です。この機能を使用すると、実行中の Kafka レコード処理設定を他のコンシューマーから Lambda にシームレスに移行できます。

コンシューマーグループ ID を指定し、そのコンシューマーグループ内に他のアクティブなポーラーが存在する場合、Kafka はすべてのコンシューマーにメッセージを配信します。言い換えると、Lambda は Kafka トピックのメッセージをすべて受け取るわけではありません。Lambda にトピック内のすべてのメッセージを処理させたい場合は、そのコンシューマーグループの他のポーラーをすべてオフにします。

さらに、コンシューマーグループ ID を指定し、Kafka が同じ ID を持つ有効な既存のコンシューマーグループを見つけた場合、Lambda は、イベントソースマッピングの `StartingPosition` パラメーターを無視します。代わりに、Lambda はコンシューマーグループのコミットされたオフセットに従ってレコードの処理を開始します。コンシューマーグループ ID を指定しても、Kafka が既存のコンシューマーグループを見つけられない場合、Lambda は指定された `StartingPosition` を使用してイベントソースを設定します。

指定するコンシューマーグループ ID は、すべての Kafka イベントソースの中で一意でなければなりません。コンシューマーグループ ID を指定して Kafka イベントソースマッピングを作成した後は、この値を更新することはできません。

セルフマネージド型 Kafka クラスターを追加する (コンソール)

セルフマネージド型 Apache Kafka クラスターと Kafka トピックを Lambda 関数のトリガーとして追加するには、次の手順を実行します。

Apache Kafka トリガーを Lambda 関数に追加するには (コンソール)

1. Lambda コンソールの [Functions] (関数) ページを開きます。
2. Lambda 関数の名前を選択します。
3. [機能の概要] で、[トリガーを追加] を選択します。
4. [Trigger configuration] (トリガー設定) で次の操作を実行します。

   a. Apache Kafka のトリガータイプを選択します。

   b. [Bootstrap servers] (ブートストラップサーバー) に、クラスター内の Kafka ブローカーのホストおよびポートのペアアドレスを入力し、[Add] (追加) を選択します。クラスター内の各 Kafka ブローカーで上記を繰り返します。

   c. [Topic name] (トピック名) に、クラスター内のレコードの保存に使用する Kafka トピックの名前を入力します。

   d. (オプション) [Batch size] (バッチサイズ) に、単一のバッチで取得できるメッセージの最大数を入力します。

   e. バッチウィンドウ では、Lambda が関数を呼び出すまで費やすレコード収集の最大時間 (秒) を入力します。

   f. (オプション) コンシューマーグループ ID で、参加する Kafka コンシューマーグループの ID を入力します。

   g. (オプション) [開始位置] で、[最新] を選択して最新のレコードからストリームの読み取りを開始するか、[水平トリム] を選択して使用可能な最も以前のレコードから開始するか、または [タイムスタンプ時点] を選択して読み取りを開始するタイムスタンプを指定します。

   h. (オプション) [VPC] で、Kafka クラスターに Amazon VPC を選択します。次に、[VPC subnets] (VPC サブネット) と [VPC security groups] (VPC セキュリティグループ) を選択します。

      VPC 内のユーザーのみがブローカーにアクセスする場合、この設定は必須です。

   i. (オプション) [Authentication] (認証) で [Add] (追加) をクリックしてから、以下を実行します。

      i. クラスター内の Kafka ブローカーのアクセスまたは認証プロトコルを選択します。

         - Kafka ブローカーが SASL/PLAIN 認証を使用する場合は、[BASIC_AUTH] を選択します。
         - ブローカーが SALS/SCRAM 認証を使用する場合は、[SASL_SCRAM] プロトコルのいずれかを選択します。

- mTLS 認証を設定している場合は、[CLIENT_CERTIFICATE_TLS_AUTH] プロトコルを選択します。

ii. SASL/SCRAM または mTLS 認証の場合は、Kafka クラスターの認証情報が含まれる Secrets Manager シークレットキーを選択します。

j. (オプション) Kafka ブローカーがプライベート CA によって署名された証明書を使用する場合、[Encryption] (暗号化) には Kafka ブローカーが TLS 暗号化に使用するルート CA 証明書が含まれる Secrets Manager シークレットを選択します。

この設定は、SASL/SCRAM または SASL/PLAIN の TLS 暗号化、および mTLS 認証に適用されます。

k. テスト用に無効状態のトリガーを作成する (推奨) には、[Enable trigger] (トリガーを有効にする) を解除します。または、トリガーをすぐに有効にするには、[Enable trigger] (トリガーを有効にする) を選択します。

5. トリガーを追加するには、[Add] (追加) を選択します。

セルフマネージド型 Kafka クラスターを追加する (AWS CLI)

Lambda 関数のセルフマネージド型 Apache Kafka トリガーを作成および表示するには、次の AWS CLI コマンドの例を使用します。

SASL/SCRAM を使用する

Kafka ユーザーがインターネット経由で Kafka ブローカーにアクセスする場合は、SASL/SCRAM 認証用に作成した Secrets Manager シークレットを指定します。次の例では、`create-event-source-mapping` AWS CLI コマンドを使用して、Lambda 関数 `my-kafka-function` を、Kafka トピック `AWSKafkaTopic` にマップします。

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

VPC の使用

Kafka ブローカーにアクセスするのが VPC 内の Kafka ユーザーのみである場合、VPC、サブネット、および VPC セキュリティグループを指定する必要があります。次の例では、`create-event-`

source-mapping AWS CLI コマンドを使用して、Lambda 関数 my-kafka-function を、Kafka トピック AWSKafkaTopic にマップします。

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

AWS CLI を使用したステータスの表示

次の例では、get-event-source-mapping AWS CLI コマンドを使用して、作成したイベントソースマッピングのステータスを記述します。

```
aws lambda get-event-source-mapping
          --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

障害発生時の送信先

失敗したイベントソースマッピング呼び出しの記録を保持するには、関数のイベントソースマッピングに送信先を追加します。送信先に送られる各レコードは、失敗した呼び出しに関するメタデータを含む JSON ドキュメントです。任意の Amazon SNS トピック、Amazon SQS キュー、または S3 バケットを送信先として設定できます。実行ロールには、送信先に対するアクセス許可が必要です。

- SQS 送信先の場合: sqs:SendMessage
- SNS 送信先の場合: sns:Publish
- S3 バケット送信先の場合: s3:PutObject および s3:ListBuckets

さらに、送信先に KMS キーを設定した場合、Lambda には送信先のタイプに応じて以下のアクセス許可が必要です。

- S3 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:GenerateDataKey が必要です。KMS キーと S3 バケットの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、kms:GenerateDataKey を許可するように実行ロールを信頼するように KMS キーを設定します。

- SQS 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:Decrypt および kms:GenerateDataKey が必要です。KMS キーと SQS キューの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、KMS キーが実行ロールを信頼し、kms:Decryp、kms:GenerateDataKey、kms:DescribeKey、および kms:ReEncrypt を許可するように設定します。
- SNS 送信先に対して独自の KMS キーによる暗号化を有効にしている場合は、kms:Decrypt と kms:GenerateDataKey が必要です。KMS キーと SNS トピックの送信先が Lambda 関数および実行ロールとは異なるアカウントにある場合は、KMS キーが実行ロールを信頼し、kms:Decryp、kms:GenerateDataKey、kms:DescribeKey、および kms:ReEncrypt を許可するように設定します。

障害発生時の送信先をコンソールを使用して設定するには、以下の手順に従います。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [機能の概要] で、[送信先を追加] を選択します。
4. [ソース] には、[イベントソースマッピング呼び出し] を選択します。
5. [イベントソースマッピング] では、この関数用に設定されているイベントソースを選択します。
6. [条件] には [失敗時] を選択します。イベントソースマッピング呼び出しでは、これが唯一受け入れられる条件です。
7. [送信先タイプ] では、Lambda が呼び出しレコードを送信する送信先タイプを選択します。
8. [送信先] で、リソースを選択します。
9. [Save] を選択します。

AWS CLI を使用して障害発生時の送信先を設定することもできます。例えば、次の create-event-source-mapping コマンドは、SQS を障害発生時の送信先として持つイベントソースマッピングを MyFunction に追加します。

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

以下の update-event-source-mapping コマンドは、S3 の障害発生時の送信先を、入力 uuid に関連付けられたイベントソースに追加します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

送信先を削除するには、destination-config パラメータの引数として空の文字列を指定します。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS および SQS の呼び出しレコードの例

以下の例は、Kafka イベントソース呼び出しが失敗した場合に Lambda が SNS トピックまたは SQS キューの送信先に送信する内容を示しています。recordsInfo の各キーには、Kafka トピックとパーティションの両方がハイフンで区切られて含まれています。例えば、キー "Topic-0" の場合、Topic は Kafka トピック、0 はパーティションです。各トピックとパーティションについて、オフセットとタイムスタンプデータを使用して元の呼び出しレコードを検索できます。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
```

```
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 送信先の呼び出しレコードの例

S3 の送信先の場合、Lambda は呼び出しレコード全体をメタデータと共に送信先に送信します。以下の例は、Kafka イベントソース呼び出しが失敗した場合に、Lambda が S3 バケットの送信先に送信することを示しています。SQS と SNS の送信先に関する前例のすべてのフィールドに加えて、payload フィールドには元の呼び出しレコードがエスケープされた JSON 文字列として含まれています。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
```

```
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 送信先バケットで S3 バージョニングを有効にすることをお勧めします。

Kafka クラスターをイベントソースとして使用する

Apache Kafka クラスターを Lambda 関数のトリガーとして追加すると、クラスターはイベントソースとして使用されます。

Lambda は、ユーザーが指定した `StartingPosition` に基づいて、CreateEventSourceMapping リクエストで `Topics` として指定された Kafka トピックからイベントデータを読み取ります。処理が成功すると、Kafka トピックは Kafka クラスターにコミットされます。

StartingPosition を LATEST として指定すると、Lambda は、そのトピックに属する各パーティション内の最新のメッセージから読み取りを開始します。トリガーが設定されてから Lambda がメッセージの読み取りを開始するまでには若干の遅延が発生することがあるため、Lambda はこの期間中に生成されたメッセージを読み取りません。

Lambda は、指定された 1 つ、または複数の Kafka トピックパーティションからのレコードを処理し、関数に JSON ペイロードを送信します。利用可能なレコードが増えると、Lambda は関数がトピックに追いつくまで、CreateEventSourceMapping リクエストで指定された BatchSize 値に基づいて、バッチ内のレコードの処理を継続します。

関数がバッチ内のいずれかのメッセージに対してエラーを返すと、Lambda は、処理が成功するかメッセージが期限切れになるまでメッセージのバッチ全体を再試行します。すべての再試行が失敗したレコードを、障害発生時の送信先に送信して、後で処理することができます。

> **Note**
>
> Lambda 関数の最大タイムアウト制限は通常 15 分ですが、Amazon MSK、自己管理型 Apache Kafka、Amazon DocumentDB、および ActiveMQ と RabbitMQ 向け Amazon MQ のイベントソースマッピングでは、最大タイムアウト制限が 14 分の関数のみがサポートされます。この制約により、イベントソースマッピングは関数エラーと再試行を適切に処理できます。

ポーリングとストリームの開始位置

イベントソースマッピングの作成時および更新時のストリームのポーリングは、最終的に一貫性があることに注意してください。

- イベントソースマッピングの作成時、ストリームからのイベントのポーリングが開始されるまでに数分かかる場合があります。
- イベントソースマッピングの更新時、ストリームからのイベントのポーリングが停止および再開されるまでに数分かかる場合があります。

つまり、LATEST をストリームの開始位置として指定すると、イベントソースマッピングの作成または更新中にイベントを見逃す可能性があります。イベントを見逃さないようにするには、ストリームの開始位置を TRIM_HORIZON または AT_TIMESTAMP として指定します。

Kafka イベントソースのオートスケーリング

初めて Apache Kafka イベントソースを作成するときは、Lambda が Kafka トピック内のすべてのパーティションを処理するために 1 つのコンシューマーを割り当てます。各コンシューマーには、増加したワークロードを処理するために同時実行される複数のプロセッサがあります。さらに、Lambda は、ワークロードに基づいてコンシューマーの数を自動的にスケールアップまたはスケールダウンします。各パーティションでメッセージの順序を保つため、コンシューマーの最大数は、トピック内のパーティションあたり 1 つとなっています。

Lambda は、1 分間隔でトピック内のすべてのパーティションのコンシューマーオフセット遅延を評価します。遅延が大きすぎる場合、パーティションは Lambda で処理可能な速度よりも速い速度でメッセージを受信します。必要に応じて、Lambda はトピックにコンシューマーを追加するか、またはトピックからコンシューマーを削除します。コンシューマーを追加または削除するスケーリングプロセスは、評価から 3 分以内に行われます。

ターゲットの Lambda 関数がオーバーロードすると、Lambda はコンシューマーの数を減らします。このアクションにより、コンシューマーが取得し関数に送信するメッセージの数が減り、関数への負荷が軽減されます。

Kafka トピックのスループットを監視するには、`consumer_lag` や `consumer_offset` などの Apache Kafka コンシューマーラグメトリックスを表示します。いくつの関数呼び出しが並行して発生しているかを確認するときは、関数の同時実行メトリクスも監視します。

イベントソースマッピングエラー

Apache Kafka クラスターを Lambda 関数のイベントソースとして追加すると、関数でエラーが発生した場合、Kafka コンシューマーはレコードの処理を停止します。トピックパーティションのコンシューマーは、レコードのサブスクライブ、読み取り、処理を行います。その他の Kafka コンシューマーは、同じエラーが発生しない限り、レコードの処理を続行できます。

停止したコンシューマの原因を特定するには、`StateTransitionReason` のレスポンスの `EventSourceMapping` フィールドを確認します。以下は、受け取る可能性があるイベントソースエラーを説明するリストです。

`ESM_CONFIG_NOT_VALID`

イベントソースマッピングの設定が無効です。

`EVENT_SOURCE_AUTHN_ERROR`

Lambda がイベントソースを認証できませんでした。

EVENT_SOURCE_AUTHZ_ERROR

Lambda にイベントソースへのアクセスに必要な許可がありません。

FUNCTION_CONFIG_NOT_VALID

関数の設定が無効です。

> **Note**
>
> Lambda のイベントレコードが許容サイズ制限である 6 MB を超えると、未処理になります。

Amazon CloudWatch メトリクス

Lambda は、関数がレコードを処理している間に `OffsetLag` メトリクスを発行します。このメトリクスの値は、Kafka イベントソーストピックに書き込まれた最後のレコードと関数のコンシューマグループが処理した最後のレコードの間のオフセットの差分です。レコードが追加されてからコンシューマグループがそれを処理するまでのレイテンシーを見積もるには、`OffsetLag` を使用できます。

`OffsetLag` の増加傾向は、関数のコンシューマグループに問題があることを示している可能性があります。詳細については、「Lambda 関数のメトリクスの使用」を参照してください。

セルフマネージド Apache Kafka の設定パラメータ

すべての Lambda イベントソースタイプは、同じ CreateEventSourceMapping および UpdateEventSourceMapping API オペレーションを共有しています。ただし、Apache Kafka に適用されるのは一部のパラメータのみです。

セルフマネージド型 Apache Kafka に適用されるイベントソースパラメータ

Parameter	必須	デフォルト	メモ
BatchSize	N	100	最大: 10,000
有効	N	有効	
FunctionName	Y		

Parameter	必須	デフォルト	メモ
FilterCriteria	N		Lambda のイベントフィルタリング
MaximumBatchingWindowInSeconds	N	500 ミリ秒	バッチ処理動作
SelfManagedEventSource	Y		Kafka ブローカーの一覧。作成時にのみ設定可能
SelfManagedKafkaEventSourceConfig	N	ConsumerGroupID フィールドを含み、デフォルトでは一意の値になっています。	作成時にのみ設定可能
SourceAccessConfigurations	N	認証情報なし	クラスターの VPC 情報または認証情報 SASL_PLAIN は、BASIC_AUTH に設定
StartingPosition	Y		AT_TIMESTAMP、TRIM_HORIZON、または LATEST 作成時にのみ設定可能
StartingPositionTimestamp	N		StartingPosition が AT_TIMESTAMP に設定されている場合にのみ必要

Parameter	必須	デフォルト	メモ
トピック	Y		トピック名 作成時にのみ設定可能

Amazon SQS での Lambda の使用

> **Note**
>
> Lambda 関数以外のターゲットにデータを送信したい、または送信する前にデータをエンリッチしたいという場合は、「Amazon EventBridge Pipes」を参照してください。

Amazon Simple Queue Service (Amazon SQS) キュー内のメッセージを処理するには、Lambda 関数を使用することができます。Lambda は、イベントソースマッピングで、標準キューとファーストイン、ファーストアウト (FIFO) キューの両方をサポートしています。

Amazon SQS イベントソースマッピングのポーリングとバッチ処理の動作を理解する

Amazon SQS イベントソースマッピングでは、Lambda はキューをポーリングし、イベントと共に関数を同期的に呼び出します。各イベントには、キューからの複数のメッセージのバッチを含めることができます。Lambda は、これらのイベントをバッチとして (一度に 1 バッチずつ) 受け取り、バッチごとに関数を 1 回呼び出します。関数が正常にバッチを処理すると、Lambda はキューからそのメッセージを削除します。

Lambda がバッチを受け取ると、メッセージはキューに留まりますが、キューの可視性タイムアウトの期間中は非表示になります。関数がバッチ内のすべてのメッセージを正常に処理すると、Lambda はそのメッセージをキューから削除します。デフォルトでは、バッチの処理中に関数でエラーが発生すると、可視性タイムアウトの期限が切れた後に、そのバッチ内のすべてのメッセージが再びキューに表示されます。このため、関数コードは、意図しない副次的影響を及ぼすことなく同じメッセージを複数回処理できるようにする必要があります。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるた

め、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

Lambda がメッセージを複数回処理しないようにするには、関数レスポンスにバッチアイテムの失敗を含めるようにイベントソースマッピングを設定するか、DeleteMessage API を使用して、Lambda 関数がメッセージを正常に処理した場合にそれらをキューから削除することができます。

Lambda が SQS イベントソースマッピングでサポートする設定パラメータの詳細については、「the section called “SQS イベントソースマッピングの作成”」を参照してください。

標準キューメッセージイベントの例

Example Amazon SQS メッセージイベント (標準キュー)

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
```

```
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

デフォルトでは、Lambda はキュー内の最大 10 個のメッセージを一度にポーリングし、そのバッチを関数に送信します。少数のレコードで関数が呼び出されることを回避するには、バッチウィンドウを設定することで、最大 5 分間レコードをバッファリングするようにイベントソースに指示できます。関数を呼び出す前に、Lambda は、バッチ処理ウィンドウの期限が切れる、呼び出しペイロードサイズのクォータに到達する、または設定された最大バッチサイズに到達するまで、標準キューからのメッセージのポーリングを継続します。

バッチウィンドウを使用していて、SQS キューのトラフィックがきわめて少ない場合、Lambda は関数を呼び出す前に最大 20 秒間待機することがあります。これは、バッチウィンドウを 20 秒未満に設定した場合であっても同様です。

> **Note**
>
> Java では、JSON を逆シリアル化するときに null ポインタエラーが発生することがあります。これは、「Records」と「eventSourceARN」のケースが JSON オブジェクトマッパーによってどのように変換されるかに起因している可能性があります。

FIFO キューメッセージイベントの例

FIFO キューの場合、レコードには、重複除外と順序付けに関連する追加属性が含まれます。

Example Amazon SQS メッセージイベント (FIFO キュー)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
```

```
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

Amazon SQS イベントソースマッピングの作成と管理

Lambda で Amazon SQS メッセージを処理するには、キューを適切に設定し、Lambda イベントソースマッピングを作成します。

Lambda で使用するキューの設定

既存の Amazon SQS キューがない場合は、Lambda 関数のイベントソースとして機能するキューを作成します。次に、Lambda 関数がイベントの各バッチを処理するのに十分な時間を確保できるようにキューを設定します。

関数がレコードの各バッチを処理する時間を確保するには、ソースキューの可視性タイムアウトを、関数の設定タイムアウトの少なくとも 6 倍に設定します。追加の時間は、関数が前のバッチの処理中にスロットリングされた場合に、Lambda が再試行することを可能にします。

デフォルトでは、Lambda がバッチを処理しているときにエラーが発生すると、そのバッチ内のすべてのメッセージがキューに戻ります。可視性タイムアウトの後、メッセージは再び Lambda に表示されるようになります。部分的なバッチレスポンスを使用するようにイベントソースマッピングを設定することで、失敗したメッセージのみがキューに戻るようにすることができます。さらに、関数が 1 つのメッセージの処理に複数回失敗する場合、Amazon SQS はそのメッセージをデッドレターキューに送信できます。ソースキューの再処理ポリシーで、maxReceiveCount を少なくとも 5 に設定することをお勧めします。これにより、Lambda は、失敗したメッセージをデッドレターキューに直接送信する前に再試行を数回行う機会を確保できます。

Lambda 実行ロールのアクセス許可の設定

AWSLambdaSQSQueueExecutionRole AWS マネージドポリシーには、Lambda が Amazon SQS キューから読み取るために必要なアクセス許可が含まれています。関数の実行ロールにこのマネージドポリシーを追加できます。

オプションで、暗号化されたキューを使用している場合は、実行ロールに次の権限を追加する必要もあります。

- kms:Decrypt

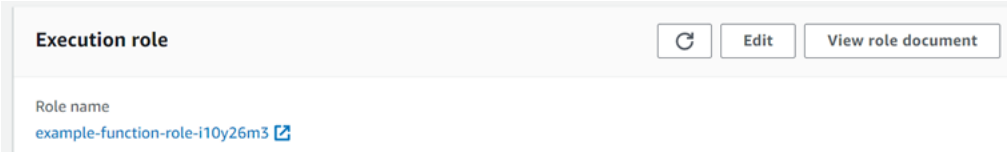
SQS イベントソースマッピングの作成

イベントソースマッピングを作成し、キューから Lambda 関数に項目を送信するように Lambda に通知します。1 つの関数で複数のキューの項目を処理するには、複数のイベントソースマッピングを作成します。Lambda がターゲットの関数を呼び出すと、このイベントには設定可能なバッチサイズまでの複数の項目が含まれている可能性があります。

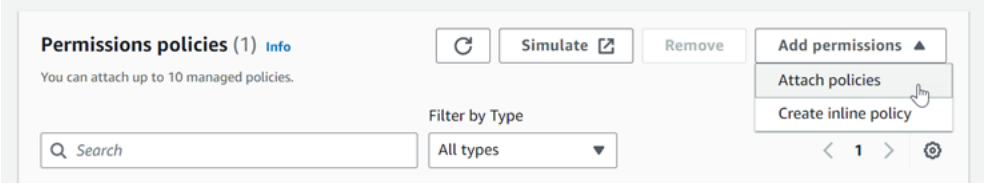
Amazon SQS から読み取るように関数を設定するには、AWSLambdaSQSQueueExecutionRole AWS マネージドポリシーを実行ロールにアタッチします。次に、以下の手順を使用して、コンソールから SQS イベントソースマッピングを作成します。

アクセス許可を追加してトリガーを作成するには

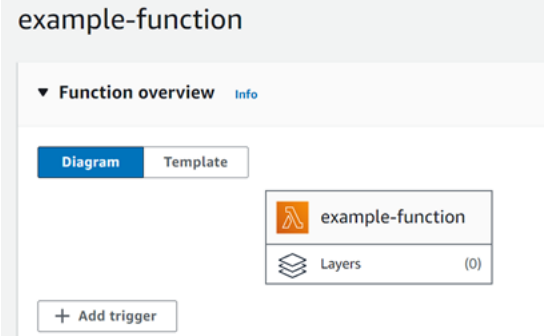
1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Configuration] (設定) タブを開き、次に [Permissions] (アクセス許可) をクリックします。
4. [実行ロール] で、実行ロールのリンクを選択します。このリンクを選択すると、IAM コンソールでロールが開きます。



5. [アクセス許可を追加]、[ポリシーをアタッチ] の順に選択します。



6. [検索] フィールドに AWSLambdaSQSQueueExecutionRole を入力します。実行ロールにポリシーを追加 これは、関数が Amazon SQS キューから読み取るために必要なアクセス許可を含む AWS 管理ポリシーです。このポリシーの詳細については、「AWS マネージドポリシーリファレンス」の「AWSLambdaSQSQueueExecutionRole」を参照してください。
7. Lambda コンソールの関数に戻ります。[関数の概要] で [トリガーを追加] をクリックします。



8. トリガーのタイプを選択します。
9. 必須のオプションを設定し、[Add] (追加) を選択します。

Lambda は、Amazon SQS イベントソースに対して以下の設定オプションをサポートしています。

SQS キュー

レコードの読み取り元である Amazon SQS キュー。

トリガーの有効化

イベントソースマッピングのステータス。[Enable trigger] (トリガーの有効化) はデフォルトで選択されています。

バッチサイズ

各バッチで関数に送信されるレコードの最大数。標準キューの場合、最大 10,000 レコードまで可能です。FIFO キューの場合、最大値は 10 です。バッチサイズが 10 を超える場合は、バッチウィンドウ (MaximumBatchingWindowInSeconds) も 1 秒以上に設定する必要があります。

関数のタイムアウトは、バッチの項目すべてを処理するために十分な時間を確保できるように設定します。項目の処理に長時間かかる場合には、より少ないバッチサイズを選択します。バッチサイズを大きくするとワークロードの効率を向上させることができ、非常に高速になるか、多くのコストがかかります。関数で予約された同時実行数を設定すると、同時実行数を 5 以上に設定した場合に、Lambda が関数を呼び出したときにスロットリングエラーが発生する可能性が少なくなります。

Lambda は、イベントの合計サイズが同期呼び出しの呼び出しペイロードサイズのクォータ (6 MB) を超えない限り、バッチ内のすべてのレコードを単一の呼び出しで関数に渡します。Lambda と Amazon SQS の両方が、レコードごとにメタデータを生成します。この追加のメタデータは合計ペイロードサイズに計上され、1 つのバッチで送信されるレコードの総数が設定されたバッチサイズよりも少なくなる可能性があります。Amazon SQS が送信するメタデータフィールドは可変長にすることができます。Amazon SQS メタデータフィールドの詳細については、「Amazon Simple Queue Service API リファレンス」の「ReceiveMessage」API 操作のドキュメントを参照してください。

バッチウィンドウ

関数を呼び出すまでのレコード収集の最大時間 (秒) です。これが適用されるのは標準キューのみです。

0 秒を超えるバッチウィンドウを使用している場合は、キューの可視性タイムアウトに処理時間の増加を考慮する必要があります。キューの可視性タイムアウトは、関数のタイムアウトの 6 倍に `MaximumBatchingWindowInSeconds` の値を加えた時間に設定することをお勧めします。これによりスロットリングエラーが発生した場合に Lambda 関数がイベントの各バッチを処理し、再試行する時間が許容されます。

メッセージが使用可能になると、Lambda はメッセージのバッチ処理を開始します。Lambda は、関数を 5 回同時に呼び出すことで、一度に 5 つのバッチの処理を開始します。メッセージがまだ利用可能な場合、Lambda は関数のインスタンスを 1 分あたり最大 300 インスタンスまで追加し、最大 1,000 インスタンスまで増やします。関数のスケーリングと同時実行の詳細について理解するには、「Lambda 関数のスケーリング」を参照してください。

より多くのメッセージを処理するには、Lambda 関数を最適化してスループットを向上させることができます。詳細については、「AWS Lambda が Amazon SQS 標準キューでどのようにスケールするかを理解する」を参照してください。

最大同時実行数

イベントソースが呼び出せる同時関数の最大数。詳細については、「Amazon SQS イベントソースの最大同時実行数の設定」を参照してください。

フィルター条件

フィルター条件を追加して、Lambda が処理のために関数に送信するイベントを制御します。詳細については、「Lambda のイベントフィルタリング」を参照してください。

SQS イベントソースマッピングのスケーリング動作の設定

標準キューの場合、Lambda はロングポーリングを使用して、キューがアクティブになるまでキューをポーリングします。メッセージが利用可能な場合、Lambda は、関数を 5 回同時に呼び出すことで、一度に 5 つのバッチの処理を開始します。メッセージがまだ利用可能な場合、Lambda はバッチを読み込むプロセスの数を 1 分あたり最大 300 インスタンスまで増やします。イベントソースマッピングによって同時に処理できるバッチの最大数は 1,000 です。

FIFO キューの場合、Lambda は、受信した順序でメッセージを関数に送信します。FIFO キューにメッセージを送信する場合、メッセージグループ ID を指定します。Amazon SQS は、同じグループ内のメッセージが Lambda に順番に配信されるようにします。Lambda がメッセージをバッチに読み込むとき、各バッチには複数のメッセージグループからのメッセージが含まれることがありますが、メッセージの順序は維持されます。関数がエラーを返す場合、その関数は、Lambda が同じグループから追加のメッセージを受信する前に、対象メッセージですべての再試行を試みます。

Amazon SQS イベントソースの最大同時実行数の設定

最大同時実行数の設定を使用して、SQS イベントソースのスケーリング動作を制御できます。最大同時実行数設定は、Amazon SQS イベントソースが呼び出せる関数の同時インスタンス数を制限します。最大同時実行数は、イベントソースレベルの設定です。1 つの関数に複数の Amazon SQS イベントソースをマップしている場合は、各イベントソースに個別の最大同時実行数を設定できます。最大同時実行数は、1 つのキューが関数の予約された同時実行のすべてを使用したり、アカウントの同時実行クォータの残りのすべてを使用したりしないようにするために使用できます。Amazon SQS イベントソースでの最大同時実行数の設定に料金はかかりません。

重要なのは、最大同時実行数と予約された同時実行は、2 つの独立した設定であるということです。最大同時実行数を、関数の予約された同時実行よりも多い数に設定しないでください。最大同時実行数を設定する場合は、関数の予約された同時実行数が、関数にマップされたすべての Amazon SQS イベントソースの合計最大同時実行数以上になるようにしてください。合計数未満になった場合は、Lambda がメッセージをスロットルする可能性があります。

最大同時実行数が設定されていない場合、Lambda は Amazon SQS イベントソースをアカウントの合計同時実行クォータ (デフォルトでは 1,000) までスケールできます。

> **Note**
>
> FIFO キューの場合、同時呼び出しの上限は、メッセージグループ ID の数 (`messageGroupId`) または最大同時実行数の設定 (どちらか小さい方) です。例えば、メッ

セージグループ ID が 6 つあり、最大同時実行数が 10 に設定されている場合、関数は最大 6 回の同時呼び出しを行うことができます。

新規および既存の Amazon SQS イベントソースマッピングに最大同時実行数を設定できます。

Lambda コンソールを使用して最大同時実行数を設定する

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Function overview] (関数の概要) で [SQS] を選択します。選択すると、[Configuration] (設定) タブが開きます。
4. Amazon SQS トリガーを選択し、[Edit] (編集) を選択します。
5. [Maximum concurrency] (最大同時実行数) には、2 から 1,000 までの数値を入力します。最大同時実行数をオフにするには、ボックスを空のままにします。
6. [保存] を選択します。

AWS Command Line Interface(AWS CLI) を使用して最大同時実行数を設定する

`--scaling-config` オプション付きの update-event-source-mapping コマンドを使用します。例：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

最大同時実行数をオフにするには、`--scaling-config` に空の値を入力します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Lambda API を使用して最大同時実行数を設定する

ScalingConfig オブジェクトを指定した CreateEventSourceMapping アクションまたは UpdateEventSourceMapping アクションを使用します。

Lambda での SQS イベントソースのエラーの処理

SQS イベントソースに関連するエラーを処理する際に、Lambda はバックオフ戦略を備えた再試行戦略を自動的に使用します。また、部分的なバッチレスポンスを返すように SQS イベントソースマッピングを設定することで、エラー処理の動作をカスタマイズすることもできます。

失敗した呼び出しに対するバックオフ戦略

呼び出しが失敗すると、Lambda はバックオフ戦略の実装中に呼び出しの再試行を試みます。バックオフ戦略は、Lambda で発生した障害が関数コード内のエラーによるものか、スロットリングによるものかに応じて若干異なります。

- 関数コードが原因でエラーが発生した場合、Lambda は処理を停止し、呼び出しを再試行します。その間、Lambda は Amazon SQS イベントソースマッピングに割り当てられた同時実行数を減らすことで、再試行を徐々にバックオフします。キューの可視性タイムアウトがタイムアウトすると、メッセージが再びキューに表示されます。
- スロットリングが原因で呼び出しが失敗する場合、Lambda は Amazon SQS イベントソースマッピングに割り当てられた同時実行数を減らすことで、再試行を徐々にバックオフします。Lambda は、メッセージのタイムスタンプがキューの可視性タイムアウトを超過するまでメッセージを再試行し続けますが、タイムアウトした時点でメッセージをドロップします。

部分的なバッチレスポンスの実装

Lambda 関数がバッチを処理しているときにエラーが発生すると、デフォルトでそのバッチ内のすべてのメッセージが再度キューに表示され、これには Lambda が正常に処理したメッセージも含まれます。その結果、関数が同じメッセージを複数回処理することになる場合があります。

失敗したバッチ内の正常に処理されたメッセージを再処理しないようにするために、失敗したメッセージのみを再び表示するようにイベントソースマッピングを設定できます。これを部分的なバッチレスポンスと呼びます。部分的なバッチレスポンスをオンにするには、イベントソースマッピングを設定するときに FunctionResponseTypes アクション用に `ReportBatchItemFailures` を指定します。そうすると、関数が部分的な成功を返すようになるため、レコードでの不必要な再試行回数を減らすことができます。

`ReportBatchItemFailures` がアクティブ化されている場合、Lambda は、関数の呼び出しが失敗したときにメッセージポーリングをスケールダウンしません。一部のメッセージが失敗することが想定され、それらの失敗によってメッセージの処理レートに影響が及ばないようにする場合は、`ReportBatchItemFailures` を使用します。

> **Note**
>
> 部分的なバッチレスポンスを使用する場合は、次の点に注意してください。
>
> - 関数が例外をスローする場合、バッチ全体が完全な失敗とみなされます。
> - この機能を FIFO キューで使用している場合、関数は最初の失敗後にメッセージの処理を停止し、batchItemFailures で失敗したメッセージと未処理のメッセージのすべてを返します。これは、キュー内のメッセージの順序を維持するのに役立ちます。

部分的なバッチレポートをアクティブ化するには

1. 部分的なバッチレスポンスを実装するためのベストプラクティスを確認します。
2. 次のコマンドを実行して、関数用に ReportBatchItemFailures をアクティブ化します。イベントソースマッピングの UUID を取得するには、list-event-source-mappings AWS CLI コマンドを実行します。

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. 関数コードを更新して、すべての例外をキャッチし、失敗したメッセージを batchItemFailures JSON レスポンスで返します。batchItemFailures レスポンスには、メッセージ ID のリストが itemIdentifier JSON 値として含まれている必要があります。

   例えば、メッセージ ID が id1、id2、id3、id4、および id5 である 5 つのメッセージのバッチがあるとします。関数は、id1、id3、id5 を正常に処理します。メッセージ id2 および id4 がキューで再び表示されるようにするには、関数が次のレスポンスを返す必要があります。

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
```

```
}
```

バッチで失敗したメッセージ ID のリストを返す関数コードの例を次に示します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
```

```
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "encoding/json"
```

```
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    const batchItemFailures = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

TypeScript を使用して Lambda で SQS バッチ項目の失敗を報告します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
```

```
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
```

```
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

失敗したイベントがキューに戻らない場合は、AWS ナレッジセンターの「Lambda 関数 SQS ReportBatchItemFailures をトラブルシューティングするにはどうすればよいですか?」を参照してください。

成功条件と失敗の条件

関数が以下のいずれかを返す場合、Lambda はバッチを完全な成功として扱います。

- 空の `batchItemFailures` リスト

- null の batchItemFailures リスト
- 空の EventResponse
- null の EventResponse

関数が以下のいずれかを返す場合、Lambda はバッチを完全な失敗として扱います。

- 無効な JSON レスポンス
- 空の文字列 itemIdentifier
- ヌル itemIdentifier
- 不正なキー名を持つ itemIdentifier
- 存在しないメッセージ ID を持つ itemIdentifier 値

CloudWatch メトリクス

関数がバッチ項目の失敗を正しく報告しているかどうかを判断するために、Amazon SQS メトリクスの NumberOfMessagesDeleted および ApproximateAgeOfOldestMessage を Amazon CloudWatch でモニタリングできます。

- NumberOfMessagesDeleted は、キューから削除されたメッセージの数を追跡します。これが 0 になるということは、関数レスポンスが失敗したメッセージを正しく返していないことを示唆しています。
- ApproximateAgeOfOldestMessage は、最も古いメッセージがキューに残っている期間を追跡します。このメトリクスの急激な増加は、関数が失敗したメッセージを正しく返していないことを示唆している可能性があります。

Amazon SQS イベントソースマッピング用の Lambda パラメータ

すべての Lambda イベントソースタイプは、同じCreateEventSourceMapping および UpdateEventSourceMapping API オペレーションを共有しています。ただし、Amazon SQS に適用されるのは一部のパラメータのみです。

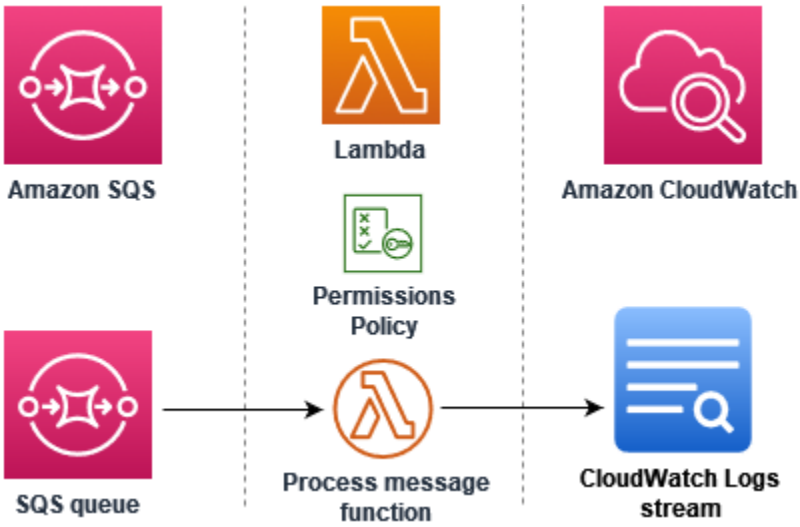
Amazon SQS に適用されるイベントソースパラメータ

[Parameter] (パラメータ)	必須	デフォルト	メモ
BatchSize	N	10	標準キューの場合、最大値は 10,000 です。FIFO キューの場合、最大値は 10 です。
有効	N	true	
EventSourceArn	Y		データストリームまたはストリーミングコンシューマーの ARN。
FunctionName	Y		
FilterCriteria	N		Lambda のイベントフィルタリング
FunctionResponseTypes	N		関数がバッチ内の特定の失敗を報告できるようにするには、`FunctionResponseTypes` に値 `ReportBatchItemFailures` を含めます。詳細については、部分的なバッチレスポンスの実装 を参照してください。
MaximumBatchingWindowInSeconds	N	0	

[Parameter] (パラメータ)	必須	デフォルト	メモ
ScalingConfig	N		Amazon SQS イベントソースの最大同時実行数の設定

チュートリアル: Amazon SQS での Lambda の使用

このチュートリアルでは、Amazon Simple Queue Service (Amazon SQS) キューからのメッセージを消費する Lambda 関数を作成します。Lambda 関数は、新しいメッセージがキューに追加されるたびに実行されます。この関数は、メッセージを Amazon CloudWatch Logs ストリームに書き込みます。次の図は、チュートリアルを完了するために使用する AWS リソースを示しています。



このチュートリアルを完了するには、次のステップを実行します。

1. CloudWatch Logs にメッセージを書き込む Lambda 関数を作成します。
2. Amazon SQS キューを作成します。
3. Lambda イベントソースマッピングを作成します。イベントソースマッピングは Amazon SQS キューを読み取り、新しいメッセージが追加されたときに Lambda 関数を呼び出します。
4. キューにメッセージを追加して設定をテストし、CloudWatch Logs で結果をモニタリングします。

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] を選んで、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理できます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウントのメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

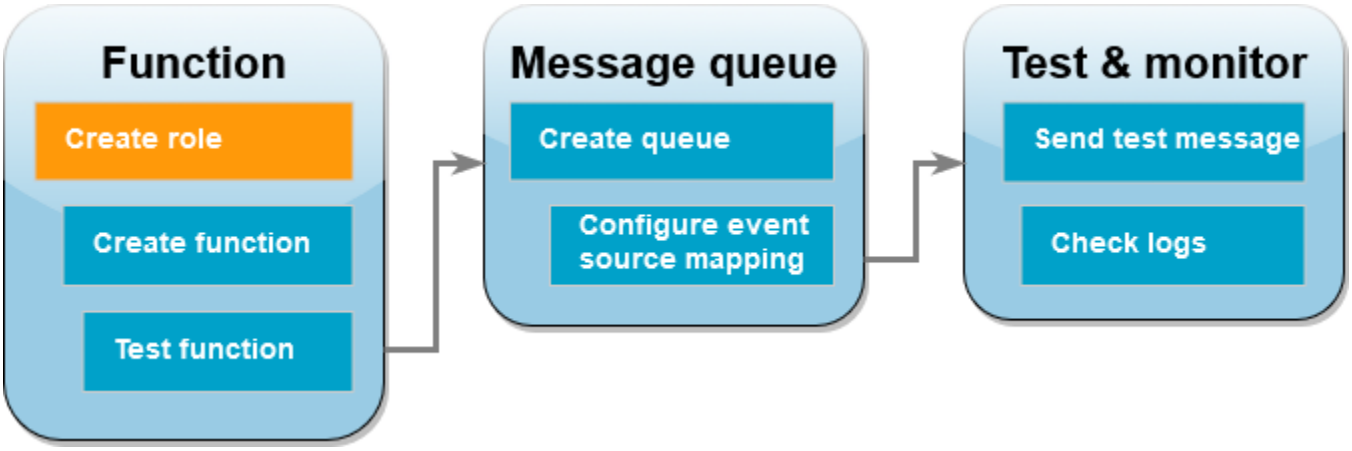
AWS Command Line Interface のインストール

AWS Command Line Interface をまだインストールしていない場合は、「最新バージョンの AWS CLI のインストールまたは更新」にある手順に従ってインストールしてください。

このチュートリアルでは、コマンドを実行するためのコマンドラインターミナルまたはシェルが必要です。Linux および macOS では、任意のシェルとパッケージマネージャーを使用してください。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (`zip` など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。

実行ロールを作成する



実行ロールとは、AWS サービスとリソースにアクセスする許可を Lambda 関数に付与する AWS Identity and Access Management (IAM) ロールです。関数が Amazon SQS から項目を読み取れるようにするには、AWSLambdaSQSQueueExecutionRole 許可ポリシーをアタッチします。

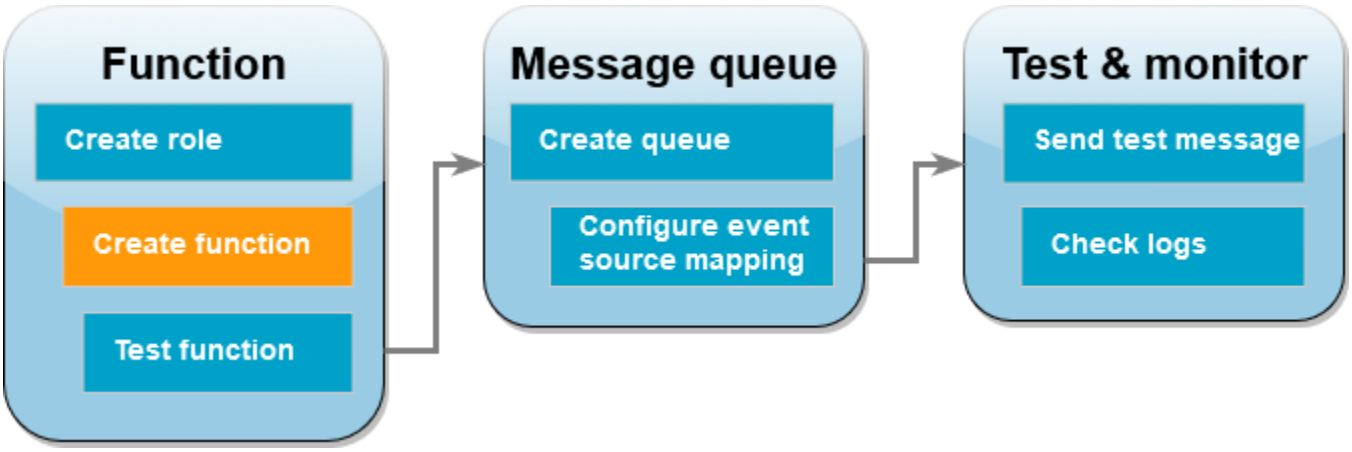
実行ロールを作成して Amazon SQS 許可ポリシーをアタッチする方法

1. IAM コンソールのロールページを開きます。
2. [ロールの作成] を選択します。
3. [信頼できるエンティティタイプ] で、[AWS サービス] を選択します。
4. [ユースケース] で、[Lambda] を選択します。

5. [Next] を選択します。
6. [許可ポリシー] 検索ボックスに **AWSLambdaSQSQueueExecutionRole** と入力します。
7. AWSLambdaSQSQueueExecutionRole ポリシーを選択し、[Next] を選択します。
8. [Role details] で [Role name] に **lambda-sqs-role** を入力してから、[Create role] を選択します。

ロールを作成したら、実行ロールの Amazon リソースネーム (ARN) を書き留めてください。これは、後のステップで必要になります。

関数を作成する



Amazon SQS メッセージを処理する Lambda 関数を作成します。この関数コードは、Amazon SQS メッセージの本文を CloudWatch Logs に記録します。

このチュートリアルでは Node.js 18.x ランタイムを使用しますが、他のランタイム言語のサンプルコードも提供しています。次のボックスでタブを選択すると、関心のあるランタイムのコードが表示されます。このステップで使用する JavaScript コードは、[JavaScript] タブに表示されている最初のサンプルにあります。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
```

```
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}
```

```
func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
```

```
            throw e;
        }

    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

TypeScript を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での SQS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SQS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で SQS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
```

```
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Node.js Lambda 関数を作成する方法

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir sqs-tutorial
cd sqs-tutorial
```

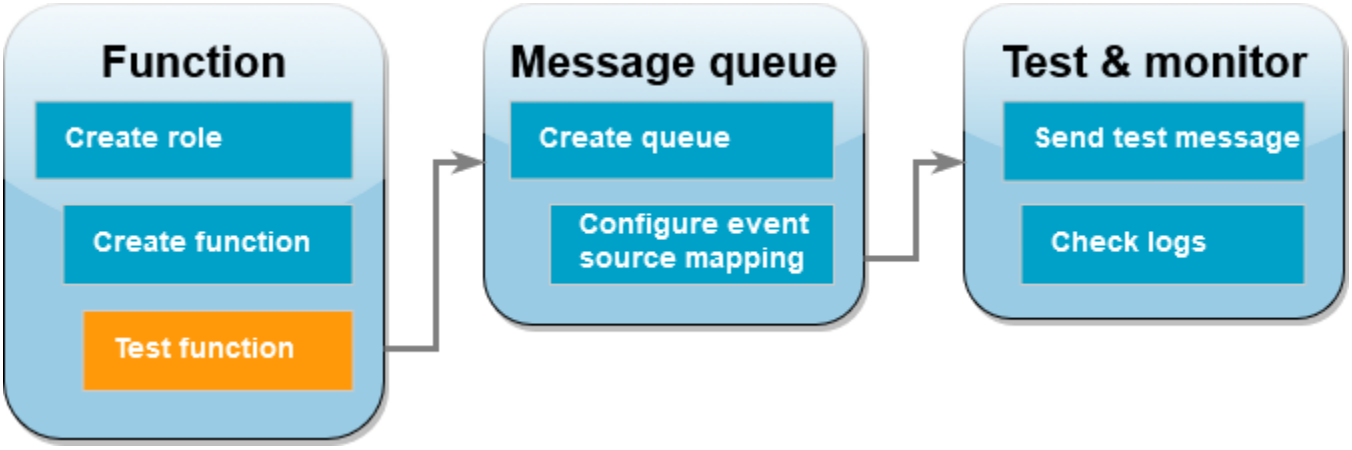
2. サンプル JavaScript コードを `index.js` という名前の新しいファイルにコピーします。
3. 以下の `zip` コマンドを使用して、デプロイパッケージを作成します。

```
zip function.zip index.js
```

4. create-function AWS CLI コマンドを使用して、Lambda 関数を作成します。`role`パラメータには、前に作成した実行ロールの ARN を入力します。

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

関数をテストする



invoke AWS CLI コマンドおよびサンプルの Amazon SQS イベントを使用して、手動で Lambda 関数を呼び出します。

サンプルイベントで Lambda 関数を呼び出す方法

1. 次の JSON をファイル名 input.json で保存します。この JSON は、Amazon SQS が Lambda 関数に送信する可能性のあるイベントをシミュレートするもので、"body" にはキューからの実際のメッセージが含まれます。この例では、メッセージは "test" です。

   Example Amazon SQS イベント

   これはテストイベントです。メッセージやアカウント番号を変更する必要はありません。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
```

```
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. 次の invoke AWS CLI コマンドを実行します。このコマンドは、レスポンスで CloudWatch ログを返します。ログの取得の詳細については、「AWS CLI を使用したログへのアクセス」を参照してください。

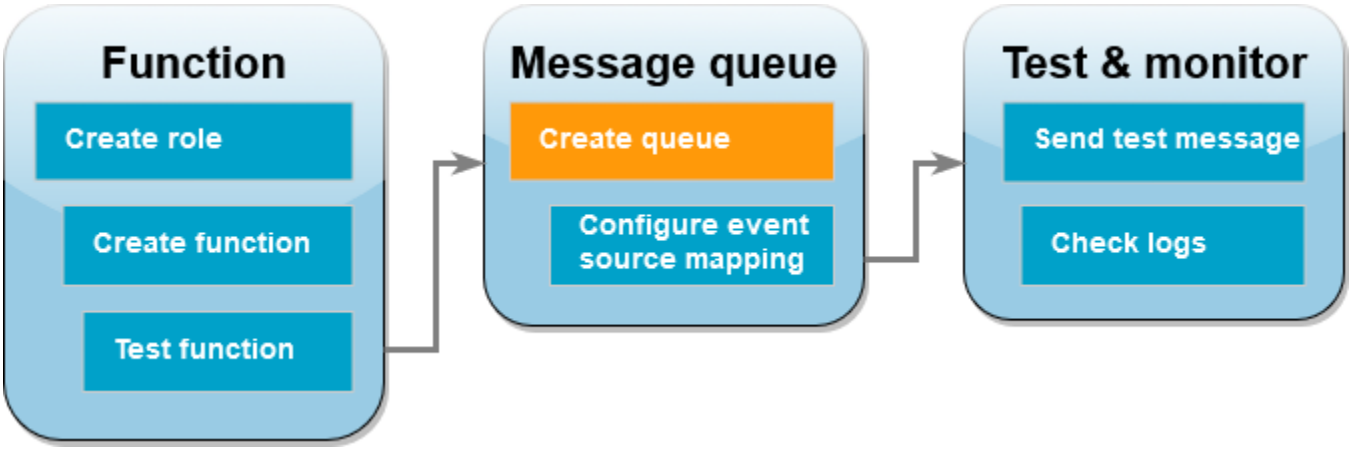
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

3. レスポンス内にある INFO ログを探します。このログは Lambda 関数がメッセージ本文を記録する場所です。次のようなログが表示されます。

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

Amazon SQS キュー を作成する



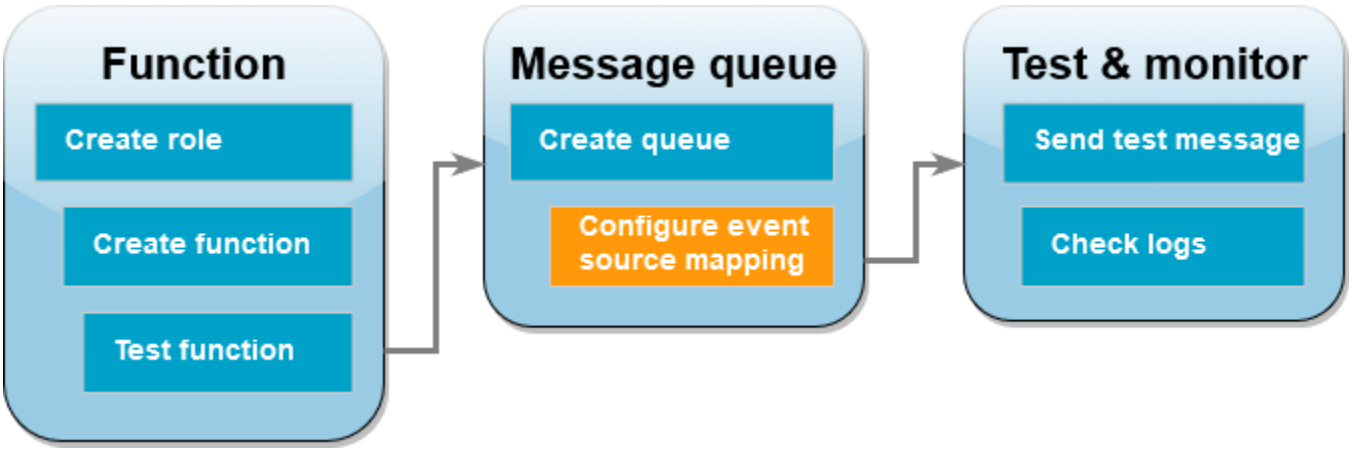
Lambda 関数がイベントソースとして使用できる Amazon SQS キューを作成します。

キューを作成するには

1. Amazon SQS コンソール を開きます。
2. [キューの作成] を選択します。
3. キューの名前を入力します。その他のオプションはすべて、デフォルト設定のままにしておきます。
4. [キューの作成]を選択します。

キューを作成したら、その ARN を書き留めます。こちらは、次のセクションでキューを Lambda 関数と関連付ける際に必要になります。

イベントソースを設定する



イベントソースマッピングを作成して、Amazon SQS キューを Lambda 関数に接続します。イベントソースマッピングは Amazon SQS キューを読み取り、新しいメッセージが追加されたときに Lambda 関数を呼び出します。

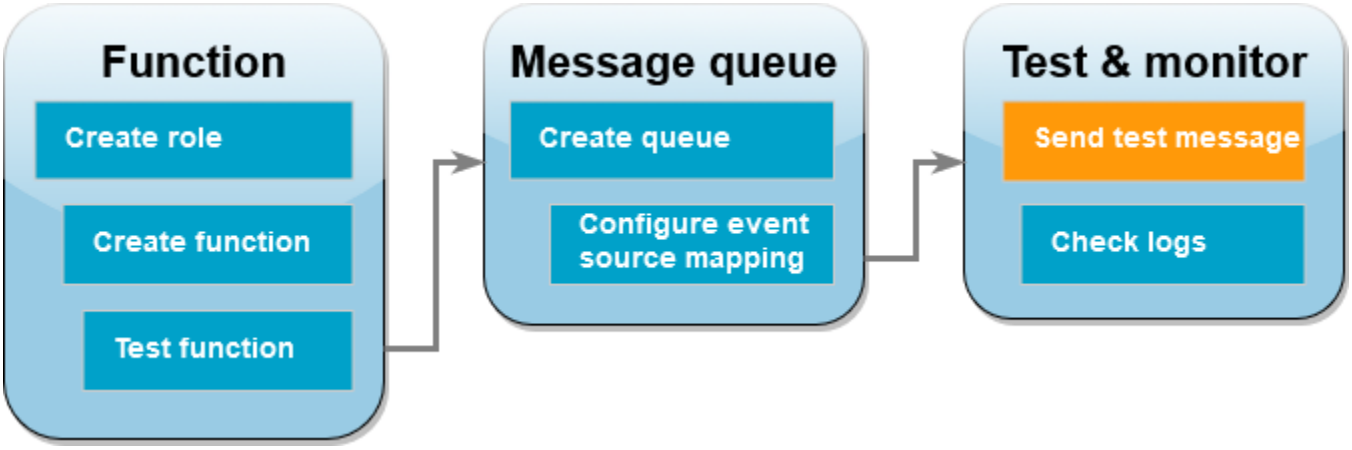
Amazon SQS キューと Lambda 関数の間でマッピングを作成するには、create-event-source-mapping AWS CLI コマンドを使用します。例：

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

イベントソースマッピングのリストを取得するには、list-event-source-mappings コマンドを使用します。例：

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

テストメッセージを送信する

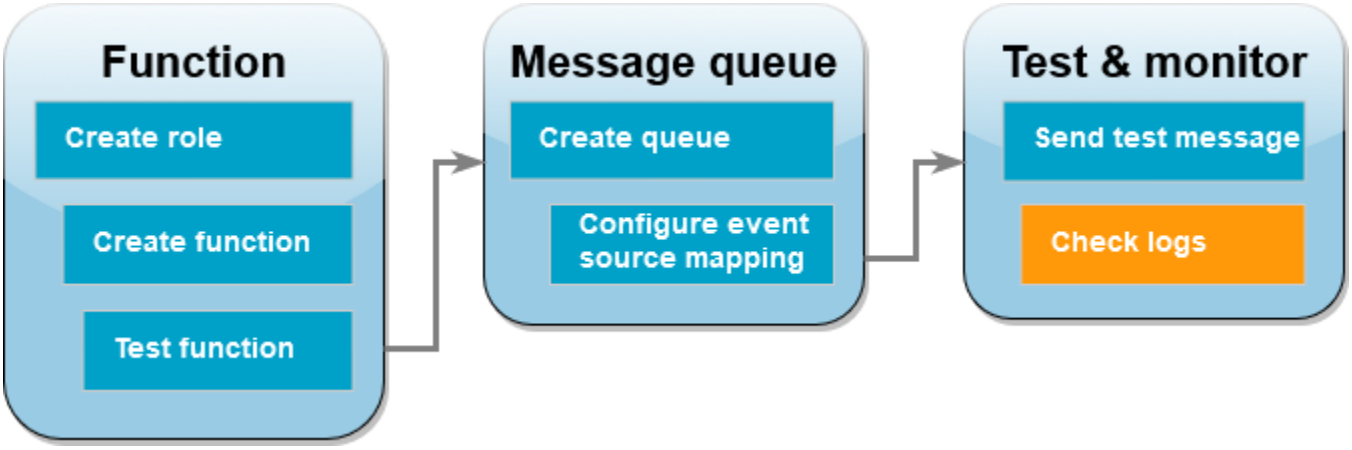


Amazon SQS メッセージを Lambda 関数に送信する方法

1. Amazon SQS コンソール を開きます。
2. 先ほど作成したキューを選択します。
3. [メッセージの送信と受信] を選択します。
4. メッセージ本文に、「これはテストメッセージです」などとテストメッセージを入力します。
5. [メッセージの送信] を選択します。

Lambdaがキューにアップデートをポーリングします。新しいメッセージがあると、Lambda はキューからのこの新しいイベントデータを使用して関数を呼び出します。　　関数ハンドラーが例外をスローせずに正常に戻った場合、Lambda はメッセージが正しく処理されたと見なし、キュー内の新しいメッセージの読み取りを開始します。メッセージが正常に処理された後、Lambdaはメッセージをキューから自動的に削除します。ハンドラーが例外をスローした場合、Lambda はメッセージの バッチが正常に処理されなかったと見なし、Lambdaは同じメッセージのバッチで関数を呼び出します。

CloudWatch のログを確認する



関数がメッセージを処理したことを確認する方法

1. Lambda コンソールの [関数ページ] を開きます。
2. ProcessSQSRecord 関数を選択します。
3. [モニター] を選択します。
4. [CloudWatch Logs を表示] を選択します。
5. CloudWatch コンソールで、関数のログストリームを選択します。
6. INFO ログを探します。このログは Lambda 関数がメッセージ本文を記録する場所です。Amazon SQS キューから送信したメッセージが表示されるはずです。例：

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。

4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[Delete] (削除) を選択します。

Amazon SQS キューを削除するには

1. AWS Management Console にサインインし、Amazon SQS コンソール (https://console.aws.amazon.com/sqs/) を開きます。
2. 作成したキューを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドに **confirm** を入力します。
5. [削除] を選択します。

チュートリアル: クロスアカウント Amazon SQS キューをイベントソースとして使用する

このチュートリアルでは、別のAWSアカウントで、 Amazon Simple Queue Service (Amazon SQS) キューからのメッセージを使用する Lambda 関数を作成します。　このチュートリアルには 2 つのAWSアカウントが含まれています: アカウント A Lambda 関数を含むアカウントを参照します。アカウント B Amazon SQS キューを含むアカウントを参照します。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する (アカウント A)

アカウント A で、必要なAWSリソースにアクセスするためのアクセス許可を関数に付与する実行ロールを作成します。

実行ロールを作成するには

1. AWS Identity and Access Management (IAM) コンソールの [Roles page (ロールページ)] を開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。
   - 信頼されたエンティティ - AWS Lambda
   - アクセス許可 - AWSLambdaSQSQueueExecutionRole
   - [ロール名] – **`cross-account-lambda-sqs-role`**

AWSLambdaSQSQueueExecutionRole ポリシーには、Amazon SQS から項目を読み取り、Amazon CloudWatch Logs にログを書き込むために関数が必要とするアクセス許可があります。

関数を作成する (アカウント A)

[アカウント A] で、Amazon SQS メッセージを処理する Lambda 関数を作成します。この例では、Node.js 18 コードが各メッセージを CloudWatch Logs のログに書き込みます。

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

関数を作成するには

> **Note**
>
> 以下の手順を実行すると、Node.js 18 で関数が作成されます。他の言語では、手順は似ていますが、いくつかの詳細が異なります。

1. サンプルコードをファイル名 `index.mjs` で保存します。
2. デプロイパッケージを作成します。

   ```
   zip function.zip index.mjs
   ```

3. `create-function` AWS Command Line Interface (AWS CLI) コマンドを使用して関数を作成します。

   ```
   aws lambda create-function --function-name CrossAccountSQSExample \
   --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
   --role arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role
   ```

関数をテストする (アカウント A)

アカウント A で、`invoke` AWS CLIコマンドおよびサンプルの Amazon SQS イベントを使用して、手動で Lambda 関数をテストします。

ハンドラーが例外をスローせずに正常に戻る場合、Lambda はメッセージが正しく処理されたと見なし、キュー内の新しいメッセージの読み取りを開始します。メッセージが正常に処理された後、Lambdaはメッセージをキューから自動的に削除します。ハンドラーが例外をスローした場合、Lambda はメッセージの バッチが正常に処理されなかったと見なし、Lambdaは同じメッセージのバッチで関数を呼び出します。

1. 次の JSON をファイル名 `input.txt` で保存します。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

"body"にキューからの実際のメッセージが含まれている場合、先行する JSON は、Amazon SQS が Lambda 関数に送信する可能性のあるイベントをシミュレートします。

2. 次の `invoke` AWS CLI コマンドを実行します。

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザー

ガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

3. outputfile.txt ファイルで出力を確認します。

Amazon SQS キュー を作成する (アカウント B)

アカウント B で、アカウント A の Lambda 関数がイベントソースとして使用できる Amazon SQS キューを作成します。

キューを作成するには

1. Amazon SQS コンソール を開きます。
2. [キューの作成] を選択します。
3. 次のプロパティでキューを作成します。
   - タイプ – スタンダード
   - 名前 – LambdaCrossAccountQueue
   - 設定 – デフォルト設定のままにします。
   - アクセスポリシー – [Advanced (アドバンスト)] を選択します。次の JSON ポリシーをペーストします:

```
{
   "Version": "2012-10-17",
   "Id": "Queue1_Policy_UUID",
   "Statement": [{
      "Sid":"Queue1_AllActions",
      "Effect": "Allow",
      "Principal": {
         "AWS": [
            "arn:aws:iam::<AccountA_ID>:role/cross-account-lambda-sqs-role"
         ]
      },
      "Action": "sqs:*",
      "Resource": "arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue"
    }
  ]
}
```

このポリシーは、アカウント A で Lambda 実行ロールに、この Amazon SQS キューからのメッセージを使用するアクセス許可を付与します。

4. キューの作成後、Amazon リソースネーム (ARN)を記録します。こちらは、次のセクションでキューを Lambda 関数と関連付ける際に必要になります。

イベントソースを設定する (アカウント A)

[アカウント A] で、[アカウント B] の Amazon SQS キューと Lambda 関数の間に、次の `create-event-source-mapping` AWS CLIコマンドを実行してイベントソースマッピングを作成します。

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-size 10 \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

次のコマンドを実行して、イベントソースのマッピングのリストを取得できます。

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

セットアップをテストする

これにより、次のようにセットアップをテストできます。

1. アカウント B で Amazon SQS コンソールを開きます。
2. 前に作成した [LambdaCrossAccountQueue] を選択します。
3. [メッセージの送信と受信] を選択します。
4. [メッセージ本文] にテストメッセージを入力します。
5. [メッセージの送信] を選択します。

[アカウント A] の Lambda 関数がメッセージを受信します。Lambda はキューに更新をポーリングし続けます。新しいメッセージがあると、Lambda はキューからのこの新しいイベントデータを使用して関数を呼び出します。 関数が実行され、Amazon CloudWatch にログが作成されます。CloudWatch コンソールでログを表示できます。

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

[アカウント A] で、実行ロールと Lambda 関数をクリーンアップします。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **`delete`** と入力し、[削除] を選択します。

[アカウント B] で、Amazon SQS キューをクリーンアップします。

Amazon SQS キューを削除するには

1. AWS Management Console にサインインし、Amazon SQS コンソール (https://console.aws.amazon.com/sqs/) を開きます。
2. 作成したキューを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドに **`confirm`** を入力します。
5. [削除] を選択します。

Lambda を使用した Amazon DocumentDB イベントの処理

Amazon DocumentDB クラスターをイベントソースとして設定することにより、Lambda 関数を使用して Amazon DocumentDB (MongoDB 互換) 変更ストリームでイベントを処理できます。その後、Amazon DocumentDB クラスターでデータが変更されるたびに Lambda 関数を呼び出すことで、イベント駆動型のワークロードを自動化できます。

> **Note**
>
> Lambda では Amazon DocumentDB のバージョン 4.0 および 5.0 のみがサポートされています。バージョン 3.6 はサポートされていません。
> また、イベントソースマッピングでは、Lambda はインスタンスベースのクラスターとリージョンレベルのクラスターのみをサポートします。Lambda は、Elastic クラスターまたはグローバルクラスターをサポートしていません。この制限は、Lambda を Amazon DocumentDB に接続するクライアントとして使用する場合には適用されません。Lambda はすべてのクラスタータイプに接続して CRUD 操作を実行できます。

Lambda は、Amazon DocumentDB 変更ストリームからのイベントを、到着した順序に沿って処理します。このため、関数は DocumentDB からの同時呼び出しを一度に 1 つしか処理できません。関数を監視するには、その同時実行メトリクスを追跡できます。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

トピック

- ポーリングとストリームの開始位置
- Amazon DocumentDB イベントソースのモニタリング
- チュートリアル: Amazon DocumentDB を用いて AWS Lambda のストリームの使用

Amazon DocumentDB イベントの例


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
    ],
    "eventSource": "aws:docdb"
```

```
}
```

この例のイベントとその形状の詳細については、MongoDB ドキュメントウェブサイトの「変更イベント」を参照してください。

前提条件とアクセス許可

Amazon DocumentDB を Lambda 関数のイベントソースとして使用する前に、次の前提条件に注意してください。必要なもの:

- 関数と同じ AWS アカウント と AWS リージョン に既存の Amazon DocumentDB クラスターが必要です。既存のクラスターがない場合は、Amazon DocumentDB デベロッパーガイドの「Amazon DocumentDB の開始方法」のステップに従って作成できます。または、チュートリアル: Amazon DocumentDB を用いて AWS Lambda のストリームの使用 ガイドの最初の一連のステップに従って、必要な前提条件をすべて備えた DocumentDB クラスターを作成する方法もあります。
- Lambda が、Amazon DocumentDB クラスターに関連付けられている Amazon Virtual Private Cloud (Amazon VPC) リソースにアクセスできるようにします。詳細については、「ネットワーク構成」を参照してください。
- Amazon DocumentDB クラスターで TLS を有効にします。これはデフォルトの設定です。TLS を無効にすると、Lambda はクラスターと通信できません。
- Amazon DocumentDB クラスターで変更ストリームをアクティブ化します。詳細については、「Amazon DocumentDB デベロッパーガイド」の「Amazon DocumentDB で変更ストリームを使用する」を参照してください。
- Amazon DocumentDB クラスターにアクセスするための認証情報を Lambda に提供します。イベントソースを設定するときは、クラスターへのアクセスに必要な認証の詳細 (ユーザー名とパスワード) を含む AWS Secrets Manager キーを指定します。セットアップ中にこのキーを指定するには、次のいずれかを行います。
  - セットアップに Lambda コンソールを使用している場合は、[Secrets Manager キー] フィールドでこのキーを指定します。
  - セットアップに AWS Command Line Interface (AWS CLI) を使用している場合は、`source-access-configurations` オプションでこのキーを指定します。このオプションは、`create-event-source-mapping` コマンドまたは `update-event-source-mapping` コマンドのどちらにも含めることができます。例:

    ```
    aws lambda create-event-source-mapping \
        ...
    ```

```
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ...
```

- Amazon DocumentDB ストリームに関連するリソースを管理するには、Lambda に許可を付与します。関数の実行ロールに次の許可を手動で追加します。
  - rds:DescribeDBClusters
  - rds:DescribeDBClusterParameters
  - rds:DescribeDBSubnetGroups
  - ec2:CreateNetworkInterface
  - ec2:DescribeNetworkInterfaces
  - ec2:DescribeVpcs
  - ec2:DeleteNetworkInterface
  - ec2:DescribeSubnets
  - ec2:DescribeSecurityGroups
  - kms:Decrypt
  - secretsmanager:GetSecretValue
- Lambda に送信する Amazon DocumentDB 変更ストリームイベントのサイズは 6 MB 未満にしてください。Lambda は最大 6 MB のペイロードサイズをサポートします。変更ストリームが 6 MB を超えるイベントを Lambda に送信しようとすると、Lambda はメッセージを削除して `OversizedRecordCount` メトリクスを発行します。Lambda は、ベストエフォートベースですべてのメトリクスを発行します。

> **Note**
>
> Lambda 関数の最大タイムアウト制限は通常 15 分ですが、Amazon MSK、自己管理型 Apache Kafka、Amazon DocumentDB、および ActiveMQ と RabbitMQ 向け Amazon MQ のイベントソースマッピングでは、最大タイムアウト制限が 14 分の関数のみがサポートされます。この制約により、イベントソースマッピングは関数エラーと再試行を適切に処理できます。

ネットワーク構成

Lambda が Amazon DocumentDB クラスターをイベントソースとして使用するには、クラスターが配置されている Amazon VPC にアクセスする必要があります。VCP にアクセスするには、Lambda に AWS PrivateLink 「VPC エンドポイント」をデプロイすることをお勧めします。Lambda の VPC エンドポイントをデプロイします。クラスターが認証を使用する場合、Secrets Manager の VPC エンドポイントもデプロイします。

または、Amazon DocumentDB クラスターに関連付けられた VPC で、パブリックサブネットごとに 1 つの NAT ゲートウェイが含まれていることを確認します。詳細については、「the section called "VPC 関数のインターネットアクセス"」を参照してください。

VPC エンドポイントを使用する場合は、プライベート DNS 名を有効にするように設定する必要もあります。

Amazon DocumentDB クラスターのイベントソースマッピングを作成するとき、Lambda はクラスターの VPC のサブネットおよびセキュリティグループに Elastic Network Interface (ENI) が既に存在するかどうかを確認します。Lambda が既存の ENI を検出した場合、再利用しようとします。それ以外の場合、Lambda は新しい ENI を作成し、イベントソースに接続して関数を呼び出します。

> **Note**
>
> Lambda 関数は、Lambda サービスが所有する VPC 内で常に実行されます。これらの VPC はサービスによって自動的に管理され、顧客には表示されません。関数を Amazon VPC に接続することもできます。いずれの場合、関数の VPC 設定はイベントソースマッピングに影響しません。Lambda がイベントソースに接続する方法を判定するのは、イベントソースの VPC の設定のみです。

VPC セキュリティグループのルール

次のルール (最低限) に従ってクラスターを含む Amazon VPC のセキュリティグループを設定してください。

- インバウンドルール – イベントソースに指定されたセキュリティグループに対し、Amazon DocumentDB クラスターポートにすべてのトラフィックを許可します。Amazon DocumentDB はデフォルトでポート 27017 を使用します。

- アウトバウンドルール – すべての送信先に対して、ポート 443 上のすべてのトラフィックを許可します。Amazon DocumentDB クラスターポートですべてのトラフィックを許可します。Amazon DocumentDB はデフォルトでポート 27017 を使用します。
- NAT ゲートウェイの代わりに VPC エンドポイントを使用している場合は、その VPC エンドポイントに関連付けられたセキュリティグループが、イベントソースのセキュリティグループからのポート 443 上のすべてのインバウンドトラフィックを許可する必要があります。

VPCエンドポイントの使用

VPC エンドポイントを使用するとき、ENI を使用し、関数を呼び出す API コールはこれらのエンドポイントを経由してルーティングされます。Lambda サービスプリンシパルは、これらの ENI を使用するすべての関数に `lambda:InvokeFunction` を呼び出す必要があります。

デフォルトでは、VPC エンドポイントはオープンな IAM ポリシーがあります。特定のプリンシパルのみがそのエンドポイントを使用して必要なアクションを実行するため、これらのポリシーを制限することがベストプラクティスです。イベントソースマッピングが Lambda 関数を呼び出せるようにするには、VPC エンドポイントポリシーは Lambda サービスプリンシパルが `lambda:InvokeFunction` を呼び出せるようにする必要があります。組織内で発生する API コールのみを許可するように VPC エンドポイントポリシーを制限すると、イベントソースマッピングが正しく機能しなくなります。

次の VPC エンドポイントポリシーの例では、Lambda エンドポイントに必要なアクセスを付与する方法を示しています。

Example VPC エンドポイントポリシー – Lambda エンドポイント

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Amazon DocumentDB クラスターが認証を使用する場合、Secrets Manager エンドポイントの VPC エンドポイントポリシーを制限することもできます。Secrets Manager API を呼び出す場合、Lambda は Lambda サービスプリンシパルではなく、関数ロールを使用します。次の例では、Secrets Manager エンドポイントポリシーを示します。

Example VPC エンドポイントポリシー - Secrets Manager エンドポイント

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

Amazon DocumentDB イベントソースマッピングを作成する (コンソール)

Amazon DocumentDB クラスターの変更ストリームから読み取る Lambda 関数に対して、イベントソースマッピングを作成します。このセクションでは、Lambda コンソールからこれを実行する方法を説明します。AWSSDK と AWS CLI 手順については、「the section called “Amazon DocumentDB イベントソースマッピングを作成する (SDK または CLI)”」を参照してください。

Amazon DocumentDB イベントソースマッピングを作成するには (コンソール)

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [関数の概要] で [トリガーを追加] をクリックします。
4. [トリガーの設定] のドロップダウンリストから [DocumentDB] を選択します。
5. 必要なオプションを設定し、[追加] を選択します。

Lambda は、Amazon DocumentDB イベントソースの以下のオプションをサポートしています。

- [DocumentDB クラスター] – Amazon DocumentDB クラスターを選択します。

- [トリガーをアクティブ化] – トリガーを今すぐアクティブ化するかどうかを選択します。このチェックボックスをオンにすると、関数は、イベントソースマッピングの作成時に、指定された Amazon DocumentDB 変更ストリームからのトラフィックの受信を直ちに開始します。チェックボックスをオフにして、テストのために非アクティブ化された状態でイベントソースマッピングを作成することを推奨します。作成後、いつでもイベントソースマッピングをアクティブ化できます。
- [データベース名] – 使用するクラスター内のデータベースの名前を入力します。
- (オプション) [コレクション名] – 使用するデータベース内のコレクションの名前を入力します。コレクションを指定しない場合、Lambda はデータベース内の各コレクションのすべてのイベントをリッスンします。
- [バッチサイズ] - 単一のバッチで取得するメッセージの最大数 (最大 10,000 件) を設定します。デフォルトバッチサイズは 100 です。
- [開始位置] – レコードの読み取りを開始するストリーム内の位置を選択します。
  - [最新] - ストリームに追加された新しいレコードのみを処理します。関数は、Lambda がイベントソースの作成を完了した後にのみ、レコードの処理を開始します。これは、イベントソースが正常に作成されるまで、一部のレコードが削除される可能性があることを意味します。
  - [水平トリム] - ストリーム内のすべてのレコードを処理します。Lambda は、クラスターの log 保持期間を使用して、イベントの読み取りを開始する場所を決定します。具体的には、Lambda は `current_time - log_retention_duration` から読み取りを開始します。Lambda がすべてのイベントを読み取るには、このタイムスタンプの前に変更ストリームが既にアクティブになっている必要があります。
  - [タイムスタンプ] - 特定の時刻以降のレコードを処理します。Lambda がすべてのイベントを適切に読み取るには、指定されたタイムスタンプの前に変更ストリームが既にアクティブになっている必要があります。
- [認証] – クラスター内のブローカーにアクセスするための認証方法を選択します。
  - [BASIC_AUTH] – 基本認証では、クラスターにアクセスするための認証情報を含む Secrets Manager キーを指定する必要があります。
- [Secrets Manager キー] – Amazon DocumentDB クラスターへのアクセスに必要な認証の詳細 (ユーザー名とパスワード) を含む Secrets Manager キーを選択します。
- (オプション) [バッチウィンドウ] - 関数を呼び出す前にレコードを収集する最大時間を 300 までの秒数で設定します。
- (オプション) [ドキュメントの完全な設定] – ドキュメントの更新オペレーションでは、ストリームに送信するものを選択します。デフォルト値は `Default` です。これは、各変更ストリームイベントについて、行われた変更について記述するデルタのみを Amazon DocumentDB が送信するこ

とを意味します。このフィールドの詳細については、MongoDB Javadocs の API ドキュメントの「FullDocument」を参照してください。

- [デフォルト] – Lambda は、行われた変更について記述する部分的なドキュメントのみを送信します。
- [UpdateLookup] – Lambda は、ドキュメント全体のコピーとともに、変更について記述するデルタを送信します。

Amazon DocumentDB イベントソースマッピングを作成する (SDK または CLI)

Amazon DocumentDB イベントソースマッピングを AWS SDK を使用して作成または管理するには、次の API オペレーションを使用します。

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

AWS CLI を使用してイベントソースマッピングを作成するには、`create-event-source-mapping` コマンドを使用します。以下の例では、このコマンドを使用して、`my-function` という名前の関数を Amazon DocumentDB 変更ストリームにマッピングします。イベントソースは Amazon リソースネーム (ARN) によって指定され、バッチサイズ 500 で、Unix 時間形式のタイムスタンプから始まります。このコマンドでは、Lambda が Amazon DocumentDB への接続に使用する Secrets Manager キーも指定します。さらに、データベースと読み取り元のコレクションを指定する `document-db-event-source-config` パラメーターも含まれています。

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

次のような出力が表示されます。

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

作成後、update-event-source-mapping コマンドを使用して、Amazon DocumentDB イベントソースに関連する設定を更新できます。次の例では、バッチサイズを 1,000 に更新し、バッチウィンドウを 10 秒に更新します。このコマンドには、list-event-source-mapping コマンドまたは Lambda コンソールから取得できるイベントソースマッピングの UUID が必要です。

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

このような出力が表示されます。

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
```

```
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda は設定を非同期的に更新するため、プロセスが完了するまでこれらの変更が出力に表示されない場合があります。イベントソースマッピングの現在の設定を表示するには、get-event-source-mapping コマンドを使用します。

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

このような出力が表示されます。

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

Amazon DocumentDB イベントソースマッピングを削除するには、delete-event-source-mapping コマンドを使用します。

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

ポーリングとストリームの開始位置

イベントソースマッピングの作成時および更新時のストリームのポーリングは、最終的に一貫性があることに注意してください。

- イベントソースマッピングの作成時、ストリームからのイベントのポーリングが開始されるまでに数分かかる場合があります。
- イベントソースマッピングの更新時、ストリームからのイベントのポーリングが停止および再開されるまでに数分かかる場合があります。

つまり、LATEST をストリームの開始位置として指定すると、イベントソースマッピングの作成または更新中にイベントを見逃す可能性があります。イベントを見逃さないようにするには、ストリームの開始位置を TRIM_HORIZON または AT_TIMESTAMP として指定します。

Amazon DocumentDB イベントソースのモニタリング

Amazon DocumentDB イベントソースのモニタリングに役立つよう、関数がレコードのバッチの処理を完了したときに、Lambda は IteratorAge メトリクスを発行します。イテレータの有効期間とは、最新のイベントのタイムスタンプと現在のタイムスタンプの差のことです。基本的に、IteratorAge メトリクスは、バッチで最後に処理されたレコードの古さを示します。関数が新しいイベントを現在処理している場合、イテレーターの有効期間を使用して、レコードが追加されてから関数によって処理されるまでのレイテンシーを推定できます。IteratorAge の増加傾向は、関数に問題があることを示している可能性があります。詳細については、「Lambda 関数のメトリクスの使用」を参照してください。

Amazon DocumentDB の変更ストリームは、イベント間の大きなタイムギャップを処理するようには最適化されていません。Amazon DocumentDB イベントソースが長期間にイベントを受信しない場合、Lambda はイベントソースマッピングを無効にすることがあります。この期間の長さは、クラスターのサイズやその他のワークロードに応じて、数週間から数か月までさまざまです。

Lambda は最大 6 MB のペイロードをサポートします。ただし、Amazon DocumentDB 変更ストリームイベントのサイズは最大 16 MB です。変更ストリームが 6 MB を超える変更ストリームイベントを Lambda に送信しようとすると、Lambda はメッセージを削除して OversizedRecordCount メトリクスを発行します。Lambda は、ベストエフォートベースですべてのメトリクスを発行します。

チュートリアル: Amazon DocumentDB を用いて AWS Lambda のストリームの使用

このチュートリアルでは、Amazon DocumentDB (MongoDB 互換) 変更ストリームからのイベントを処理する基本的な Lambda 関数を作成します。このチュートリアルは、以下の段階を通じて完了します。

- Amazon DocumentDB クラスターをセットアップして接続し、そのクラスターで変更ストリームをアクティブ化します。
- Lambda 関数を作成し、Amazon DocumentDB クラスターを関数のイベントソースとして設定します。
- Amazon DocumentDB データベースにアイテムを挿入して、エンドツーエンドセットアップをテストします。

トピック

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [アカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウント のメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

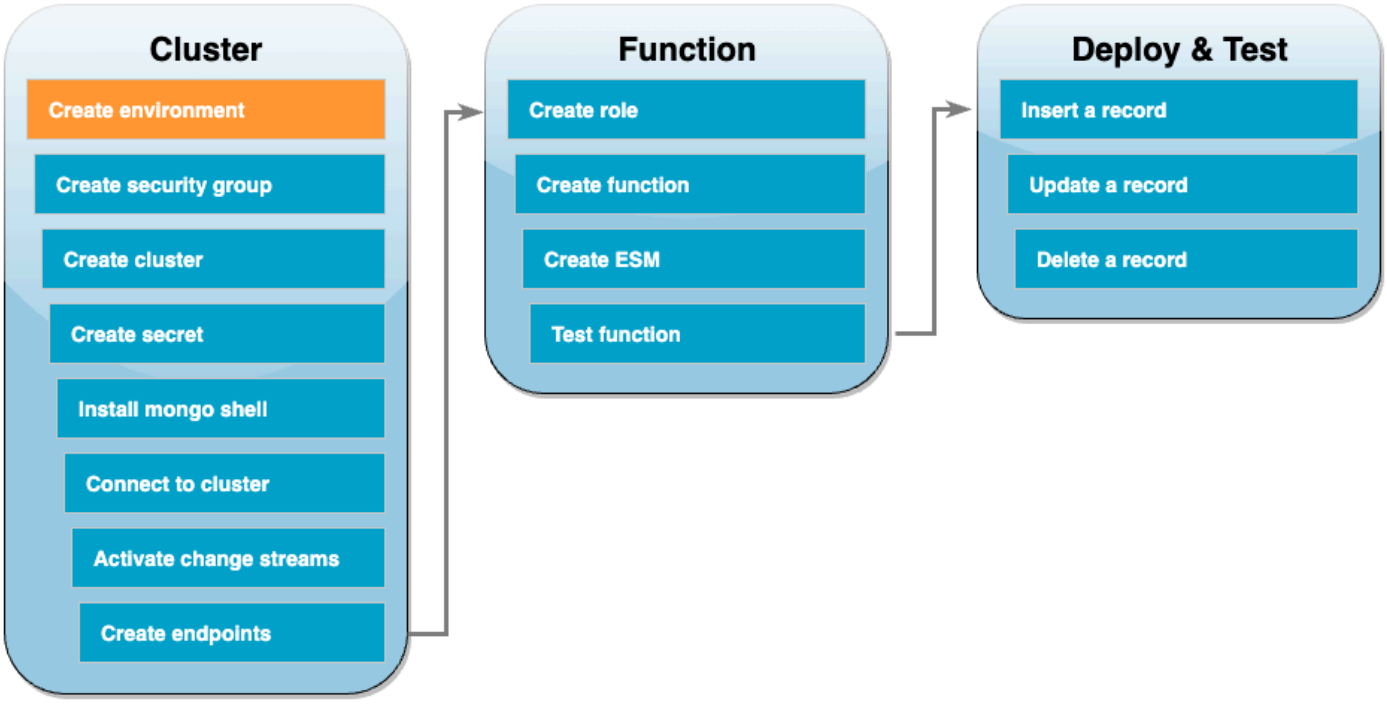
AWS Command Line Interface のインストール

AWS Command Line Interface をまだインストールしていない場合は、「最新バージョンの AWS CLI のインストールまたは更新」にある手順に従ってインストールしてください。

このチュートリアルでは、コマンドを実行するためのコマンドラインターミナルまたはシェルが必要です。Linux および macOS では、任意のシェルとパッケージマネージャーを使用してください。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (`zip` など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。

AWS Cloud9 環境を作成します。



Lambda 関数を作成する前に、Amazon DocumentDB クラスターを作成して設定する必要があります。このチュートリアルでクラスターをセットアップする手順は、「Amazon DocumentDB を開始する」の手順に基づいています。

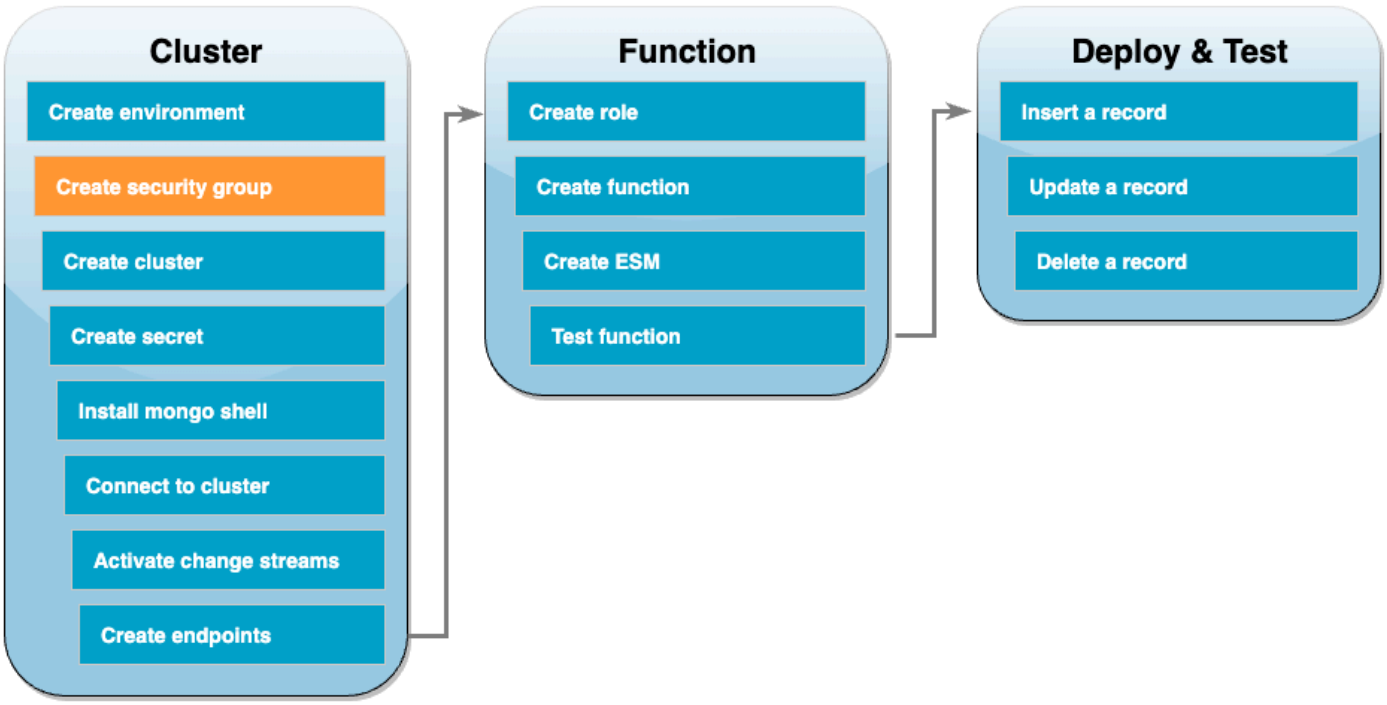
> **Note**
>
> Amazon DocumentDB クラスターをすでにセットアップしている場合は、必ず変更ストリームをアクティブ化し、必要なインターフェイス VPC エンドポイントを作成してください。これで、関数作成の手順に直接進むことができます。

最初に、AWS Cloud9 環境を作成します。このチュートリアルを通じて、DocumentDB クラスターに接続してクエリを実行するには、この環境を使用します。

AWS Cloud9 環境を作成するには

1. [Cloud9] コンソールを開いて、[環境を作成] を選択します。
2. 以下の構成で環境を作成します。
   - [詳細] の下:
     - 名前 – `DocumentDBCloud9Environment`
     - 環境タイプ — 新しい EC2 インスタンス
   - 新しい EC2 インスタンスの場合 の下:
     - インスタンスタイプ — `t2.micro` (1 GiB RAM + 1 vCPU)
     - プラットフォーム — Amazon Linux 2
     - タイムアウト — 30 分
   - [ネットワーク設定] の下:
     - 接続 — AWS Systems Manager (SSM)
     - [VPC 設定] ドロップダウンを展開します。
     - Amazon Virtual Private Cloud (VPC) — デフォルトの VPC を選択します。
     - サブネット — 指定なし
   - 他のデフォルト設定をすべて維持します。
3. [Create] (作成) を選択します。新しい AWS Cloud9 環境のプロビジョニングには数分かかることがあります。

EC2 セキュリティグループの作成



次に、DocumentDB クラスターと Cloud9 環境間のトラフィックを許可するルールを含む EC2 セキュリティグループを作成します。

EC2 セキュリティグループを作成するには

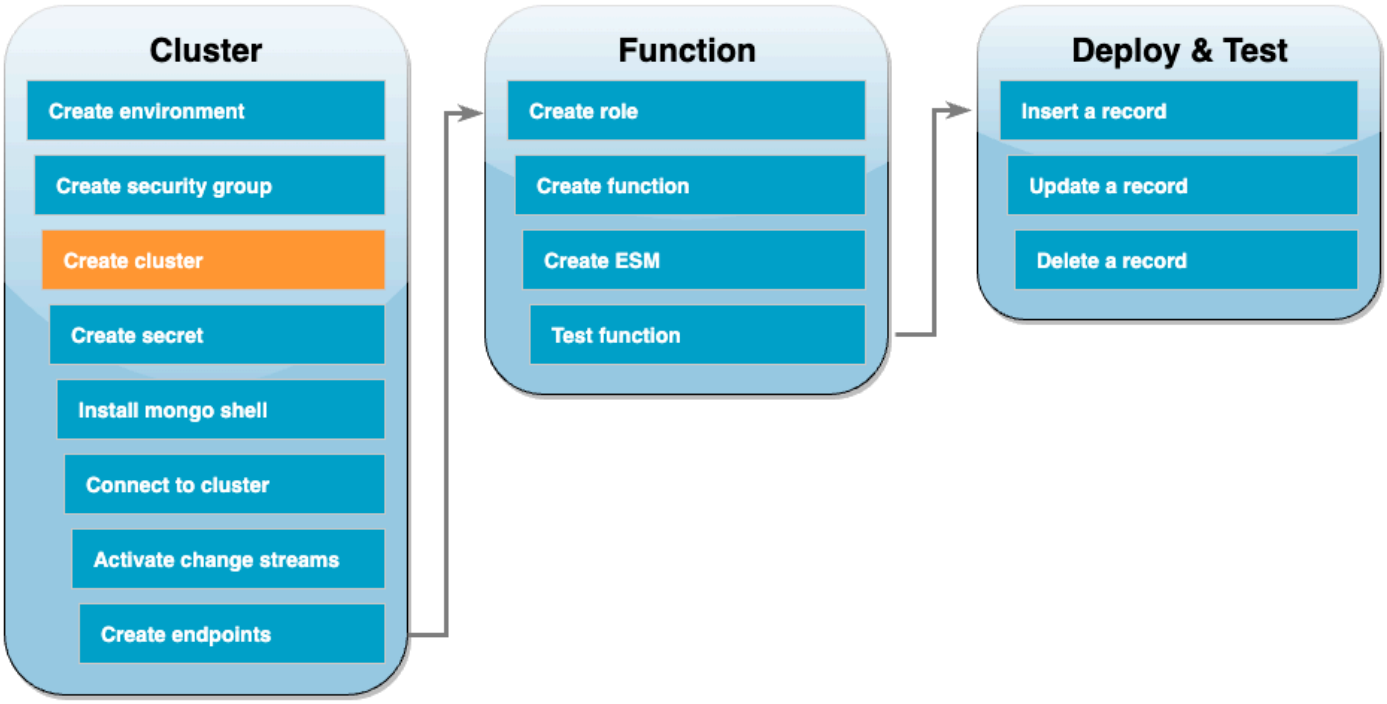
1. EC2 コンソールを開きます。[ネットワークとセキュリティ] で、[セキュリティグループ] を選択します。
2. [Create Security Group] を選択します。
3. 次の構成でセキュリティグループを作成します。

   - [基本情報] の下:
     - セキュリティグループ名: - `DocDBTutorial`
     - Description — Cloud9 と DocumentDB 間のトラフィック用のセキュリティグループ。
     - VPC — [デフォルトの VPC] を選択します。
   - [インバウンドルール] で、[ルールの追加] を選択します。次の設定でルールを作成します。
     - Type - カスタム TCP
     - Port range - 27017
     - Source - カスタム

- [Source] の横にある検索ボックスで、前のステップで作成した AWS Cloud9 環境のセキュリティグループを選択します。使用可能なセキュリティグループのリストを表示するには、検索ボックスに「cloud9」を入力します。aws-cloud9-<environment_name> という名前のセキュリティグループを選択します。
- 他のデフォルト設定をすべて維持します。

4. [Create Security Group] を選択します。

DocumentDB クラスターを作成



このステップでは、前のステップからのセキュリティグループを使用して DocumentDB クラスターを作成します。

DocumentDB クラスターを作成するには

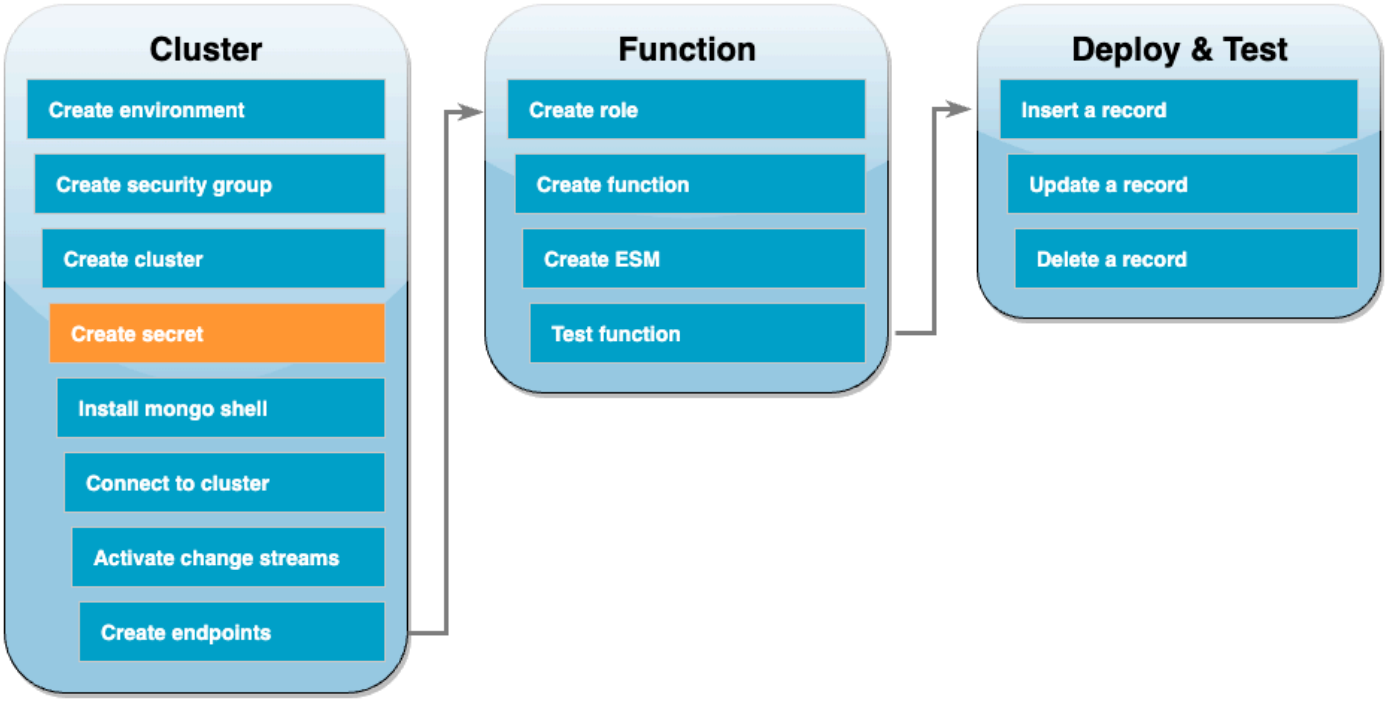
1. [DocumentDB] コンソールを開きます。[クラスタ] で [作成] を選択します。
2. 次の設定でクラスターを作成します。
   - [クラスタータイプ] には、インスタンスベースのクラスターを選択します。
   - [構成] の下:
     - エンジンバージョン - 5.0.0

- インスタンスクラス – db.t3.medium (無料トライアル対象)
- インスタンス数 - 1

- [認証] の下:
  - クラスターへの接続に必要なユーザー名とパスワードを入力します (前のステップでシークレットを作成したときと同じ認証情報)。[パスワードの確認] で、パスワードを確認します。
- [アドバンスト設定の表示] を切り替えます。
- [ネットワーク設定] の下:
  - 仮想プライベートクラウド (VPC)]) - [デフォルトの VPC] を選択します。
  - [サブネットグループ] - デフォルト
  - VPC セキュリティグループ — `default（VPC）`に加え、前のステップで作成した `DocDBTutorial（VPC）`セキュリティグループを選択します。
- 他のデフォルト設定をすべて維持します。

3. [クラスターを作成] を選択します。DocumentDB クラスターのプロビジョニングには数分がかかる場合があります。

Secrets Manager でシークレットを作成する



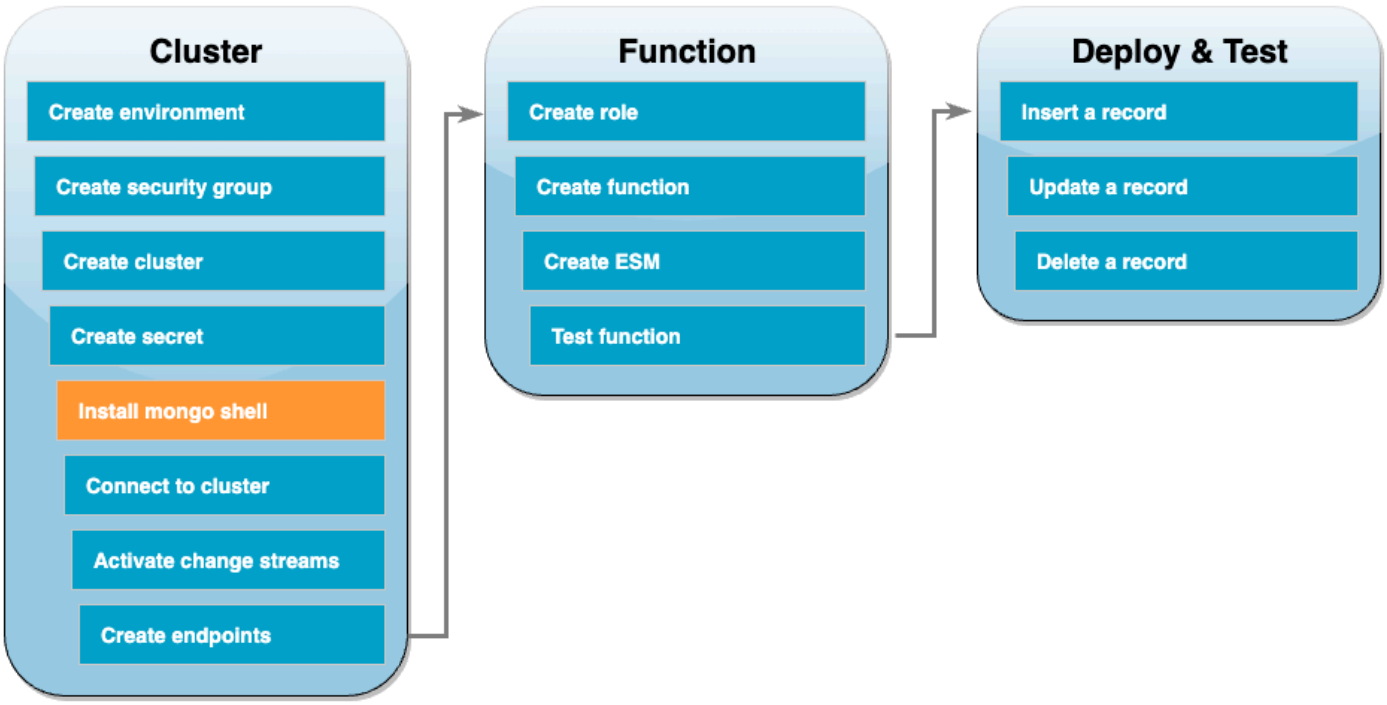
DocumentDB クラスターに手動でアクセスするには、ユーザー名とパスワードの認証情報を入力する必要があります。Lambda がクラスターにアクセスするには、イベントソースマッピングを設定するときに、同じアクセス認証情報を含む Secrets Manager のシークレットを指定する必要があります。このステップでは、このシークレットを作成します。

Secrets Manager でシークレットを保存するには

1. [Secrets Manager] コンソールを開き、[新しいシークレットを保存] を選択します。
2. [シークレットのタイプを選択] で、以下のいずれかのオプションを選択します。
   - [基本情報] の下:
     - Secret type — Amazon DocumentDB データベース用認証情報
     - [認証情報] で、DocumentDB クラスターへのアクセスに使用するユーザー名とパスワードを入力します。
     - Database — ご使用の DocumentDB クラスターを選択します。
     - [Next] を選択します。
3. [条件] は、以下のオプションから選択します。
   - シークレット名 – `DocumentDBSecret`
   - [Next] を選択します。
4. [Next] を選択します。
5. [保存する] を選択します。
6. コンソールを更新して、`DocumentDBSecret` シークレットが正常に保存されたことを確認します。

シークレットの「シークレット ARN」を書き留めておきます。これは、後のステップで必要になります。

mongo シェルをインストールする



このステップでは、Cloud9 環境に mongo シェルをインストールします。mongo シェルは、DocumentDB クラスターを接続してクエリするために使用するコマンドラインユーティリティです。

Cloud9 環境に mongo シェルをインストールするには

1. [Cloud9] コンソール を開きます。以前に作成した DocumentDBCloud9Environment 環境の横にある [Cloud9 IDE] 列の下の [開く] リンクをクリックします。
2. ターミナルウィンドウで、次のコマンドを使用して MongoDB リポジトリファイルを作成します。

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://www.mongodb.org/static/pgp/server-5.0.asc" | sudo tee /etc/
yum.repos.d/mongodb-org-5.0.repo
```

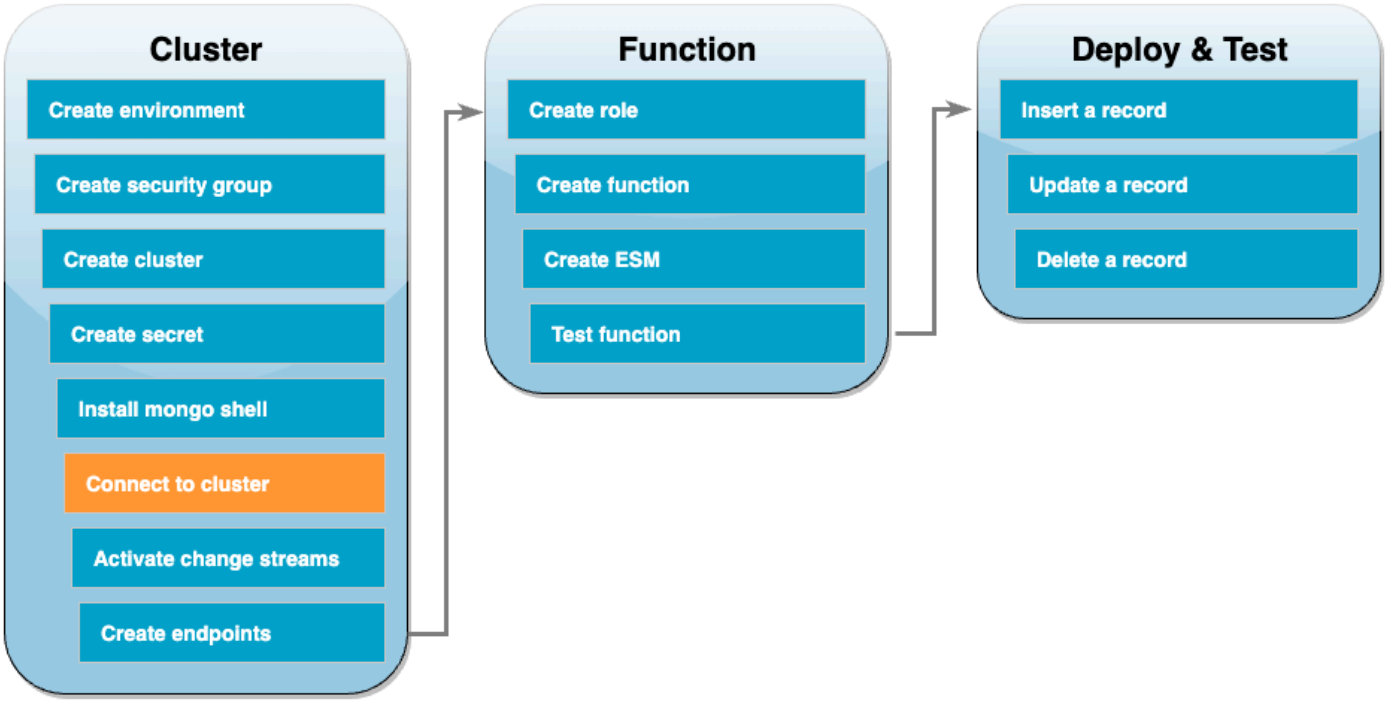
3. 次に、以下のコマンドを使用して mongo シェルをインストールします。

```
sudo yum install -y mongodb-org-shell
```

4. 転送中のデータを暗号化するには、Amazon DocumentDB のパブリックキーをダウンロードします。次のコマンドでは、global-bundle.pem という名前のファイルをダウンロードします。

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

DocumentDB クラスターに接続する



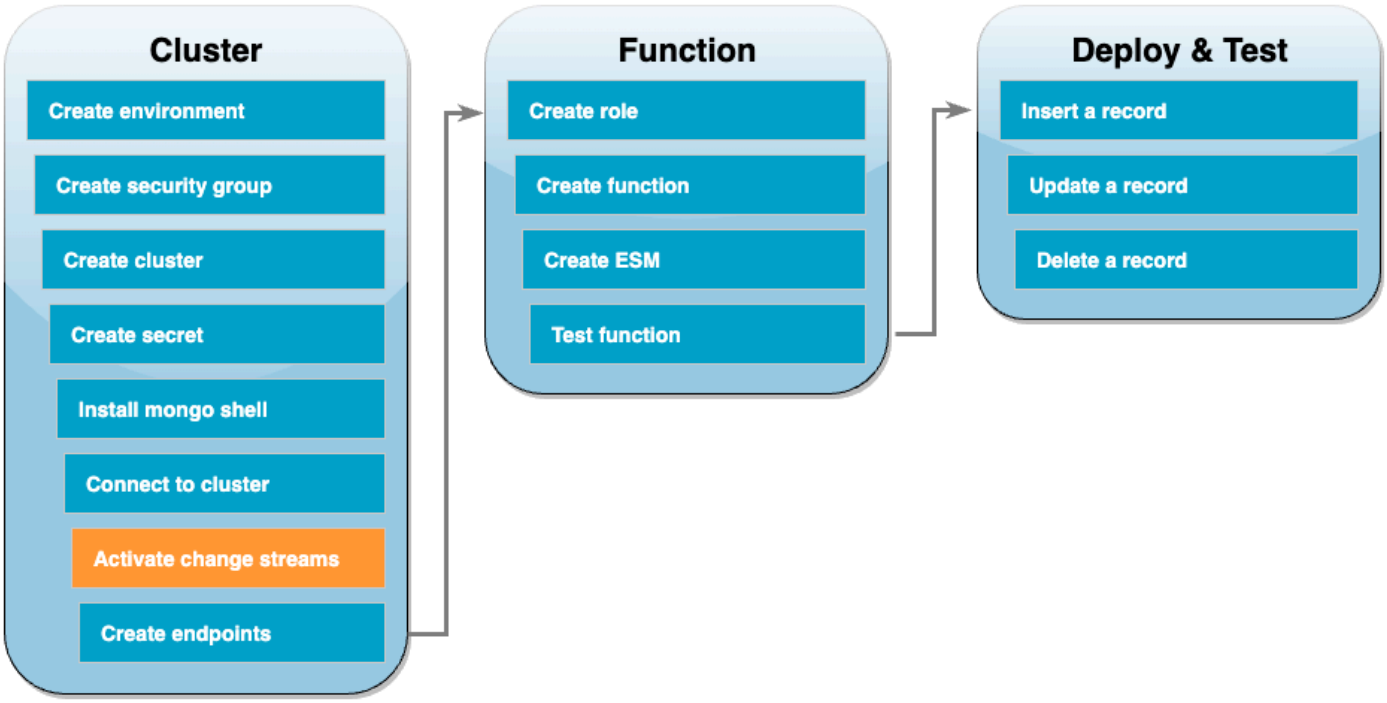
これで、mongo シェルを使用して DocumentDB クラスターに接続する準備が整いました。

DocumentDB クラスターに接続するには

1. [DocumentDB] コンソールを開きます。[クラスター] で、クラスター識別子を選択してクラスターを選択します。
2. [接続性とセキュリティ] タブの、[mongo シェルを使用してこのクラスターに接続] の下で [コピー] を選択します。
3. Cloud9 環境で、このコマンドをターミナルに貼り付けます。<insertYourPassword> を正しいパスワードと交換します。

このコマンドを入力した後、コマンドプロンプトが `rs0:PRIMARY>` になれば、Amazon DocumentDB クラスターに接続されています。

変更ストリームを有効にする



このチュートリアルでは、DocumentDB クラスター内の`docdbdemo` データベースの `products` コレクションの変更をトラックします。これを行うには、[変更ストリーム] を有効にします。まず、`docdbdemo` データベースを作成し、レコードを挿入してテストします。

クラスター内に新しいデータベースを作成するには

1. Cloud9 環境で、まだ DocumentDB クラスターに接続しているかを確認します。
2. ターミナルウィンドウで、次のコマンドを使用して、`docdbdemo` という名前の新しいデータベースを作成します。

   ```
   use docdbdemo
   ```

3. 次に、以下のコマンドを使用してレコードを `docdbdemo` に挿入します。

   ```
   db.products.insert({"hello":"world"})
   ```

   次のような出力が表示されます。

```
WriteResult({ "nInserted" : 1 })
```

4. すべてのデータベースを一覧表示するには、以下のコマンドを使用します。

```
show dbs
```

出力に docdbdemo データベースが含まれていることを確認してください。

```
docdbdemo  0.000GB
```

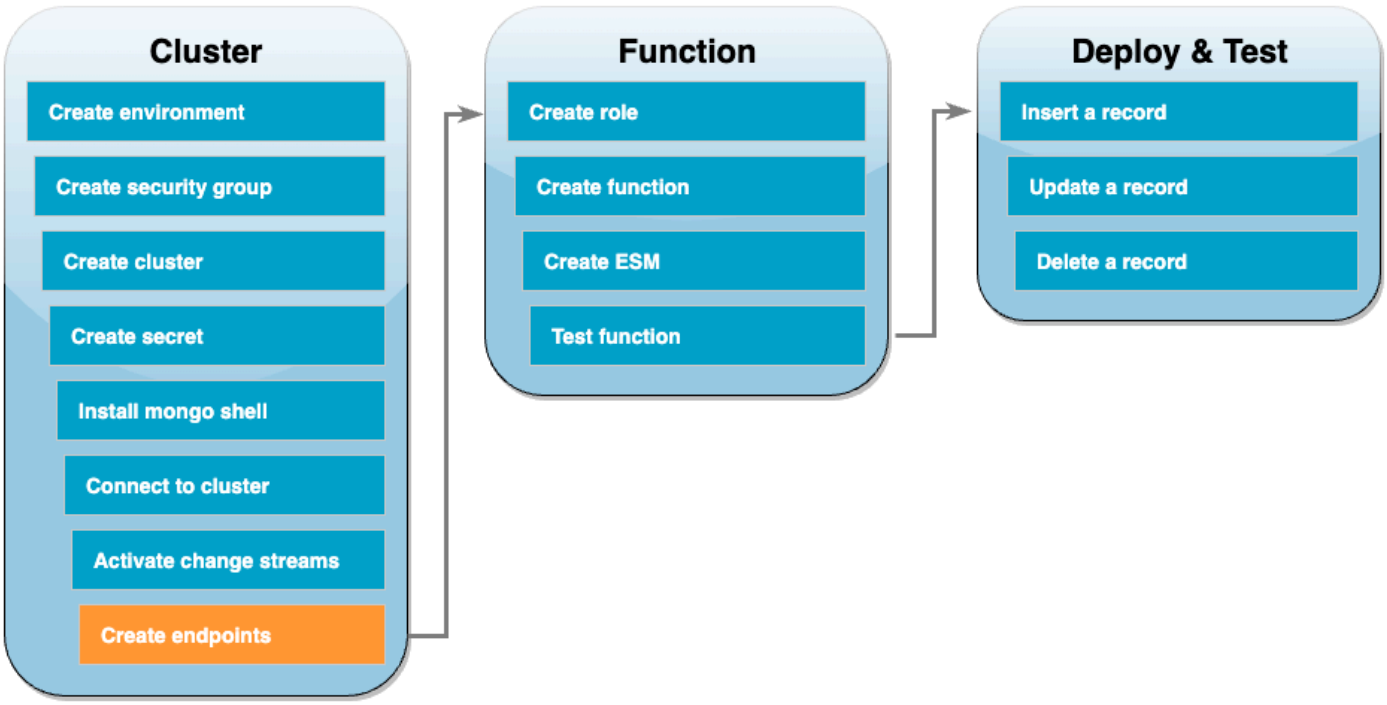
次に、次のコマンドを使用して、docdbdemo データベースの products コレクションの変更ストリームを有効にします。

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

次のような出力が表示されます。

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

インターフェイス VPC エンドポイントを作成する



次に、インターフェイス VPC エンドポイントを作成して、Lambda と Secrets Manager (後でクラスターアクセス認証情報を保存するために使用) がデフォルト VPC に接続できるようにします。

インターフェイス VPC エンドポイントを作成するには

1. [VPC] コンソールを開きます。左側のメニューの [仮想プライベートクラウド] で、[エンドポイント] を選択します。
2. [エンドポイントの作成] を選択します。次の構成でエンドポイントを作成します。
   - [名前タグ] に「`lambda-default-vpc`」を入力します。
   - [サービスカテゴリ] で、[AWS サービス] を選択します。
   - サービスには、検索ボックスで「`lambda`」と入力します。フォーマット `com.amazonaws.<region>.lambda` のサービスを選択してください。
   - [VPC] で「デフォルトの VPC」を 選択します。
   - サブネット には、各アベイラビリティーゾーンの横にあるボックスをチェックします。それぞれのアベイラビリティゾーンに正しいサブネット ID を選択します。
   - [IP アドレスの種類] には [IPv4] を選択します。

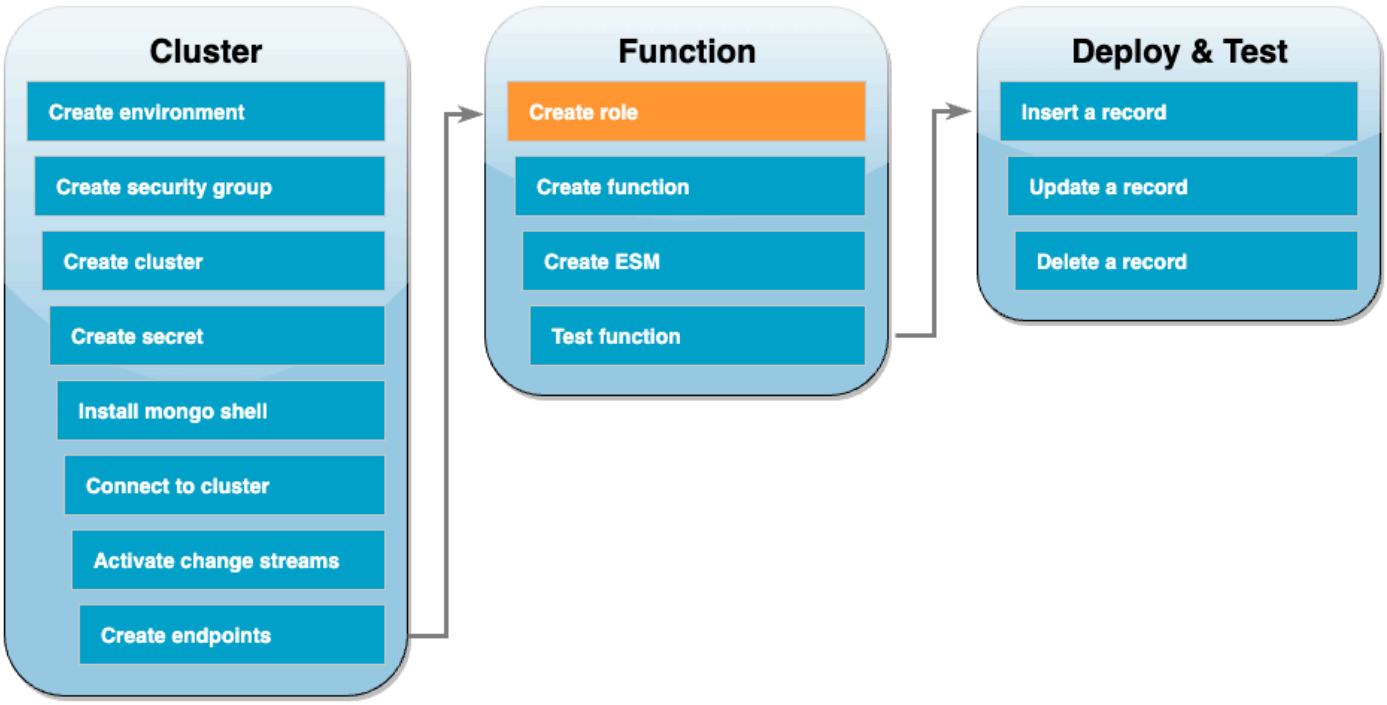
- セキュリティグループには、デフォルトの VPC セキュリティグループ (default のグループ名) と、以前に作成したセキュリティグループ (DocDBTutorial のグループ名) を選択します。
- 他のデフォルト設定をすべて維持します。
- [エンドポイントの作成] を選択します。

3. [エンドポイントの作成] を再び選択します。次の構成でエンドポイントを作成します。

- [名前タグ] に「secretsmanager-default-vpc」を入力します。
- [サービスカテゴリ] で、[AWS サービス] を選択します。
- サービスには、検索ボックスで「secretsmanager」と入力します。フォーマット com.amazonaws.<region>.secretsmanager のサービスを選択してください。
- [VPC] で「デフォルトの VPC」を 選択します。
- サブネット には、各アベイラビリティーゾーンの横にあるボックスをチェックします。それぞれのアベイラビリティゾーンに正しいサブネット ID を選択します。
- [IP アドレスの種類] には [IPv4] を選択します。
- セキュリティグループには、デフォルトの VPC セキュリティグループ (default のグループ名) と、以前に作成したセキュリティグループ (DocDBTutorial のグループ名) を選択します。
- 他のデフォルト設定をすべて維持します。
- [エンドポイントの作成] を選択します。

これで、このチュートリアルのクラスターセットアップの部分は完了です。

実行ロールを作成する



次のステップでは、Lambda 関数を作成します。まず、クラスターにアクセスするためのアクセス許可を関数に付与する実行ロールを作成する必要があります。これを行うには、最初に IAM ポリシーを作成してから、次にこのポリシーを IAM ロールにアタッチします。

IAM ポリシーを作成するには

1. IAM コンソールの [ポリシー] ページ を開き、[ポリシーの作成] を選択します。
2. [JSON] タブを選択します。次のポリシーでは、ステートメントの最後の行にある Secrets Manager リソース ARN を以前のシークレット ARN で置き換え、ポリシーをエディタにコピーします。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
```

```
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

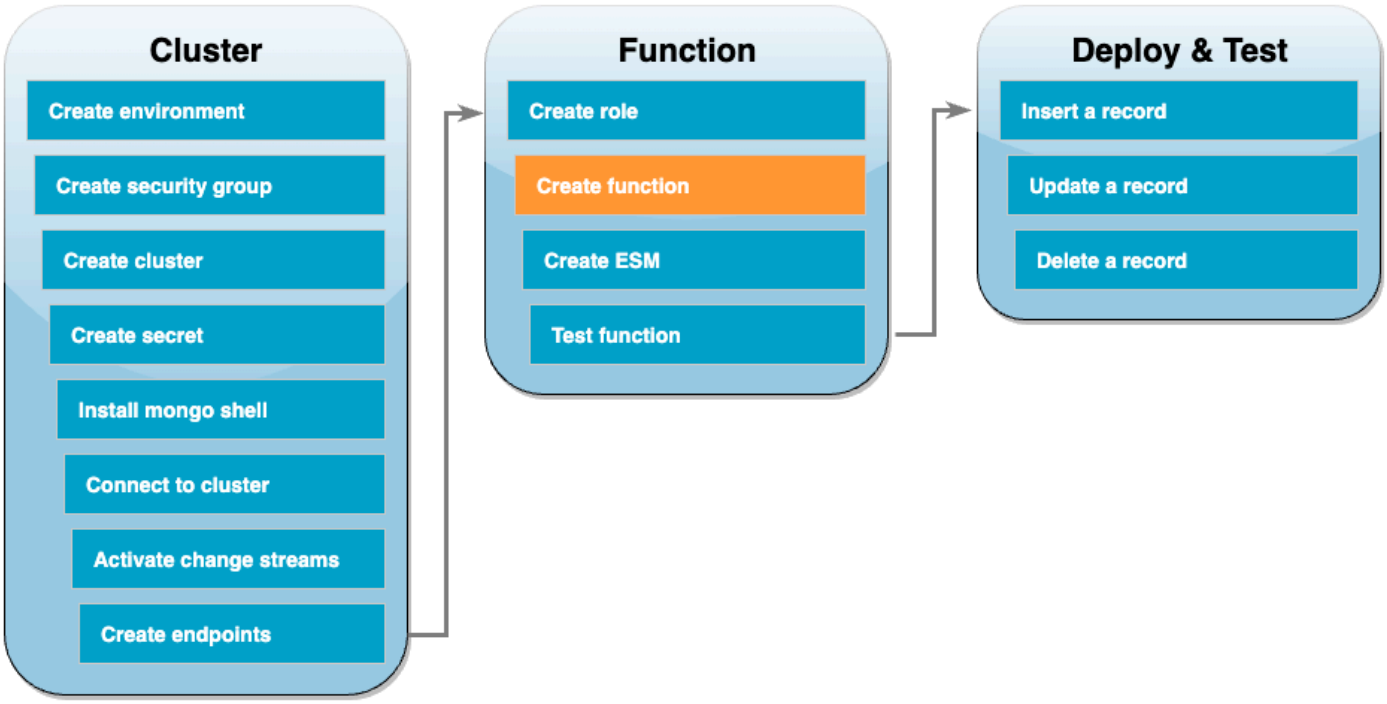
3. [次へ: タグ]、[次へ: 確認] の順に選択します。
4. [Name] (名前) に `AWSDocumentDBLambdaPolicy` と入力します。
5. [Create policy] を選択します。

IAM ロールを作成するには

1. IAM コンソールの [ロール] ページを開いて、[ロールの作成] を選択します。
2. [信頼できるエンティティを選択] には、次のオプションを選択します。
   - [信頼できるエンティティタイプ] – AWS サービス
   - [ユースケース] – Lambda
   - [Next] を選択します。

3. [アクセス権限の追加] では、作成したばかりの AWSDocumentDBLambdaPolicy ポリシーを選択し、AWSLambdaBasicExecutionRole と同様に関数に Amazon CloudWatch Logs への書き込み権限を付与します。
4. [Next] を選択します。
5. [Role name] (ロール名) にAWSDocumentDBLambdaExecutionRoleと入力します。
6. [ロールの作成] を選択します。

Lambda 関数を作成する



以下のコード例では、DocumentDB イベント入力を受け取り、含まれるメッセージを処理します。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で Amazon DocumentDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
```

```
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で Amazon DocumentDB イベントの消費。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で Amazon DocumentDB イベントの消費。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で Amazon DocumentDB イベントの消費。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Lambda 関数を作成するには

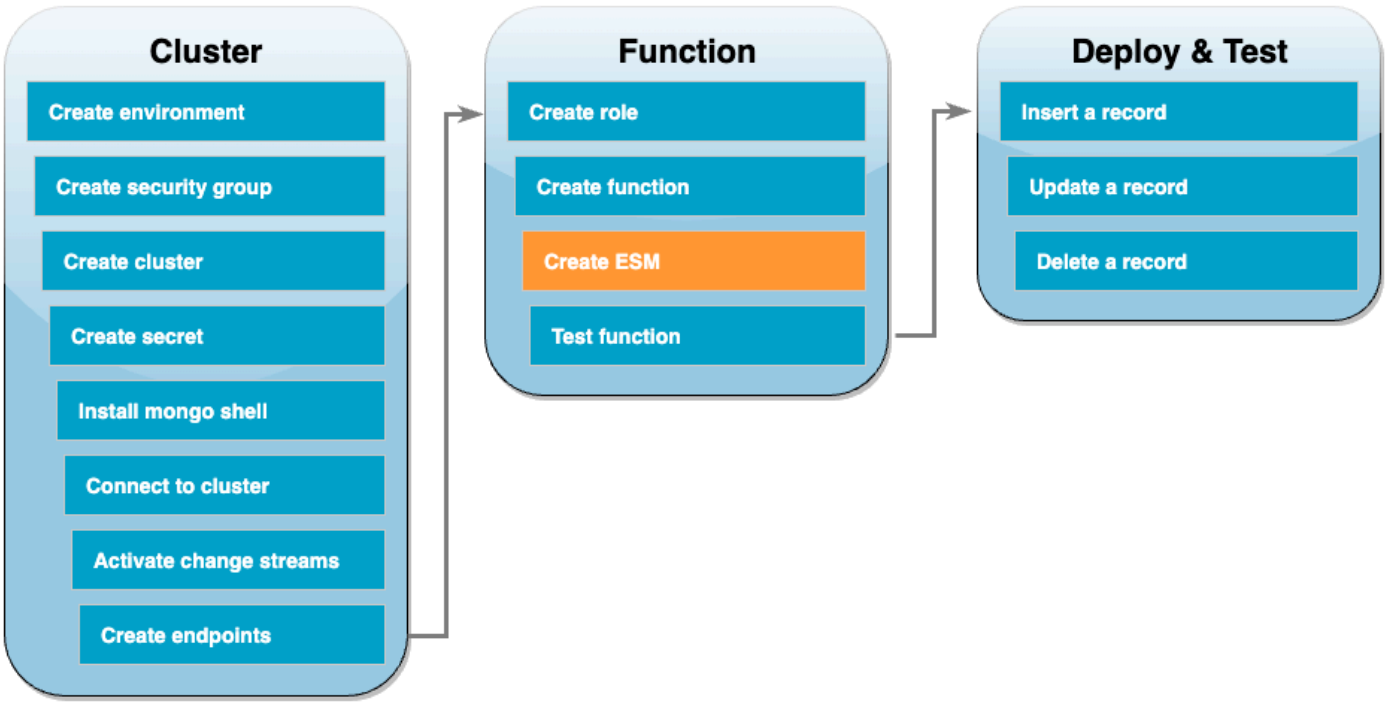
1. サンプルコードを index.js という名前のファイルにコピーします。
2. 以下のコマンドを使用して、デプロイパッケージを作成します。

```
zip function.zip index.js
```

3. CLI コマンドを使用して、関数を作成します。us-east-1 をリージョンに、123456789012 をアカウント ID で置き換えます。

```
aws lambda create-function --function-name ProcessDocumentDBRecords \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
    --region us-east-1 \
    --role arn:aws:iam::123456789012:role/AWSDocumentDBLambdaExecutionRole
```

Lambda イベントソースマッピングを作成します。



DocumentDB 変更ストリームを Lambda 関数と関連付けるイベントソースマッピングを作成します。このイベントソースマッピングを作成すると、AWS Lambda はストリームのポーリングをすぐに開始します。

イベントソースマッピングを作成するには

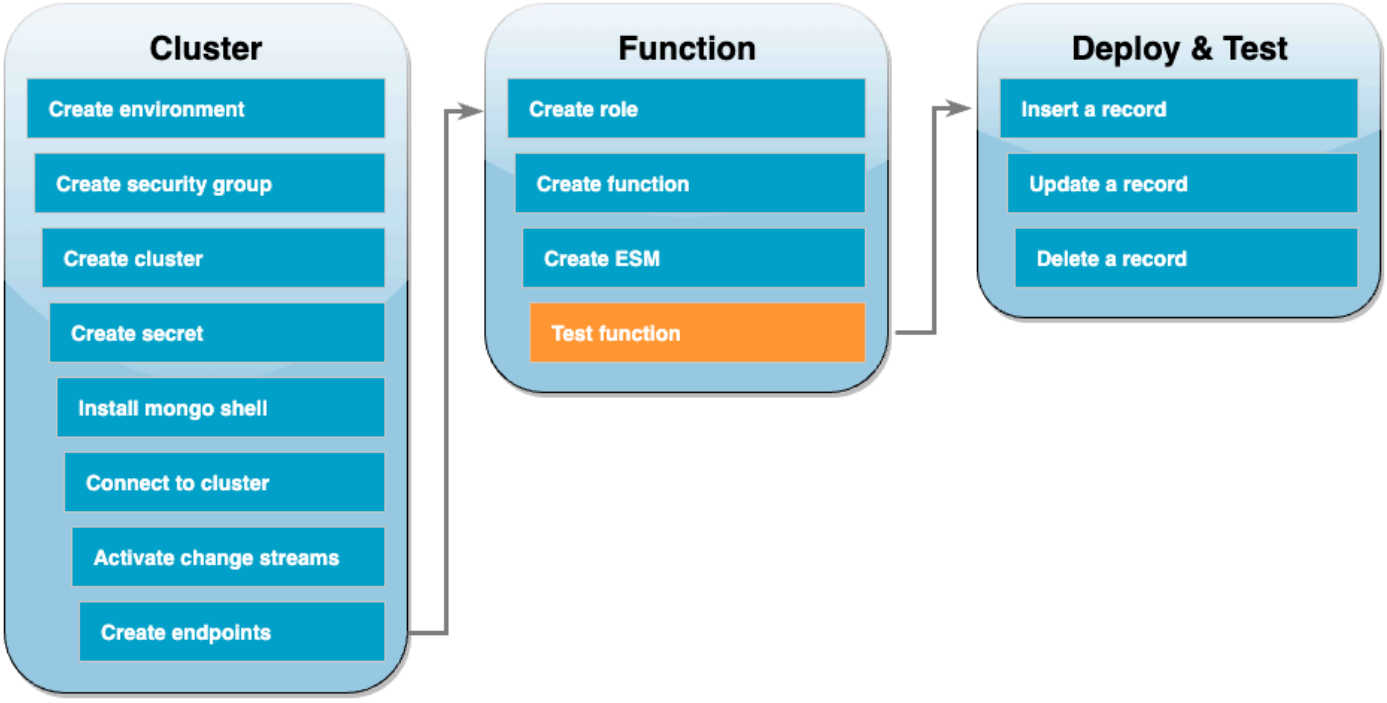
1. Lambda コンソールの [関数] ページを開きます。
2. 先ほど作成した `ProcessDocumentDBRecords` 関数を選択します。
3. [設定] タブを選択し、左側のメニューで [トリガー] を選択します。
4. [Add trigger] を選択します。
5. [トリガー設定] で、ソースとして [DocumentDB] を選択します。
6. イベントソースマッピングには、次の設定制限があります。

   - DocumentDB クラスター — 以前に作成したクラスターを選択します。
   - データベース名 – `docdbdemo`
   - コレクション名 — 製品
   - バッチサイズ - 1
   - 開始位置 — 最新

- 認証 — BASIC_AUTH
- Secrets Manager キー — 作成したばかりの DocumentDBSecret キーを選択します。
- バッチウィンドウ – 1
- フルドキュメント設定 — UpdateLookup

7. 追加 を選択します。イベントソースマッピングの作成には数分かかる場合があります。

関数をテストする - 手動呼び出し



関数とイベントソースマッピングが正しく作成されたことをテストするには、invoke コマンドを使用して関数を呼び出します。そのためには、まず次のイベント JSON を input.txt という名前のファイルにコピーします。

```
{
  "eventSourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:canaryclusterb2a659a2-
qo5tcmqkcl03",
  "events": [
    {
      "event": {
        "_id": {
          "_data": "0163eeb6e700000009010000009000041e1"
        },
```

```
        "clusterTime": {
          "$timestamp": {
            "t": 1676588775,
            "i": 9
          }
        },
        "documentKey": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          }
        },
        "fullDocument": {
          "_id": {
            "$oid": "63eeb6e7d418cd98afb1c1d7"
          },
          "anyField": "sampleValue"
        },
        "ns": {
          "db": "docdbdemo",
          "coll": "products"
        },
        "operationType": "insert"
      }
    }
  ],
  "eventSource": "aws:docdb"
}
```

次に、以下のコマンドを使用して、このイベントを処理する関数を呼び出します。

```
aws lambda invoke --function-name ProcessDocumentDBRecords \
    --cli-binary-format raw-in-base64-out \
    --region us-east-1 \
    --payload file://input.txt out.txt
```

以下のようなレスポンスが表示されます。

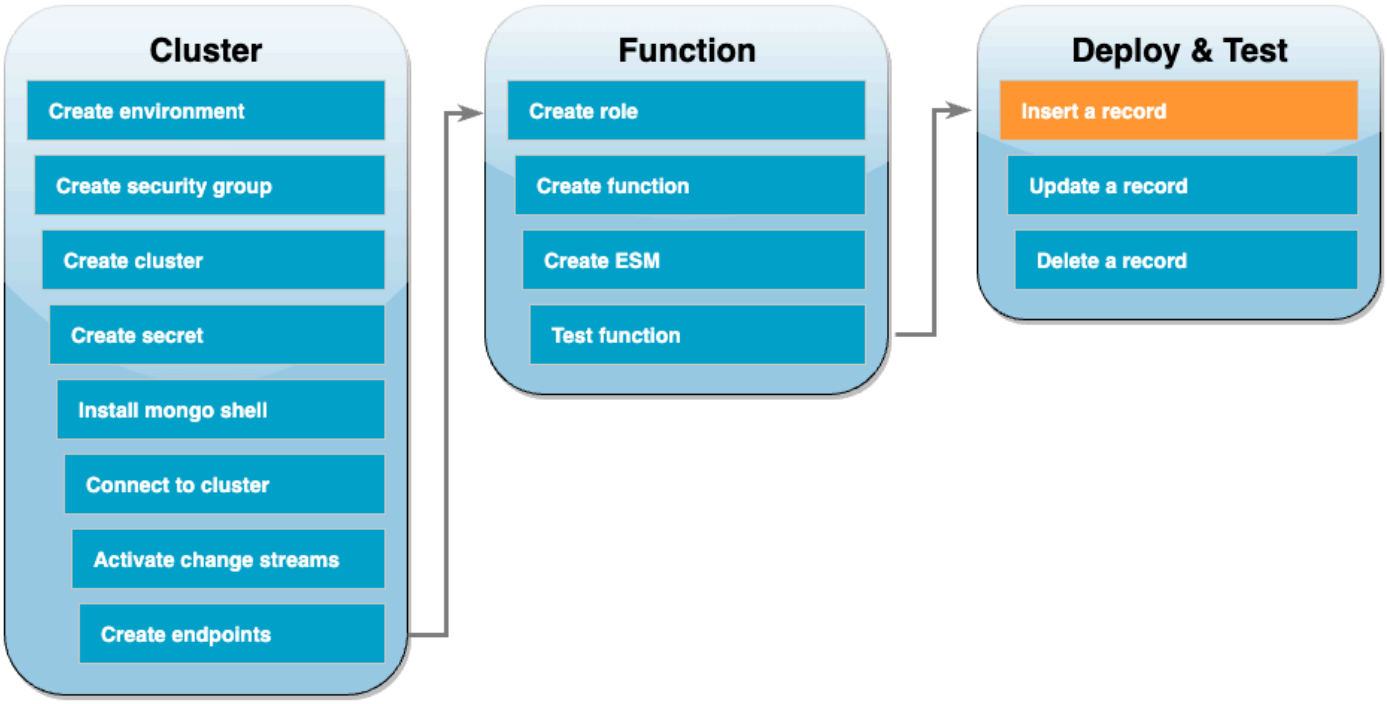
```
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```

関数がイベントを正常に処理したかどうかは、CloudWatch Logs を確認することで確認できます。

CloudWatch Logs による手動呼び出しを確認するには

1. Lambda コンソールの [関数] ページを開きます。
2. [モニタリング] タブから、[CloudWatch のログを表示] を選択します。これにより、CloudWatch コンソールの関数に関連する特定のロググループに移動します。
3. 最新のログストリームを選択します。ログメッセージには、イベント JSON が表示されます。

関数のテスト - レコードを挿入



DocumentDB データベースと直接やり取りして、エンドツーエンドのセットアップをテストします。次のステップでは、レコードを挿入して更新し、削除します。

レコードを挿入するには

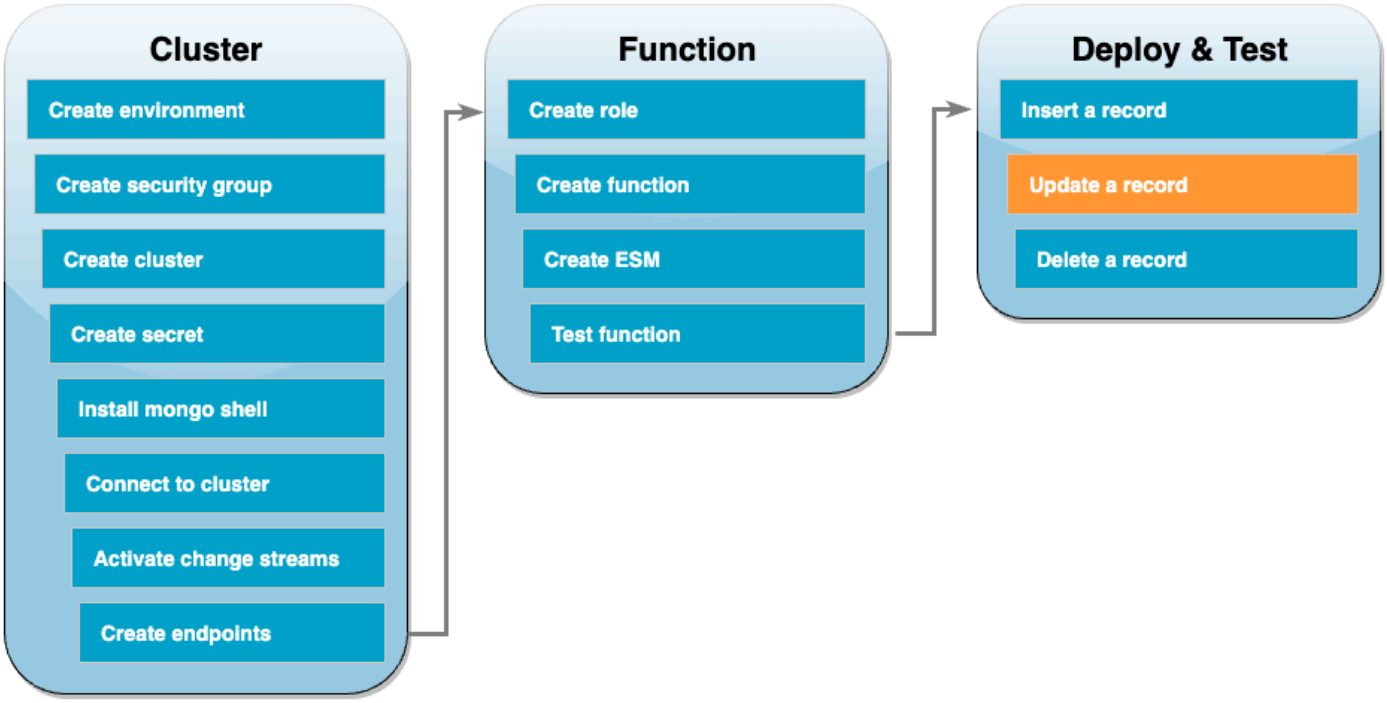
1. Cloud9 環境の DocumentDB クラスターに再接続します。
2. このコマンドを使用して、次の docdbdemo データベースを使用していることを確認してください。

```
use docdbdemo
```

3. docdbdemo データベースの products コレクションにレコードを挿入します。

```
db.products.insert({"name":"Pencil", "price": 1.00})
```

関数のテスト - レコードの更新

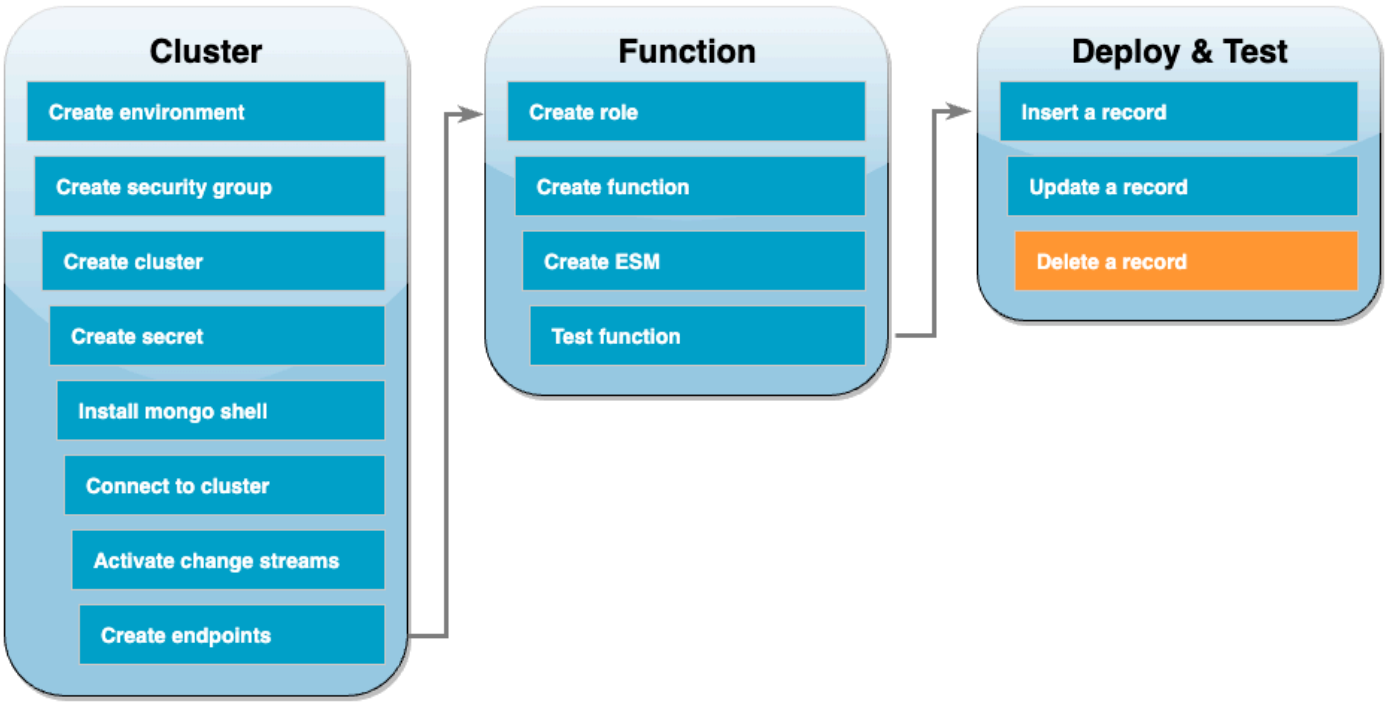


次に、以下のコマンドを使用して、挿入したレコードを更新します。

```
db.products.update(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

CloudWatch Logs をチェックして、関数がこのイベントを正常に処理したことを確認します。

関数のテスト - レコードの削除



最後に、次のコマンドを使用して更新したレコードを削除します。

```
db.products.remove( { "name": "Pencil" } )
```

CloudWatch Logs をチェックして、関数がこのイベントを正常に処理したことを確認します。

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

VPC エンドポイントを削除するには

1. [VPC] コンソールを開きます。左側のメニューの [仮想プライベートクラウド] で、[エンドポイント] を選択します。
2. 作成したエンドポイントを選択します。
3. [Actions] (アクション)、[Delete VPC endpoints] (VPC エンドポイントを削除) の順に選択します。
4. テキスト入力フィールドに **`delete`** を入力します。
5. [削除] を選択します。

Amazon DocumentDB クラスターを削除するには

1. [DocumentDB] コンソールを開きます。
2. このチュートリアル用に作成した DocumentDB クラスターを選択し、削除保護を無効にします。
3. メインのクラスターページで、DocumentDB クラスターをもう一度選択します。
4. [アクション]、[削除] の順に選択します。
5. [最終クラススナップショットの作成] で [いいえ] を選択します。
6. テキスト入力フィールドに **`delete`** を入力します。
7. [削除] を選択します。

シークレットを Secrets Manager で削除するには

1. Secrets Manager コンソールを開きます。
2. このチュートリアルで作成したシークレットを選択します。
3. [アクション]、[シークレットの削除] を選択します。
4. [Schedule deletion] (削除をスケジュールする) を選択します。

Amazon EC2 セキュリティグループを削除するには

1. EC2 コンソールを開きます。[ネットワークとセキュリティ] で、[セキュリティグループ] を選択します。
2. このチュートリアルで作成したセキュリティグループを選択します。
3. [アクション]、[セキュリティグループの削除] の順に選択します。
4. [削除] を選択します。

Cloud9 環境を削除するには

1. [Cloud9] コンソール を開きます。
2. このチュートリアル用に作成した環境を選択します。
3. [削除] を選択します。
4. テキスト入力フィールドに **delete** を入力します。
5. [削除] を選択します。

Lambda のイベントフィルタリング

イベントフィルタリングを使用して、Lambda が関数に送信するストリームまたはキューからのレコードを制御することができます。たとえば、フィルターを追加して、関数が特定のデータパラメータを含む Amazon SQS メッセージのみを処理するようにできます。イベントフィルタリングはイベントソースマッピングと連動します。次の AWS サービスのイベントソースマッピングにフィルターを追加できます。

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- セルフマネージド Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

Lambda は Amazon DocumentDB のイベントフィルタリングをサポートしていません。

デフォルトでは、単一のイベントソースマッピングに最大 5 つの異なるフィルターを定義することができます。フィルターは論理 OR で結合されています。イベントソースのレコードが 1 つ以上の

フィルター条件を満たす場合、Lambda は関数に送信する次のイベントにそのレコードを含めます。どのフィルターも条件を満たさない場合、Lambda はレコードを破棄します。

> **Note**
>
> イベントソースに 5 つを超えるフィルターを定義する必要がある場合、イベントソースごとに最大 10 フィルターまでクォータ引き上げをリクエストできます。現在のクォータで許可されている数よりも多くのフィルターを追加しようとすると、イベントソースの作成時に Lambda はエラーを返します。

トピック

イベントフィルタリングの基本

フィルター条件 (`FilterCriteria`) オブジェクトは、フィルターのリスト (`Filters`) で構成される構造です。各フィルターは、イベントのフィルタリングパターン (`Pattern`) を定義する構造です。パターンは、JSON フィルタールールを文字列で表したものです。`FilterCriteria` オブジェクトの構造は次の通りです。

```
{
   "Filters": [
        {
```

```
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

フィルターパターンには、メタデータプロパティ、データプロパティ、またはその両方を含めることができます。使用可能なメタデータパラメータおよびデータパラメータの形式は、イベントソースとして機能している AWS のサービス によって異なります。たとえば、イベントソースマッピングが Amazon SQS キューから次のレコードを受け取ったとします。

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "City": "Seattle",\n "State": "WA",\n "Temperature": "46"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- メタデータプロパティは、レコードを作成したイベントに関する情報を含むフィールドです。Amazon SQS レコードの例では、メタデータプロパティには messageID、eventSourceArn、awsRegion などのフィールドが含まれます。

- データプロパティは、ストリームまたはキューからのデータを含むレコードのフィールドです。Amazon SQS イベントの例では、データフィールドのキーは body であり、データプロパティはフィールド City、State、Temperature です。

イベントソースの種類が異なれば、データフィールドに使用するキー値も異なります。データプロパティをフィルタリングするには、フィルターのパターンに正しいキーが使われていることを確認してください。データフィルタリングキーのリストおよびサポートされている各 AWS のサービス のフィルターパターンの例については、異なる AWS のサービス を持つフィルターの使用 を参照してください。

イベントフィルタリングは、マルチレベル JSON のフィルタリングを処理できます。たとえば、DynamoDB ストリームのレコードに次のフラグメントがあるとします。

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

ソートキー Number の値が 4567 のレコードのみを処理するとします。この場合、FilterCriteria オブジェクトは以下のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
```

```
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

フィルター条件を満たさないレコードの処理

フィルター条件を満たさないレコードの処理方法は、イベントソースによって異なります。

- Amazon SQSでは、メッセージがフィルター条件を満たさない場合、Lambda はそのメッセージをキューから自動的に削除します。Amazon SQS でこれらのメッセージを手動で削除する必要はありません。
- Kinesis および DynamoDB では、フィルター条件がレコードを処理すると、ストリームイテレータはこのレコードを通り越して先に進みます。レコードがフィルター条件を満たさない場合に、そのレコードをイベントソースから手動で削除する必要はありません。保持期間が過ぎると、Kinesis と DynamoDB はこれらの古いレコードを自動的に削除します。それより早くレコードを削除したい場合は、「データ保持期間の変更」を参照してください。
- Amazon MSK、セルフマネージド Apache Kafka、Amazon MQ メッセージの場合、Lambda はフィルターに含まれるすべてのフィールドに一致しないメッセージを削除します。セルフマネージド Apache Kafka では、Lambda は関数を正常に呼び出した後、一致するメッセージと一致しないメッセージのオフセットをコミットします。Amazon MQ の場合、Lambda は関数を正常に呼び出した後で一致するメッセージを認識し、一致しないメッセージはフィルタリング時に認識します。

フィルタールールの構文

フィルタールールの場合、Lambda は Amazon EventBridge ルールをサポートしており、EventBridge と同じ構文を使用します。詳細については、「Amazon EventBridge ユーザーガイド」の「Amazon EventBridge のイベントパターン」を参照してください。

以下は、Lambda のイベントフィルタリングで使用できるすべての比較演算子の概要です。

Comparison operator (比較演算子)	例	ルール構文
Null	UserId が Null	"UserID": [null]
空	LastName が空白	"LastName": [""]
等しい	Name が「Alice」	"Name": ["Alice"]
等しい (大文字と小文字を区別しない)	Name が「Alice」	"Name": [{ "equals-ignore-case": "alice" }]
And	Location が「New York」、および Day が「Monday」	"Location": ["New York"], "Day": ["Monday"]
または	PaymentType が「Credit」または「Debit」	"PaymentType": ["Credit", "Debit"]
Or (複数フィールド)	Location が「New York」、または Day が「Monday」	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
以外	Weather が「Raining」以外	"Weather": [{ "anything-but": ["Raining"] }]
数値 (等しい)	Price が 100	"Price": [{ "numeric": ["=", 100] }]
数値 (範囲)	Price が 10 より大きく 20 以下	"Price": [{ "numeric": [">", 10, "<=", 20] }]
存在する	ProductName が存在	"ProductName": [{ "exists": true }]
存在しない	ProductName が存在しない	"ProductName": [{ "exists": false }]
で始まる	Region が US にある	"Region": [{"prefix": "us-" }]

Comparison operator (比較演算子)	例	ルール構文
で終わる	FileName は.png 拡張子で終わります。	"FileName": [{ "suffix": ".png" }]

> **Note**
>
> EventBridge と同様に、Lambda は文字列に対して文字単位の厳密な一致を使用し、大文字変換やその他の文字列の正規化は行いません。数値の場合、Lambda は文字列表現も使用します。たとえば、300、300.0、3.0e2 は等しいとはみなされません。

EXISTS 演算子は、イベントソース JSON のリーフノードでのみ機能することに注意してください。中間ノードとは一致しません。例えば、次の JSON では "address" は中間ノードであるため、{ "person": { "address": [{ "exists": true }] } }" のフィルターパターンで一致するものが見つかりません。

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

イベントソースマッピングへのフィルター条件のアタッチ (コンソール)

Lambda コンソールを使用してフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下の手順を実行します。

フィルター条件が設定された新しいイベントソースマッピングを作成する (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. イベントソースマッピングを作成する関数の名前を選択します。

3. [Function overview] (関数の概要) で [Add trigger] (トリガーを追加) をクリックします。
4. [トリガーの設定] で、イベントフィルタリングをサポートするトリガータイプを選択します。サポートされているサービスのリストについては、このページの冒頭にあるリストを参照してください。
5. [追加の設定] を展開します。
6. [フィルター条件] で [追加] を選択してから、フィルターを定義して入力します。たとえば、次の内容を入力できます。

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   これは、フィールド `Metadata` が 1 または 2 に等しいレコードのみを処理するように Lambda に指示します。引き続き [追加] を選択し、最大許容数までフィルターを追加できます。
7. フィルターの追加を完了したら、[保存] を選択します。

コンソールを使用してフィルター条件を入力するとき、フィルターパターンのみを入力し、`Pattern` キーまたはエスケープの引用符を入力する必要はありません。上記の手順のステップ 6 では、`{ "Metadata" : [ 1, 2 ] }` は次の `FilterCriteria` に対応します。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

コンソールでイベントソースマッピングを作成すると、トリガーの詳細にフォーマットされた `FilterCriteria` が表示されます。コンソールを使用してイベントフィルターを作成するその他の例については、異なる AWS のサービス を持つフィルターの使用 を参照してください。

イベントソースマッピングへのフィルター条件のアタッチ (AWS CLI)

イベントソースマッピングに以下の `FilterCriteria` を設定するとします。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
```

```
      }
   ]
}
```

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

この「create-event-source-mapping」コマンドは、指定された FilterCriteria を持つ関数 my-function の新しい Amazon SQS イベントソースマッピングを作成します。

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

イベントソースマッピングを更新するには、その UUID が必要であることに注意してください。UUID は「list-event-source-mappings」コールから取得できます。Lambda は、「create-event-source-mapping」の CLI レスポンスでも UUID を返します。

イベントソースからフィルター条件を削除するには、空の FilterCriteria オブジェクトを持つ次の「update-event-source-mapping」コマンドを実行できます。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

AWS CLI を使用してイベントフィルターを作成するその他の例については、異なる AWS のサービス を持つフィルターの使用 を参照してください。

イベントソースマッピングへのフィルター条件のアタッチ (AWS SAM)

以下のフィルタ条件を使用するように AWS SAM 内のイベントソースを設定したいとします。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

これらのフィルタ条件をイベントソースマッピングに追加するには、イベントソースの YAML テンプレートに以下のスニペットを挿入します。

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

イベントソースマッピング用の AWS SAM テンプレートの作成と設定に関する詳細については、「AWS SAM デベロッパーガイド」の「EventSource」セクションを参照してください。AWS SAM テンプレートを使用してイベントフィルターを作成するその他の例については、異なる AWS のサービス を持つフィルターの使用 を参照してください。

異なる AWS のサービス を持つフィルターの使用

イベントソースの種類が異なれば、データフィールドに使用するキー値も異なります。データプロパティをフィルタリングするには、フィルターのパターンに正しいキーが使われていることを確認してください。次の表では、サポートされている各 AWS のサービス のフィルタリングキーが示されます。

AWS のサービス	フィルタリングキー
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
セルフマネージド Apache Kafka	`value`
Amazon SQS	`body`

次のセクションでは、さまざまなタイプのイベントソースにおけるフィルターパターンの例が示されます。サポートされている受信データ形式の定義およびサポートされている各サービスのフィルターパターン本体形式も提供します。

DynamoDB でフィルタリング

プライマリキー CustomerName、属性 AccountManager、属性 PaymentTerms を含む DynamoDB テーブルがあるとします。次の内容では、DynamoDB テーブルのストリームからのレコード例が示されています。

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              },
              "NewImage": {
                  "AccountManager": {
                      "S": "Pat Candella"
                  },
                  "PaymentTerms": {
                      "S": "60 days"
                  },
                  "CustomerName": {
                      "S": "AnyCompany Industries"
                  }
              },
              "SequenceNumber": "111",
              "SizeBytes": 26,
              "StreamViewType": "NEW_IMAGE"
          }
      }
  }
```

DynamoDB テーブルのキーおよび属性値に基づいてフィルタリングするには、レコードで dynamodb キーを使用します。次のセクションでは、さまざまなフィルタータイプの例を示します。

テーブルキーでフィルタリングする

プライマリキー CustomerName が「AnyCompany Industries」のレコードのみを関数で処理するとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
        }
     ]
 }
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
                "S": [ "AnyCompany Industries" ]
                }
            }
        }
 }
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

テーブル属性でフィルタリングする

DynamoDB を使用すると、NewImage および OldImage キーを使用して属性値をフィルタリングすることもできます。最新のテーブル画像の AccountManager 属性が「Pat Candella」または「Shirley Rodriguez」のレコードをフィルタリングするとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

ブール式でフィルタリングする

ブール型 AND 式を使用してフィルターを作成することもできます。これらの式には、テーブルの主パラメータと属性パラメータの両方を含めることができます。AccountManager の NewImage の値が「Pat Candella」で、OldImage の値が「Terry Whitlock」であるレコードをフィルタリングするとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
```

```
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB イベントフィルタリングは、数値演算子 (数値等式および数値範囲) の使用をサポートしていません。テーブルの項目が数値として保存されている場合でも、これらのパラメータは JSON レコードオブジェクトの文字列に変換されます。

DynamoDB で EXISTS 演算子を使用する

DynamoDB の JSON イベントオブジェクトは構造化されているため、EXISTS 演算子の使用には特別な注意が必要です。EXISTS 演算子はイベント JSON のリーフノードでのみ機能するため、フィルターパターンが EXISTS を使用して中間ノードをテストしても機能しません。次の DynamoDB テーブルの項目を考慮します。

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
```

```
  }
}
```

次のように、"Organizations" を含むイベントをテストするフィルターパターンを作成できます。

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

ただし、"Organizations" はリーフノードではないため、このフィルターパターンに一致するものは返されません。次の例では、EXISTS 演算子を適切に使用して目的のフィルターパターンを作成する方法を示しています。

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

DynamoDB フィルタリングの JSON 形式

DynamoDB ソースのイベントを適切にフィルタリングするには、データフィールドおよびデータフィールド (dynamodb) のフィルター条件の両方が有効な JSON 形式である必要があります。フィールドのどちらかが有効な JSON 形式ではない場合、Lambda はメッセージをドロップするか、例外をスローします。以下は、特定の動作を要約した表です。

着信データの形式	データプロパティのフィルターパターンの形式	結果として生じるアクション
有効な JSON	有効な JSON	Lambda がフィルター条件に基づいてフィルタリングを実行します。
有効な JSON	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	JSON 以外	Lambda がイベントソースマッピングの作成または更新時に例外をスローします。データプロパティのフィルターパ

着信データの形式	データプロパティのフィルターパターンの形式	結果として生じるアクション
		ターンは、有効な JSON 形式である必要があります。
JSON 以外	有効な JSON	Lambda がレコードをドロップします。
JSON 以外	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
JSON 以外	JSON 以外	Lambda がイベントソースマッピングの作成または更新時に例外をスローします。データプロパティのフィルターパターンは、有効な JSON 形式である必要があります。

Kinesis でフィルタリング

プロデューサーが JSON 形式のデータを Kinesis データストリームに入力するとします。レコードの例は次のようになり、data フィールドで JSON データが Base64 でエンコードされた文字列に変換されます。

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
```

```
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

プロデューサーがストリームに入力するデータが有効な JSON である限り、イベントフィルタリングを使用して data キーを使用するレコードをフィルタリングできます。プロデューサーが次の JSON 形式でレコードを Kinesis ストリームに入力するとします。

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

注文タイプが「購入」のレコードのみをフィルタリングするには、FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

Kinesis ソースからイベントを適切にフィルタリングするには、データフィールドおよびデータフィールドのフィルター条件の両方が有効な JSON 形式である必要があります。フィールドのどち

らかが有効な JSON 形式ではない場合、Lambda はメッセージをドロップするか、例外をスローします。以下は、特定の動作を要約した表です。

着信データの形式	データプロパティのフィルターパターンの形式	結果として生じるアクション
有効な JSON	有効な JSON	Lambda がフィルター条件に基づいてフィルタリングを実行します。
有効な JSON	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	JSON 以外	Lambda がイベントソースマッピングの作成または更新時に例外をスローします。データプロパティのフィルターパターンは、有効な JSON 形式である必要があります。
JSON 以外	有効な JSON	Lambda がレコードをドロップします。
JSON 以外	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
JSON 以外	JSON 以外	Lambda がイベントソースマッピングの作成または更新時に例外をスローします。データプロパティのフィルターパターンは、有効な JSON 形式である必要があります。

Kinesis 集約レコードのフィルタリング

Kinesis を使用すると、複数のレコードを 1 つの Kinesis データストリームレコードに集約し、データスループットを増加させることができます。Lambda は、Kinesis 「拡張ファンアウト」を使用する場合に限り、集約レコードにフィルター条件を適用できます。標準 Kinesis による集約レコードのフィルタリングはサポートされていません。拡張ファンアウトを使用するときは、Kinesis 専用スループットコンシューマーが Lambda 関数のトリガーとして機能するように設定します。次に、Lambda は集約されたレコードをフィルタリングし、フィルター条件を満たすレコードのみを渡します。

Kinesis レコード集約の詳細については、「Kinesis プロデューサーライブラリ (KPL) のキーコンセプト」ページの「集約」セクションを参照してください。Kinesis 拡張ファンアウトを用いた Lambda の使用に関する詳細については、「AWS コンピュートブログ」の「Amazon Kinesis Data Streams 拡張ファンアウトおよび AWS Lambda でのリアルタイムストリーム処理パフォーマンスの向上」を参照してください。

Amazon MQ でフィルタリング

Amazon MQ メッセージキューには、有効な JSON 形式またはプレーン文字列でメッセージが含まれているとします。レコードの例は次のようになり、`data` フィールドでデータが Base64 でエンコードされた文字列に変換されます。

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
west-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
```

```
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  49
                ]
            },
            "header2": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
```

```
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

Active MQ および Rabbit MQ ブローカーの両方では、イベントフィルタリングを使用して `data` キーを使用するレコードをフィルタリングできます。Amazon MQ キューに次の JSON 形式のメッセージが含まれているとします。

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

`timeout` フィールドが 0 より大きいレコードのみをフィルタリングするには、`FilterCriteria` オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの `Pattern` の値を記載しています。

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
```

```
        }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従い、[フィルター条件] に次の文字列を入力します。

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

Amazon MQ を使用すると、メッセージがプレーン文字列のレコードをフィルタリングすることもできます。メッセージが「Result:」で始まるレコードのみを処理するとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ メッセージは UTF-8 でエンコードされた文字列 (プレーン文字列または JSON 形式) である必要があります。これは、Lambda がフィルター条件を適用する前に Amazon MQ のバイト配列を UTF-8 にデコードするためです。メッセージが UTF-16 や ASCII などの別のエンコーディングを使用している場合、またはメッセージ形式が `FilterCriteria` 形式と一致しない場合、Lambda はメタデータフィルターのみを処理します。以下は、特定の動作を要約した表です。

着信メッセージの形式	メッセージプロパティのフィルターパターン形式	結果として生じるアクション
プレーン文字列	プレーン文字列	Lambda がフィルター条件に基づいてフィルタリングを実行します。
プレーン文字列	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。

着信メッセージの形式	メッセージプロパティのフィルターパターン形式	結果として生じるアクション
プレーン文字列	有効な JSON	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	プレーン文字列	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	有効な JSON	Lambda がフィルター条件に基づいてフィルタリングを実行します。
UTF-8 以外でエンコードされた文字	JSON、プレーン文字列、またはパターンなし	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。

Amazon MSK およびセルフマネージド Apache Kafka でフィルタリング

プロデューサーが Amazon MSK またはセルフマネージド Apache Kafka クラスターのトピックに、有効な JSON 形式またはプレーン文字列として、メッセージを書き込むとします。レコードの例は次のようになり、`value` フィールドでメッセージが Base64 でエンコードされた文字列に変換されます。

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
```

```
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Apache Kafka プロデューサーが次の JSON 形式でトピックにメッセージを書き込むとします。

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

value キーを使用してレコードをフィルタリングできます。device_ID が AB の文字で始まるレコードのみをフィルタリングするとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

Amazon MSK およびセルフマネージド Apache Kafka では、メッセージがプレーン文字列のレコードをフィルタリングすることもできます。文字列が「error」を含むメッセージを無視するとします。`FilterCriteria` オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの `Pattern` の値を記載しています。

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Amazon MSK およびセルフマネージド Apache Kafka メッセージは UTF-8 でエンコードされた文字列 (プレーン文字列または JSON 形式) である必要があります。これは、Lambda がフィルター条件を適用する前に Amazon MSK のバイト配列を UTF-8 にデコードするためです。メッセージが UTF-16 や ASCII などの別のエンコーディングを使用している場合、またはメッセージ形式が `FilterCriteria` 形式と一致しない場合、Lambda はメタデータフィルターのみを処理します。以下は、特定の動作を要約した表です。

着信メッセージの形式	メッセージプロパティのフィルターパターン形式	結果として生じるアクション
プレーン文字列	プレーン文字列	Lambda がフィルター条件に基づいてフィルタリングを実行します。
プレーン文字列	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。

着信メッセージの形式	メッセージプロパティのフィルターパターン形式	結果として生じるアクション
プレーン文字列	有効な JSON	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	プレーン文字列	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	有効な JSON	Lambda がフィルター条件に基づいてフィルタリングを実行します。
UTF-8 以外でエンコードされた文字	JSON、プレーン文字列、またはパターンなし	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。

Amazon SQS でフィルタリング

Amazon SQS キューに次の JSON 形式のメッセージが含まれているとします。

```
{
    "RecordNumber": 0000,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

このキューのレコード例は次のようになります。

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 0000,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

Amazon SQS メッセージの内容に基づいてフィルタリングするには、Amazon SQS メッセージレコードの body キーを使用します。Amazon SQS メッセージの RequestCode が「BBBB」のレコードのみを処理するとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

関数が、RecordNumber が 9999 を超えるレコードのみを処理するとします。FilterCriteria オブジェクトは次のようになります。

```
{
    "Filters": [
```

```
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

以下は、わかりやすくするためにプレーン JSON で展開したフィルターの Pattern の値を記載しています。

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

コンソール、AWS CLI、または AWS SAM テンプレートを使用してフィルターを追加できます。

Console

コンソールを使用してこのフィルターを追加するには、イベントソースマッピングへのフィルター条件のアタッチ (コンソール) の指示に従って [フィルター条件] に次の文字列を入力します。

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

AWS Command Line Interface (AWS CLI) を使用してこれらのフィルター条件を持つ新しいイベントソースマッピングを作成するには、以下のコマンドを実行します。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

これらのフィルター条件を既存のイベントソースマッピングに追加するには、次のコマンドを実行します。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

AWS SAM を使用してこのフィルターを追加するには、イベントソースの YAML テンプレートに次のスニペットを追加します。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

Amazon SQS では、メッセージ本文は任意の文字列にすることができます。body が有効な JSON フォーマットであることが FilterCriteria で想定されている場合は、これが問題になる可能性があります。逆の場合も同様です。着信メッセージの本文が JSON 形式であっても、フィルター条件が body をプレーン文字列であると想定する場合、意図しない動作が発生する可能性があります。

この問題を回避するには、FilterCriteria の本文の形式がキューで受信するメッセージで想定する body の形式と一致することを確認してください。メッセージをフィルタリングする前に、Lambda は着信メッセージの本文の形式および body のフィルターパターンの形式を自動的に評価します。一致しない場合、Lambda はメッセージを除外します。この評価のまとめは、以下の表のとおりです。

着信メッセージの **body** の形式	フィルターパターンの **body** の形式	結果として生じるアクション
プレーン文字列	プレーン文字列	Lambda がフィルター条件に基づいてフィルタリングを実行します。
プレーン文字列	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。

着信メッセージの **body** の形式	フィルターパターンの **body** の形式	結果として生じるアクション
プレーン文字列	有効な JSON	Lambda はメッセージを除外します。
有効な JSON	プレーン文字列	Lambda はメッセージを除外します。
有効な JSON	データプロパティのフィルターパターンがない	Lambda がフィルター条件に基づいて (他のメタデータプロパティのみを) フィルタリングします。
有効な JSON	有効な JSON	Lambda がフィルター条件に基づいてフィルタリングを実行します。

コンソールでの Lambda 関数のテスト

コンソールで Lambda 関数をテストするには、テストイベントを使用して関数を呼び出します。テストイベントとは、関数への JSON 入力です。関数が入力を必要としない場合、イベントは空のドキュメント ({}) にすることができます。

コンソールでテストを実行すると、Lambda はテストイベントと同期して関数を呼び出します。関数ランタイムは イベント JSON をオブジェクトに変換し、コードのハンドラーメソッドに渡して処理します。

> **テストイベントを作成する**
>
> コンソールでテストする前に、プライベートまたは共有可能なテストイベントを作成する必要があります。

テストイベントで関数を呼び出す

関数をテストするには

1. Lambda コンソールの [関数ページ] を開きます。
2. テストする関数の名前を選択します。
3. [Test] (テスト) タブを選択します。
4. [テストイベント] で、[新しいイベントを作成] または [保存されたイベント] を選択し、使用する保存済みのイベントを選択します。
5. 必要に応じて、イベント JSON の [テンプレート] を選択します。
6. [テスト] を選択します。
7. テスト結果を確認するには、[Execution result] (実行結果) で、[Details] (詳細) を展開します。

テストイベントを保存せずに関数を呼び出すには、保存する前に [Test] (テスト) を選択します。これにより、Lambda がセッション期間中だけ保持し、保存されないテストイベントが作成されます。

また、[Code] (コード) タブで保存されたテストイベントおよび保存されないテストイベントにアクセスすることもできます。そこから、[Test] (テスト) を選択し、テストイベントを選択します。

プライベートテストイベントを作成する

プライベートテストイベントは、イベント作成者のみが使用でき、追加のアクセス許可は必要ありません。関数ごとに、最大 10 個のプライベートテストイベントを作成し保存できます。

プライベートテストイベントを作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. テストする関数の名前を選択します。
3. [Test] (テスト) タブを選択します。
4. [Test event] (テストイベント) で、次の作業を行います。
    a. [Template] (テンプレート) を選択します。
    b. テストの [Name] (名前) を入力します。
    c. テキスト入力ボックスに、JSON テストイベントを入力します。
    d. [Event sharing settings] (イベント共有設定) で、[Private] (プライベート) を選択します。
5. [変更の保存] をクリックします。

また、[Code] (コード) タブで新しいテストイベントを作成することもできます。そこから、[Test] (テスト)、[Configure test event] (テストイベントの設定) を選択します。

共有可能なテストイベントを作成する

共有可能なテストイベントは、同じ AWS アカウント内の他のユーザーと共有できるテストイベントです。他のユーザーの共有可能なテストイベントを編集し、それを使用して関数を呼び出すことができます。

Lambda は、共有可能なテストイベントをスキーマとして という名前の Amazon EventBridge (CloudWatch Events) スキーマレジストリに保存しますlambda-testevent-schemas。Lambda はこのレジストリを使用して、作成した共有可能なテストイベントを保存したり呼び出したりするので、このレジストリの編集や lambda-testevent-schemas 名を使用したレジストリの作成はしないようにお勧めします。

共有可能なテストイベントを表示、共有、編集するには、以下の EventBridge (CloudWatch イベント) スキーマレジストリ API オペレーションのすべてに対するアクセス許可が必要です。

- schemas.CreateRegistry
- schemas.CreateSchema

- schemas.DeleteSchema
- schemas.DeleteSchemaVersion
- schemas.DescribeRegistry
- schemas.DescribeSchema
- schemas.GetDiscoveredSchema
- schemas.ListSchemaVersions
- schemas.UpdateSchema

共有可能なテストイベントに加えた編集を保存すると、そのイベントが上書きされることに注意してください。

共有可能なテストイベントを作成、編集、または表示できない場合は、アカウントにこれらのオペレーションに必要なアクセス許可があることを確認してください。必要なアクセス許可は持っていても、共有可能なテストイベントにアクセスできない場合は、 EventBridge （ CloudWatch Events) レジストリへのアクセスを制限する可能性のあるリソースベースのポリシーがないか確認します。

共有可能なテストイベントを作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. テストする関数の名前を選択します。
3. [Test] (テスト) タブを選択します。
4. [Test event] (テストイベント) で、次の作業を行います。
   a. [Template] (テンプレート) を選択します。
   b. テストの [Name] (名前) を入力します。
   c. テキスト入力ボックスに、JSON テストイベントを入力します。
   d. [Event sharing settings] (イベント共有設定) で、[Shareable] (共有可能) を選択します。
5. [変更の保存] をクリックします。

AWS Serverless Application Model で共有可能なテストイベントを使用する。

AWS SAM を使用して共有可能なテストイベントを呼び出すことができます。「AWS Serverless Application Model デベロッパーガイド」の「sam remote test-event」を参照してください。

共有可能なテストイベントスキーマの削除

共有可能なテストイベントを削除すると、Lambda はそれらを `lambda-testevent-schemas` レジストリから削除します。レジストリから最後の共有可能なテストイベントを削除すると、Lambda はレジストリを削除します。

関数を削除しても、Lambda は関連付けられた共有可能なテストイベントスキーマを削除しません。これらのリソースはEventBridge 、 (CloudWatch イベント) コンソールから手動でクリーンアップする必要があります。

Lambda 関数の状態

関数を呼び出す準備ができたことを示すために、Lambda はすべての関数の関数設定に状態フィールドを含めます。State は、関数の現在の関数のステータスに関する情報 (関数を正常に呼び出すことができるかどうかなど) を提供します。関数の状態は、関数呼び出しの動作や関数によるコードの実行方法を変更するものではありません。関数の状態は以下のとおりです。

- Pending — Lambda は関数を作成した後、関数の状態を保留に設定します。保留状態の場合、Lambda は関数のリソース (VPC や EFS リソースなど) の作成または設定を試みます。Lambda は、保留状態の関数を呼び出しません。関数に対する呼び出しやその他の API アクションは、すべて失敗します。
- Active — Lambda がリソース設定とプロビジョニングを完了した後、関数はアクティブ状態に移行します。関数は、アクティブな間のみ正常に呼び出すことができます。
- Failed — リソース設定またはプロビジョニングでエラーが発生したことを示します。
- Inactive — Lambda 用に設定された外部リソースを再利用するのに十分な時間アイドル状態を維持した関数は非アクティブになります。非アクティブな関数を呼び出そうとすると、呼び出しは失敗し、関数リソースが再作成されるまで、Lambda は関数を保留状態に設定します。Lambda がリソースの再作成に失敗した場合、関数は非アクティブ状態に戻ります。関数が非アクティブ状態のままである場合、関数の StatusCode 属性と StatusCodeReason 属性を参照して、さらにトラブルシューティングを行ってください。エラーを解決し、関数を再デプロイしてアクティブ状態に復元する必要がある場合があります。

SDK ベースのオートメーションワークフローを使用している場合、または Lambda のサービス API を直接呼び出す場合は、呼び出し前に関数の状態をチェックして、関数がアクティブであることを確認します。この操作は、Lambda API アクションの GetFunction を使用するか、AWS SDK for Java 2.0 を使用してウェーターを設定することによって実行することができます。

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

以下の出力が表示されます。

```
[
 "Active",
 "Successful"
]
```

関数の作成が保留中の場合、以下のオペレーションは失敗します。

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

更新中の関数の状態

Lambda は、LastUpdateStatus 属性を使用して、更新中の関数の追加コンテキストを提供します。これは、次のいずれかのステータスです。

- InProgress — 既存の関数で更新が進行中です。関数の更新が進行している間、呼び出しは関数の以前のコードと設定に移動します。
- Successful — 更新が完了しました。Lambda が更新を完了すると、さらに更新されるまでこの状態を維持します。
- Failed — 関数の更新が失敗しました。Lambda は更新を中止します。関数の以前のコードと設定を引き続き使用できます。

Example

更新中の関数の get-function-configuration の結果を以下に示します。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
```

```
    "LastUpdateStatus": "InProgress",
    ...
}
```

FunctionConfiguration には、更新の問題のトラブルシューティングに役立つ他の 2 つの属性 (LastUpdateStatusReason と LastUpdateStatusReasonCode) があります。

非同期更新の進行中、以下のオペレーションは失敗します。

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource

Lambda での再試行動作について

関数を直接呼び出すときは、関数コードに関連するエラーを処理する方法を決定します。Lambda は、ユーザーに代わってこれらのタイプのエラーを自動的に再試行しません。再試行するには、関数を手動で再呼び出しするか、失敗したイベントをデバッグのためにキューに送信するか、エラーを無視します。関数コードが部分的にまたは完全に実行されている可能性もあれば、まったく実行されない可能性もあります。再試行する場合は、関数コードが重複したトランザクションや望ましくない副作用を引き起こすことなく同じイベントを複数回処理できるようにしてください。

関数を間接的に呼び出す際、呼び出し元やリクエストが遭遇するすべてのサービスの再試行動作について把握する必要があります。これには、以下のシナリオが含まれます。

- 非同期呼び出し - Lambda は、関数エラーを 2 回再試行します。関数にすべての受信リクエストを処理する十分なキャパシティがない場合、関数に送信されるまで、イベントはキューの中に数時間または数日間保持される可能性があります。正常に処理できなかったイベントを把握するために、デッドレターキューを設定できます。詳細については、「非同期呼び出し」を参照してください。
- イベントソースマッピング - ストリームから読み取るイベントソースマッピングは、項目のバッチ全てに対して再試行を実施します。繰り返されるエラーは、そのエラーが解決されるか、項目が期限切れになるまで、影響を受けるシャードの処理を妨げます。停止しているシャードを検出するには、Iterator Age メトリクスをモニタすることができます。

  キューから読み取るイベントソースマッピングについては、ソースキューの再試行ポリシーおよび可視性タイムアウトを設定することで、失敗したイベントの送信先と再試行の間の時間の長さを決定します。詳細については、Lambda がストリームおよびキューベースのイベントソースからのレコードを処理する方法 および 他の AWS サービスからのイベントを使用した Lambda の呼び出し のサービス固有のドキュメントを参照してください。
- AWS のサービス - AWS のサービスでは、同期的または非同期的に関数を呼び出す場合があります。同期呼び出しの場合、サービスは再試行するかどうかを決定します。例えば、Amazon S3 バッチオペレーションは、Lambda 関数が `TemporaryFailure` 応答コードを返す場合、オペレーションを再試行します。プロキシがアップストリームユーザーまたはクライアントからリクエストするサービスは、再試行戦略を持っているか、またはリクエスタにエラー応答をリレーする場合があります。例えば、API Gateway は常にエラー応答をリレーしてリクエスタに返します。

  非同期呼び出しの場合、動作は関数を非同期的に呼び出した場合と同じです。詳細については、他の AWS サービスからのイベントを使用した Lambda の呼び出し のサービス固有のトピックおよび呼び出しサービスのドキュメントを参照してください。

- 他のアカウントやクライアント - 他のアカウントへのアクセス権限を付与する場合、関数を呼び出すために設定できるサービスやリソースの制限にリソースベースのポリシーを使用することができます。関数を過負荷から守るためには、Amazon API Gateway を使用して、関数の前に API レイヤーを設定することを考慮してください。

Lambda アプリケーションのエラーに対応できるよう、Lambda では、Amazon CloudWatch や AWS X-Ray などのサービスを統合しています。ログやメトリクス、アラーム、関数コード内の問題を迅速に検出および特定するトレーシング、API またはアプリケーションをサポートするその他のリソースを組み合わせて使用することができます。詳細については、「Lambda 関数のモニタリングおよびトラブルシューティング」を参照してください。

Lambda 再帰ループ検出を使用した無限ループの防止

関数を呼び出すのと同じサービスまたはリソースに出力するように Lambda 関数を設定すると、無限再帰ループが作成される可能性があります。例えば、Lambda 関数が Amazon Simple Queue Service (Amazon SQS) キューにメッセージを書き込み、その書き込まれたキューによって同じ関数が呼び出される場合があります。この呼び出しにより、関数はキューに別のメッセージを書き込み、キューによって関数が再び呼び出されます。

意図しない再帰ループが発生すると、AWS アカウント に予期しない料金が請求される可能性があります。ループが原因で Lambda がスケールし、アカウントで利用可能なすべての同時実行性が使用されることもあります。意図しないループの影響を減らすために、Lambda は特定のタイプの再帰ループを発生後すぐに検出することができます。Lambda が再帰ループを検出すると、関数の呼び出しを停止し、ユーザーに通知します。

意図的に再帰パターンを使用するようデザインされている場合は、Lambda 再帰ループ検出の無効化をリクエストできます。この変更をリクエストするには、AWS Support にお問い合わせください。

> **Important**
>
> Lambda 関数を使用して意図的に AWS 関数を呼び出すのと同じリソースにデータを書き戻すように設計している場合は、AWS アカウント に予期しない料金が請求されないように注意し、適切なガードレールを実装してください。再帰呼び出しパターンを使用する際のベストプラクティスの詳細については、Serverless Land の「Lambda 関数が暴走する原因となる再帰パターン」を参照してください。

セクション

再帰ループ検出を理解する

Lambda の再帰ループ検出は、イベントを追跡することで動作します。Lambda はイベント駆動型のコンピューティングサービスで、特定のイベントが発生すると関数コードを実行します。例えば、ア

イテムが Amazon SQS キューまたは Amazon Simple Notification Service (Amazon SNS) トピックに追加された場合などが挙げられます。Lambda は、システム状態の変化に関する情報を含んだイベントを JSON オブジェクトとして関数に渡します。イベントによって関数が実行されるとき、これを呼び出しと呼びます。

再帰ループを検出するために、Lambda は AWS X-Ray トレースヘッダーを使用します。再帰ループ検出をサポートしている AWS のサービス が Lambda にイベントを送信すると、それらのイベントには自動的にメタデータの注釈が付けられます。Lambda 関数がこれらのイベントの 1 つを、サポートされている別の AWS のサービス に対してサポートされているバージョンの AWS SDK を使用して書き込むと、このメタデータが更新されます。更新されたメタデータには、イベントによって関数が呼び出された回数が含まれます。

> **Note**
>
> この機能を動作させるのに、X-Ray アクティブトレーシングを有効にする必要はありません。再帰ループ検出は、AWS をお使いのすべてのお客様に対してデフォルトでオンになっています。この機能は無料で使用できます。

リクエストチェーンとは、同じトリガーイベントによって発生する Lambda 呼び出しのシーケンスです。例えば、ある Amazon SQS キューが Lambda 関数を呼び出すとします。呼び出された Lambda 関数は、次に、処理されたイベントを同じ Amazon SQS キューに送り返し、そこで関数が再度呼び出されます。この例では、それぞれの関数呼び出しは同じリクエストチェーンに分類されます。

同じリクエストチェーンで関数が 16 回を超えて呼び出された場合、Lambda はそのリクエストチェーン内の次の関数呼び出しを自動的に停止し、ユーザーに通知します。関数が複数のトリガーで構成されている場合、他のトリガーからの呼び出しには影響しません。

> **Note**
>
> ソースキューの再処理ポリシーの `maxReceiveCount` 設定が 16 より大きい場合、Lambda 再帰保護は、再帰ループが検出され終了された後に Amazon SQS がメッセージを再試行することを防止しません。Lambda は、再帰ループを検出し、それ以降の呼び出しを中止すると、イベントソースマッピングに `RecursiveInvocationException` を返します。メッセージの `receiveCount` 値を増加します。Amazon SQS が `maxReceiveCount` の超過を判定して設定されたデッドレターキューにメッセージを送信するまで、Lambda はメッセージを再試行し続け、関数の呼び出しをブロックし続けます。

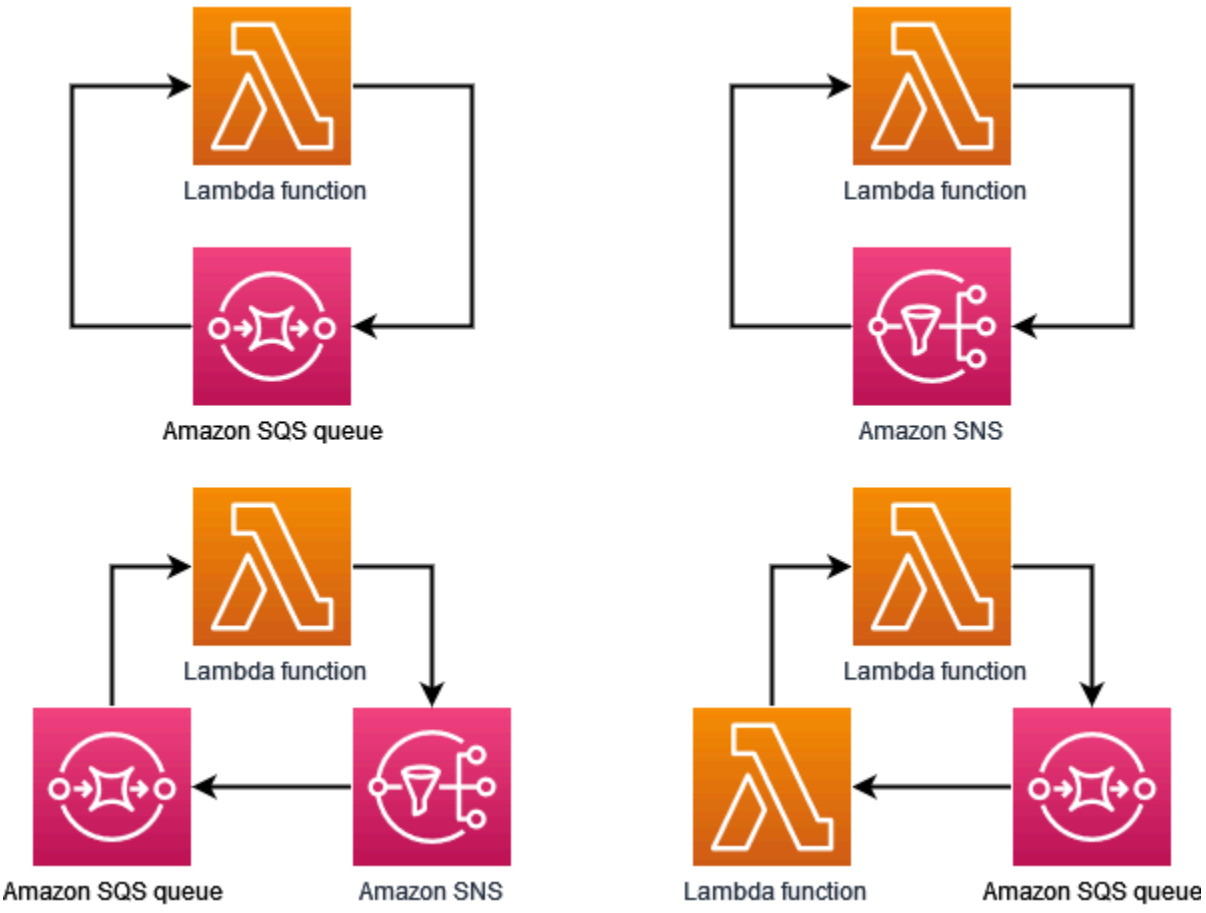
障害発生時の宛先またはデッドレターキューが関数に設定されている場合、Lambda はその停止した呼び出しからのイベントを指定された宛先またはデッドレターキューにも送信します。関数に宛先またはデッドレターキューを設定する場合は、関数がイベントトリガーまたはイベントソースマッピングとしても使用する Amazon SNS トピックまたは Amazon SQS キューを使用しないようにしてください。関数を呼び出すのと同じリソースにイベントを送信すると、別の再帰ループを作成できます。

サポートされている AWS のサービス および SDK

Lambda が検出できるのは、サポートされている特定の AWS のサービス を含む再帰ループのみです。再帰ループを検出するには、関数はサポートされている AWS SDK のいずれかを使用する必要もあります。

サポートされている AWS のサービス

Lambda は現在、関数、Amazon SQS、Amazon SNS の間の再帰ループを検出します。Lambda は、Lambda 関数のみで構成されるループも検出します。これらの関数は、同期的に、または非同期的にお互いを呼び出す可能性があります。次の図に、Lambda が検出できるループの例を示します。



Amazon DynamoDB や Amazon Simple Storage Service (Amazon S3) などの別の AWS のサービス がループの一部を形成してる場合、現時点では Lambda はそのループを検出して停止することはできません。

現時点では、Lambda は Amazon SQS と Amazon SNS に関連する再帰ループのみを検出するため、他の AWS のサービス が関与するループによって Lambda 関数が意図しない形で使用される可能性があります。

ユーザーの AWS アカウント に予期しない料金が請求されるのを防ぐため、Amazon CloudWatch アラームを設定して、通常とは異なる使用パターンが警告されるようにすることをお勧めします。例えば、Lambda 関数の同時実行数や呼び出し回数の急増が検知された場合に、CloudWatch を設定しておくとユーザーに通知が送信されます。また、請求アラームを設定しておくことで、アカウントでの支出が指定したしきい値を超えたときにも通知を受け取ることができます。AWS Cost Anomaly Detection を使用すると、通常とは異なる請求パターンがあった場合に警告を受け取ることができます。

サポートされている AWS SDK

Lambda が再帰ループを検出するには、関数で次のバージョンまたはそれ以降の SDK を使用する必要があります。

ランタイム	最低限必要な AWS SDK バージョン
Node.js	2.1147.0 (SDK バージョン 2) 3.105.0 (SDK バージョン 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 および Java 11	1.12.200 (SDK バージョン 1) 2.17.135 (SDK バージョン 2)
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0

ランタイム	最低限必要な AWS SDK バージョン
Ruby	3.134.0
PHP	3.232.0

Python や Node.js などの一部の Lambda ランタイムには、あるバージョンの AWS SDK が含まれています。関数のランタイムに含まれている SDK のバージョンが必要最低限よりも低い場合は、サポートされているバージョンの SDK を関数のデプロイパッケージに追加することができます。Lambda レイヤーを使用して、サポートされている SDK バージョンを関数に追加することもできます。各 Lambda ランタイムに含まれている SDK のリストについては、Lambda ランタイム を参照してください。

Lambda 再帰検出は Lambda Go ランタイムではサポートされていません。

再帰ループ通知

Lambda が再帰ループを停止すると、AWS Health Dashboard やメールで通知が届きます。CloudWatch メトリクスを使用して、Lambda が停止した再帰呼び出しの数をモニタリングすることもできます。

AWS Health Dashboard の通知

Lambda が再帰呼び出しを停止すると、AWS Health Dashboard は、[アカウントヘルス] ページの [未解決の問題と最近の問題] に通知を表示します。Lambda が再帰呼び出しを停止してからこの通知が表示されるまでに、最大 3 時間かかる場合があることに注意してください。AWS Health Dashboard でのアカウントイベントの表示の詳細については、AWS Health ユーザーガイドの「AWS Health ダッシュボードの使用開始 — アカウントの正常性」を参照してください。

E メールアラート

Lambda が関数の再帰呼び出しを初めて停止すると、E メールアラートが送信されます。Lambda は、AWS アカウント の各関数につき 24 時間ごとに最大 1 通のメールを送信します。Lambda から E メール通知が送信された後の 24 時間は、Lambda によって関数の再帰呼び出しが再度停止された場合でも、その関数に関するメールが届きません。Lambda が再帰呼び出しを停止してからこの E メールアラートを受信するまでに、最大 3 時間かかる場合があることに注意してください。

再帰的なループに関する E メールアラートは、Lambda からお客様の AWS アカウント のプライマリアカウント連絡先と代替オペレーション連絡先に送信されます。アカウントのメールアドレスの表示または更新については、AWS 全般のリファレンスの「連絡先情報の更新」を参照してください。

Amazon CloudWatch メトリクス

CloudWatch メトリクスの `RecursiveInvocationsDropped` には、1 回のリクエストチェーンで関数が 16 回を超えて呼び出されたために Lambda が停止した関数の呼び出し回数が記録されます。Lambda は再帰呼び出しを停止するとすぐにこのメトリクスを出力します。このメトリクスを表示するには、「CloudWatch コンソールでメトリクスを表示する」に記載の指示に従い、メトリクス `RecursiveInvocationsDropped` を選択します。

再帰ループ検出通知への対応

同じトリガーイベントによって関数が 16 回を超えて呼び出された場合、Lambda はそのイベントの次の関数呼び出しを停止して再帰ループを中断します。Lambda が中断した再帰ループの再発を防ぐには、以下を実行してください。

- 関数が同時に実行できる回数を 0 に減らすと、以降の呼び出しがすべてスロットリングされます。
- 関数を呼び出すトリガーやイベントソースマッピングを削除または無効にします。
- 関数を呼び出している AWS リソースにイベントを書き戻すコードを特定し、欠陥を修正します。よくある不具合の原因として、変数を使用して関数のイベントソースとターゲットを定義している場合が挙げられます。両方の変数に同じ値が使用されていないことを確認してください。

さらに、Lambda 関数のイベントソースが Amazon SQS キューの場合は、ソースキューにデッドレターキューを設定することを検討してください。

> **Note**
>
> Lambda 関数ではなく、ソースキューのデッドレターキューを設定するようにしてください。関数で設定したデッドレターキューは、イベントソースキューではなく、関数の非同期呼び出しキューに使用されます。

イベントソースが Amazon SNS トピックの場合は、関数に障害発生時の宛先を追加することを検討してください。

関数で利用できる同時実行数をゼロにするには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数の名前を選択します。
3. [スロットル] を選択します。
4. [関数のスロットル] ダイアログボックスで、[確認] を選択します。

関数のトリガーまたはイベントソースマッピングを削除するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数の名前を選択します。
3. [Configuration] タブを選択し、[Triggers] を選択します。
4. [Triggers] で、削除するトリガーまたはイベントソースマッピングを選択し、[Delete] を選択します。
5. [トリガーの削除] ダイアログボックスで、[削除] を選択します。

関数のイベントソースマッピングを無効にするには (AWS CLI)

1. 無効にするイベントソースマッピングの UUID を見つけるには、AWS Command Line Interface (AWS CLI) の list-event-source-mappings コマンドを実行します。

```
aws lambda list-event-source-mappings
```

2. イベントソースマッピングを無効にするには、次の AWS CLI update-event-source-mapping コマンドを実行します。

```
aws lambda update-event-source-mapping --function-name MyFunction \
--uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
```

Lambda 関数 URL

関数 URL は、Lambda 関数のための専用 HTTP エンドポイントです。関数 URL の作成と設定には、Lambda コンソールまたは Lambda API を使用します。関数 URL を作成すると、一意の URL エンドポイントが Lambda により自動的に生成されます。関数 URL を作成した後に、その URL エンドポイントが変更されることはありません。関数 URL のエンドポイントでは、次の形式を使用します。

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 関数 URLsは、アジアパシフィック (ハイデラバード) (ap-south-2) 、アジアパシフィック (メルボルン) ()、ap-southeast-4カナダ西部 (カルガリー) ()、ca-west-1欧州 (スペイン) ()、eu-south-2欧州 (チューリッヒ) ()、eu-central-2イスラエル (テルアビブ) ()、il-central-1中東 (アラブ首長国連邦) () の各リージョンではサポートされていませんme-central-1。

関数 URL はデュアルスタックに対応しており、IPv4 と IPv6 をサポートしています。関数のために URL を設定した後は、ウェブブラウザ、curl、Postman、または任意の HTTP クライアントから HTTP エンドポイントを介して、その関数を呼び出せるようになります。

> **Note**
>
> 関数 URL には、パブリックインターネット経由でしかアクセスできません。Lambda 関数は AWS PrivateLink をサポートしていますが、関数 URL ではサポートされません。

Lambda 関数 URL では、セキュリティとアクセスコントロールのために、リソースベースのポリシーを使用します。また、Cross-Origin Resource Sharing (CORS) 設定オプションを、関数 URL に設定することも可能です。

関数 URL は、任意の関数エイリアス、もしくは $LATEST の未公開な関数バージョンに対して適用できます。関数 URL は、他の関数バージョンに追加することはできません。

トピック

- Lambda 関数 URL の作成と管理

- Lambda 関数 URL へのアクセスの制御
- Lambda 関数 URL の呼び出し
- Lambda 関数 URL のモニタリング
- チュートリアル: 関数 URL を使用する Lambda 関数の作成

Lambda 関数 URL の作成と管理

関数 URL は、Lambda 関数のための専用 HTTP エンドポイントです。関数 URL の作成と設定には、Lambda コンソールまたは Lambda API を使用します。関数 URL を作成すると、一意の URL エンドポイントが Lambda により自動的に生成されます。関数 URL を作成した後に、その URL エンドポイントが変更されることはありません。関数 URL のエンドポイントでは、次の形式を使用します。

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 関数 URL は、アジアパシフィック (ハイデラバード) (`ap-south-2`)、アジアパシフィック (メルボルン) (`ap-southeast-4`)、カナダ西部 (カルガリー) (`ca-west-1`)、欧州 (スペイン) (`eu-south-2`)、欧州 (チューリッヒ) (`eu-central-2`)、イスラエル (テルアビブ) (`il-central-1`)、および中東 (UAE) (`me-central-1`) ではサポートされていません。

次のセクションでは、Lambda コンソール、AWS CLI、および AWS CloudFormation テンプレートを使用して関数 URL を作成し、管理する方法を説明します。

トピック

関数 URL の作成 (コンソール)

コンソールを使用して関数 URL を作成するには、以下の手順に従います。

既存の関数のために関数 URL を作成するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数 URL を作成する関数の名前を選択します。
3. [Configuration] (設定) タブを開き、次に [Function URL] (関数 URL) をクリックします。
4. [Create function URL] (関数 URL を作成) をクリックします。
5. [Auth Type] (認証タイプ) で、[AWS_IAM] または [NONE] (無し) を選択します。関数 URL での認証の詳細については、「アクセスコントロール」を参照してください。
6. (オプション) [Cross-Origin Resource Sharing (CORS)] をクリックした後に、関数 URL の CORS 設定を作成します。CORS の詳細については、クロスオリジンリソース共有 (CORS) を参照してください。
7. [Save] を選択します。

これにより、未公開関数の $LATEST バージョン用として関数 URL が作成されます。関数 URL は、コンソールの [Function overview] (関数の概要) セクションに表示されます。

既存のエイリアスのために関数 URL を作成するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数 URL を作成するエイリアスを持つ関数の名前を選択します。
3. [Aliases] (エイリアス) タブを開き、関数 URL を作成するエイリアスの名前を選択します。
4. [Configuration] (設定) タブを開き、次に [Function URL] (関数 URL) をクリックします。
5. [Create function URL] (関数 URL を作成) をクリックします。
6. [Auth Type] (認証タイプ) で、[AWS_IAM] または [NONE] (無し) を選択します。関数 URL での認証の詳細については、「アクセスコントロール」を参照してください。
7. (オプション) [Cross-Origin Resource Sharing (CORS)] をクリックした後に、関数 URL の CORS 設定を作成します。CORS の詳細については、クロスオリジンリソース共有 (CORS) を参照してください。
8. [Save] を選択します。

これにより、関数エイリアス用に関数 URL が作成されます。関数 URL は、コンソール上で、エイリアスの [Function overview] (関数の概要) セクションに表示されます。

関数 URL を使用して新しい関数を作成するには (コンソール)

関数 URL を使用して新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。

2. [Create function (関数の作成)] を選択します。
3. [基本的な情報] で、以下を実行します。
   a. [Function name] (関数名) に関数の名前を入力します (例: **my-function**)。
   b. [Runtime] (ランタイム) で、希望する言語のランタイム (Node.js 18.x など) を選択します。
   c. [Architecture] (アーキテクチャ) では、「x86_64」または「arm64」のいずれかを選択します。
   d. [Permissions] (アクセス許可) を展開し、新しい実行ロールを作成するか、既存のロールを使用するかを選択します。
4. [Advanced settings] (詳細設定) を展開し、[Function URL] (関数 URL) を選択します。
5. [Auth Type] (認証タイプ) で、[AWS_IAM] または [NONE] (無し) を選択します。関数 URL での認証の詳細については、「アクセスコントロール」を参照してください。
6. (オプション) [Configure cross-origin resource sharing (CORS)] (Cross-Origin Resource Sharing (CORS) の設定) をクリックします。関数の作成時にこのオプションを選択すると、関数 URL はデフォルトで、すべてのオリジンからのリクエストを許可します。関数 URL の CORS 設定は、関数の作成が完了した後に編集できます。CORS の詳細については、クロスオリジンリソース共有 (CORS) を参照してください。
7. [Create function (関数の作成)] を選択します。

これにより、未公開の関数バージョン $LATEST で、関数 URL を持つ新しい関数が作成されます。関数 URL は、コンソールの [Function overview] (関数の概要) セクションに表示されます。

関数 URL の作成 (AWS CLI)

AWS Command Line Interface (AWS CLI) を使用して、既存の Lambda 関数の関数 URL を作成するには、次のコマンドを実行します。

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

これにより、**my-function** 関数の **prod** 修飾子に関数 URL が追加されます。これらの構成パラメータの詳細については、API リファレンスの「CreateFunctionUrlConfig」を参照してください。

> **Note**
>
> AWS CLI を介して関数 URL を作成するには、対象の関数が既に存在している必要があります。

CloudFormation テンプレートへの関数 URL の追加

`AWS::Lambda::Url` リソースを AWS CloudFormation テンプレートに追加するには、次の構文を使用します。

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

パラメータ

- (必須) `AuthType` – 関数 URL で使用する認証のタイプを定義します。これに使用できる値は、`AWS_IAM` または `NONE` です。アクセスを認証されたユーザーのみに制限するには、`AWS_IAM` に設定します。IAM 認証をバイパスし、任意のユーザーが関数にリクエストを送信できるようにするには、`NONE` をセットします。

- (オプション) `Cors` – 関数 URL のための CORS 設定を定義します。CloudFormation 内の `AWS::Lambda::Url` リソースに `Cors` を追加するには、次の構文を使用します。

  Example AWS::Lambda::Url.Cors (JSON)

```
{
  "AllowCredentials" : Boolean,
  "AllowHeaders" : [ String, ... ],
  "AllowMethods" : [ String, ... ],
  "AllowOrigins" : [ String, ... ],
  "ExposeHeaders" : [ String, ... ],
  "MaxAge" : Integer
}
```

  Example AWS::Lambda::Url.Cors (YAML)

```
  AllowCredentials: Boolean
  AllowHeaders:
    - String
  AllowMethods:
    - String
  AllowOrigins:
    - String
  ExposeHeaders:
    - String
  MaxAge: Integer
```

- (オプション) `Qualifier` – エイリアス名。
- (必須) `TargetFunctionArn` – Lambda 関数の名前、または Amazon リソースネーム (ARN)。名前として使用できる値には次のようなものがあります。
  - 関数名 – `my-function`
  - 関数 ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
  - 部分的な ARN – `123456789012:function:my-function`

クロスオリジンリソース共有 (CORS)

さまざまなオリジンによる、関数 URL へのアクセス方法を定義するには、Cross-Origin Resource Sharing (CORS) を使用します。異なるドメインから関数 URL を呼び出す場合は、CORS を設定す

ることをお勧めします。Lambda では、関数 URL 用として、以下の CORS ヘッダーをサポートしています。

CORS ヘッダー	CORS 設定プロパティ	値の例
Access-Control-Allow-Origin	AllowOrigins	* (すべてのオリジンを許可する) https://www.example.com http://localhost:60905
Access-Control-Allow-Methods	AllowMethods	GET, POST, DELETE, *
Access-Control-Allow-Headers	AllowHeaders	Date, Keep-Alive , X-Custom-Header
Access-Control-Expose-Headers	ExposeHeaders	Date, Keep-Alive , X-Custom-Header
Access-Control-Allow-Credentials	AllowCredentials	TRUE
Access-Control-Max-Age	MaxAge	5 (デフォルト)300

Lambda コンソールまたは AWS CLI を使用して、関数 URL のために CORS を設定すると、Lambda は関数 URL を介して、すべてのレスポンスに CORS ヘッダーを自動的に追加するようになります。あるいは、手動で CORS ヘッダーを関数レスポンスに追加することも可能です。競合するヘッダーがある場合、関数 URL で設定された CORS ヘッダーが優先されます。

関数 URL のスロットリング

スロットリングは、関数がリクエストを処理するレートを制限します。これは、関数からダウンストリームリソースに過剰な負荷がかからないようにしたり、リクエストの急増に対応するなど、多くの状況で役立ちます。

予約済み同時実行数を設定することで、関数 URL を介して Lambda 関数が処理するリクエストのレートをスロットリングできます。予約済み同時実行数は、関数を同時に呼び出せる最大数を制限します。関数に対する 1 秒あたりの最大リクエストレート (RPS) は、設定された予約済み同時実行数の 10 倍に相当します。例えば、関数の予約済み同時実行数を 100 に設定すると、最大 RPS は 1,000 になります。

関数の同時実行数が、予約済み同時実行数を超えるたびに、関数 URL は HTTP ステータスコードの 429 を返します。設定された予約済み同時実行数を基に、それを 10 倍した最大の RPS 値を超えるリクエストを関数が受信した場合は、ユーザーに対しても HTTP の 429 エラーが表示されます。予約済み同時実行数の詳細については、「関数に対する予約済み同時実行数の設定」を参照してください。

関数 URL の非アクティブ化

緊急時には、関数 URL へのすべてのトラフィックを拒否する必要が生じることがあります。関数 URL を非アクティブ化するには、予約済み同時実行数を 0 に設定します。これにより、関数 URL へのすべてのリクエストがスロットリングされ、HTTP ステータスレスポンスには 429 が返されます。関数 URL を再アクティブ化するには、予約済み同時実行数を 1 以上の値に設定するか、この設定全体を削除します。

関数 URL の削除

関数の URL を削除すると、元に戻すことはできません。新しい関数 URL を作成すると、異なる URL アドレスになります。

> **Note**
>
> 認証タイプ NONE で関数 URL を削除した場合、Lambda では、関連するリソースベースのポリシーが自動的に削除されません。このポリシーを削除する場合は、手動で削除する必要があります。

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数の名前を選択します。
3. [設定] タブを開き、次に [関数 URL] をクリックします。
4. [削除] を選択します。
5. フィールドに [delete] (削除) という単語を入力して、削除を確認します。
6. [削除] を選択します。

> **Note**
>
> 関数 URL を持つ関数を削除すると、Lambda により関数 URL が非同期的に削除されます。同じアカウントで同じ名前の新しい関数をすぐに作成した場合、元の関数 URL は削除されず、新しい関数にマッピングされる可能性があります。

Lambda 関数 URL へのアクセスの制御

特定の関数にアタッチされたリソースベースのポリシーに、AuthType パラメータを組み合わせて使用することで、Lambda 関数 URL へのアクセスを制御できます。これら 2 つのコンポーネントの設定によって、関数 URL のために他の管理アクションを呼び出したり実行できるユーザーを指定します。

AuthType パラメーターは、関数 URL へのリクエストを Lambda が認証または承認する方法を決定します。関数 URL を設定する場合は、以下の AuthType オプションのいずれかを指定する必要があります。

- AWS_IAM – Lambda では、AWS Identity and Access Management (IAM) を使用して、IAM プリンシパルの ID ポリシーと関数のリソースベースのポリシーに基づいて、リクエストを認証および承認します。関数 URL を使用しての関数呼び出しを、認証済みユーザーとロールのみに制限する場合は、このオプションを選択します。

- NONE – Lambda では、関数を呼び出す前の認証を実行しません。ただし、関数のリソースベースのポリシーは常に有効であり、関数 URL がリクエストを受信できるようにするには、パブリックアクセスを許可する必要があります。関数 URL への、未認証でパブリックなアクセスを許可するには、このオプションを選択します。

関数を呼び出そうとする他の AWS アカウント に対しては、AuthType だけではなくリソースベースのポリシーを使用することでも、アクセス許可を付与することができます。詳しくは、「Lambda でのリソースベースのポリシーの使用」を参照してください。

セキュリティに関するさらなるインサイトを得るには、AWS Identity and Access Management Access Analyzer 使用して、関数 URL への外部アクセスの包括的な分析を取得します。IAM Access Analyzer は、Lambda 関数における新規または更新済みのアクセス許可を監視し、パブリックおよびクロスアカウントのアクセスのために付与する許可を特定するのに役立ちます。すべての AWS のお客様は、IAM Access Analyzer を無料で利用できます。IAM Access Analyzer の使用開始方法については、「AWS IAM Access Analyzer を使用する」を参照してください。

このページでは、両方の認証タイプによるリソースベースのポリシーの例と、これらのポリシーの作成に、AddPermission API オペレーションまたは Lambda コンソールを使用する方法について説明します。アクセス許可の設定後に関数 URL を呼び出す方法については、「Lambda 関数 URL の呼び出し」を参照してください。

トピック

- AWS_IAM 認証タイプの使用
- NONE 認証タイプの使用
- ガバナンスとアクセスコントロール

AWS_IAM 認証タイプの使用

AWS_IAM 認証タイプが選択されている場合は、Lambda 関数 URL を呼び出す必要のあるユーザーには、lambda:InvokeFunctionUrl アクセス許可が付与されている必要があります。呼び出しリクエストを実行するユーザーによっては、このアクセス許可を付与するために、リソースベースのポリシーを使用することが必要になります。

リクエストを実行するプリンシパルが関数 URL と同じ AWS アカウント 内にある場合、このプリンシパルには、アイデンティティベースのポリシー内に lambda:InvokeFunctionUrl のアクセス許可があるか、または、関数のリソースベースのポリシー内でアクセス許可が付与されているか、そのどちらかが必要です。つまり、既にユーザーがアイデンティティベースのポリシー内にアクセス許可 lambda:InvokeFunctionUrl を持っている場合は、リソースベースのポリシーはオプションです。ポリシー評価は、「Determining whether a request is allowed or denied within an account」(アカウント内のリクエストの許可または拒否の決定) で説明されているルールに従います。

リクエストを行うプリンシパルが別のアカウントにある場合、そのプリンシパルには、アクセス許可 lambda:InvokeFunctionUrl を付与する ID ベースのポリシーがあること、および、呼び出そうとしている関数のためのリソースベースのポリシー内でアクセス許可が付与されていること、その両方が必要となります。これらクロスアカウントのケースでのポリシー評価は、「Determining whether a cross-account request is allowed」(クロスアカウントリクエスト許可されているかどうかの確認) で説明されているルールに従います。

次に示す、クロスアカウントでのインタラクション用のリソースベースのポリシー例では、AWS アカウント 444455556666 内で、example ロールに対し関数 my-function に関連付けられた関数 URL を呼び出すことを許可しています。

Example 関数 URL のクロスアカウント呼び出しポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
```

```
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

このポリシーステートメントは、以下の手順に従いコンソールから作成できます。

別のアカウントに URL の呼び出し許可を付与するには (コンソール)

1. Lambda コンソールの関数ページを開きます。
2. URL での呼び出し許可を付与する関数の名前を選択します。
3. [Configuration] (設定) タブを開き、次に [Permissions] (アクセス許可) をクリックします。
4. [Resource-based policy] (リソースベースのポリシー) で、[Add permissions] (アクセス許可を追加) をクリックします。
5. [Function URL] (関数 URL) をクリックします。
6. [Auth Type] (認証タイプ) で [AWS_IAM] を選択します。
7. (オプション) [ステートメント ID]で、ポリシーステートメントのステートメント ID を入力します。
8. [プリンシパル] には、アカウント ID または許可の付与先となるユーザーまたはロールの Amazon リソースネーム (ARN) を入力します。例: **444455556666**。
9. [保存] を選択します。

または、次の add-permission AWS Command Line Interface (AWS CLI) コマンド 使用して、このポリシーステートメントを作成することもできます。

```
aws lambda add-permission --function-name my-function \
    --statement-id example0-cross-account-statement \
    --action lambda:InvokeFunctionUrl \
    --principal 444455556666 \
```

```
    --function-url-auth-type AWS_IAM
```

前の例では、条件キー lambda:FunctionUrlAuthType の値は AWS_IAM です。このポリシーでは、関数 URL の認証タイプも AWS_IAM である場合にのみアクセスを許可します。

NONE 認証タイプの使用

> **Important**
>
> 関数 URL の認証タイプが NONE で、リソースベースのポリシーでパブリックアクセスを許可している場合には、認証されていないすべてのユーザーが、関数 URL を使用して関数を呼び出すことができます。

場合によると、関数 URL の公開が必要な場合もあります。例えば、ウェブブラウザから直接行われたリクエストを処理する場合などです。関数 URL へのパブリックアクセスを許可するには、認証タイプに NONE を選択します

NONE の認証タイプを選択すると、Lambda は関数 URL へのリクエストに対して IAM による認証を行いません。この場合でも、依然としてユーザーには、関数 URL を正常に呼び出すためのアクセス許可 lambda:InvokeFunctionUrl が必要です。アクセス許可 lambda:InvokeFunctionUrl は、以下のリソースベースのポリシーを使用して付与することが可能です。

Example すべての認証されていないプリンシパル向けの関数 URL 呼び出しポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunctionUrl",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
```

```
}
```

> **Note**
>
> コンソール経由または AWS Serverless Application Model (AWS SAM) を使用し、認証タイプ NONE の関数 URL を作成すると、Lambda は上記のリソースベースのポリシーステートメントを自動的に作成します。ポリシーがすでに存在する場合、またはアプリケーションを作成しているユーザーまたはロールに適切なアクセス権限がない場合、Lambda はポリシーを作成しません。AWS CLI、AWS CloudFormation、または Lambda API を直接使用している場合には、`lambda:InvokeFunctionUrl` のアクセス許可を手作業で追加する必要があります。これにより、関数はパブリックになります。
>
> さらに、認証タイプ NONE で関数 URL を削除した場合、Lambda では、関連するリソースベースのポリシーが自動的に削除されません。このポリシーを削除する場合は、手動で削除する必要があります。

このステートメントでは、条件キー `lambda:FunctionUrlAuthType` の値は NONE です。このポリシーステートメントは、関数 URL の認証タイプも NONE である場合にのみアクセスを許可します。

関数のリソースベースのポリシーがアクセス許可 `lambda:invokeFunctionUrl` を付与していない場合、関数 URL の呼び出しを試みたユーザーは 403 Forbidden エラーコードを受け取ります。これは、関数 URL の認証タイプが NONE の場合も発生します。

ガバナンスとアクセスコントロール

関数 URL の呼び出しに関するアクセス許可に加えて、関数 URL の設定に使用するアクションによってもアクセス制御が可能です。Lambda では、関数 URL に対して次の IAM ポリシーアクションがサポートされます。

- `lambda:InvokeFunctionUrl` – 関数 URL を使用して Lambda 関数を呼び出します。
- `lambda:CreateFunctionUrlConfig` – 関数 URL を作成し、その AuthType を設定します。
- `lambda:UpdateFunctionUrlConfig` – 関数 URL 設定とその AuthType を更新します。
- `lambda:GetFunctionUrlConfig` – 関数 URL の詳細を表示します。
- `lambda:ListFunctionUrlConfigs` – 関数 URL 設定内容を一覧表示します。
- `lambda:DeleteFunctionUrlConfig` – 関数 URL を削除します。

> **Note**
>
> Lambda コンソールは、`lambda:InvokeFunctionUrl` に対するアクセス許可の追加のみをサポートしています。その他のすべてのアクションについては、アクセス許可の追加に Lambda API か AWS CLI を使用する必要があります。

他の AWS エンティティに対し、関数 URL によるアクセスを許可または拒否するには、IAM ポリシーにこれらのアクションを含めます。例えば、次のポリシーでは、AWS アカウント 444455556666 内にある `example` ロールに対し、アカウント 123456789012 にある関数 **my-function** の関数 URL を更新するためのアクセス許可を付与しています

Example クロスアカウント関数 URL のポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function"
        }
    ]
}
```

条件キー

関数 URL に対しきめ細かなアクセス制御を行うには、条件キーを使用します。Lambda では、関数 URL に対して 1 つの追加的な条件キー `FunctionUrlAuthType` をサポートしています。`FunctionUrlAuthType` キーは、関数 URL が使用する認証タイプを記述するための列挙値を定義します。この値は `AWS_IAM` または `NONE` となります。

この条件キーは、関数に関連付けられたポリシーの中で使用できます。例えば、関数 URL の構成を変更できるユーザーを制限したい場合があります。URL 認証タイプに `NONE` を使用する任意の関数に対して、すべての `UpdateFunctionUrlConfig` リクエストを拒否するには、次のポリシーを定義します。

Example 明示的な拒否を行う関数 URL ポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

AWS アカウント 444455556666 の example ロールに、URL 認証タイプ AWS_IAM を使用する関数に対して CreateFunctionUrlConfig および UpdateFunctionUrlConfig リクエストを許可するには、次のポリシーを定義できます。

Example 明示的な許可を行う関数 URL ポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
```

```
                }
            }
        }
    ]
}
```

また、この条件キーをサービスコントロールポリシー (SCP) で使用することもできます。SCP は、AWS Organizations にある組織全体でアクセス許可を管理する際に使用します。例えば、認証タイプに AWS_IAM を使用していない関数 URL を作成または更新することを、ユーザーに対し拒否するには、次のサービスコントロールポリシーを使用します。

Example 明示的な拒否を行う関数 URL SCP

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

Lambda 関数 URL の呼び出し

関数 URL は、Lambda 関数のための専用 HTTP エンドポイントです。関数 URL の作成と設定には、Lambda コンソールまたは Lambda API を使用します。関数 URL を作成すると、一意の URL エンドポイントが Lambda により自動的に生成されます。関数 URL を作成した後に、その URL エンドポイントが変更されることはありません。関数 URL のエンドポイントでは、次の形式を使用します。

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 関数 URLsは、アジアパシフィック (ハイデラバード) (ap-south-2) 、アジアパシフィック (メルボルン) ()、ap-southeast-4カナダ西部 (カルガリー) ()、ca-west-1欧州 (スペイン) ()、eu-south-2欧州 (チューリッヒ) ()、eu-central-2イスラエル (テルアビブ) ()、il-central-1中東 (アラブ首長国連邦) () の各リージョンではサポートされていませんme-central-1。

関数 URL はデュアルスタックに対応しており、IPv4 と IPv6 をサポートしています。関数 URL の設定が完了すると、ウェブブラウザ、curl、Postman、または任意の HTTP クライアントからの、HTTP エンドポイントを介した関数の呼び出しが可能になります。関数 URL を呼び出すには、アクセス許可 `lambda:InvokeFunctionUrl` が必要です。詳しくは、「アクセスコントロール」を参照してください。

トピック

関数 URL 呼び出しの基本

関数 URL が認証タイプに `AWS_IAM` を使用している場合、各 HTTP リクエストには、AWS Signature Version 4 (SigV4) による署名が必要です。awscurl、Postman、および AWS SigV4 Proxy などのツールには、Sigv4 でリクエストに署名するための方法が既に組み込まれています。

関数 URL への HTTP リクエストの署名にツールを使用しない場合は、各リクエストには、SigV4 を使用して手動で署名する必要があります。関数 URL がリクエストを受信すると、Lambda が

Sigv4 署名の処理を行います。署名が一致した場合のみ、Lambda によりリクエストが処理されます。SigV4 を使用してリクエストに手動で署名する方法については、Amazon Web Services 全般のリファレンス ガイドの「署名バージョン 4 を使用した AWS リクエストへの署名」を参照してください。

関数 URL が認証タイプに NONE を使用している場合は、リクエストには Sigv4 を使用した署名の必要はありません。ウェブブラウザ、curl、Postman、または任意の HTTP クライアントを使用して関数を呼び出すことができます。

関数へのシンプルな GET リクエストをテストするには、ウェブブラウザを使用します。例えば、関数 URL が `https://abcdefg.lambda-url.us-east-1.on.aws` で文字列パラメータ `message` を取り込む関数の場合、リクエストする URL は次のようになります。

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

POST リクエストなど、他の HTTP リクエストをテストする場合は、curl などのツールが利用できます。例えば、関数 URL への POST リクエストに一定の JSON データを含めたい場合には、次の curl コマンドを使用します。

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

リクエストとレスポンスのペイロード

クライアントが関数の URL を呼び出すと、Lambda は、まずこのリクエストをイベントオブジェクトにマップしてから、関数に受け渡します その関数の応答は、Lambda が関数 URL を介してクライアントに返信する HTTP レスポンスにマッピングされます。

このリクエストとレスポンスのイベント形式は、Amazon API Gateway ペイロード形式バージョン 2.0 と同じスキーマに従います。

リクエストペイロードの形式

リクエストペイロードは次の構造を持ちます。

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
```

```
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
```

```
  "isBase64Encoded": false,
  "stageVariables": null
}
```

パラメータ	説明	例
version	このイベントでのペイロード形式のバージョン。現在 Lambda 関数 URL では、ペイロード形式バージョン 2.0 をサポートしています。	2.0
routeKey	関数 URL ではこのパラメーターを使用しません。Lambda は、プレースホルダーとしてこの値に $default を設定します。	$default
rawPath	リクエストパス。例えば、リクエスト URL が https://{url-id}.lambda-url.{region}.on.aws/example/test/demo の場合は、raw パス値は /example/test/demo となります。	/example/test/demo
rawQueryString	リクエストのクエリ文字列パラメータを含む raw 文字列。サポートされている文字は、a-z、A-Z、0-9、.、_、-、%、&、=、+ などです。	"?parameter1=value1&parameter2=value2"
cookies	リクエストの一部として送信されたすべての Cookie を含む配列。	["Cookie_1=Value_1", "Cookie_2=Value_2"]

パラメータ	説明	例
headers	キーと値のペアで表されるリクエストヘッダーのリスト。	{"header1": "value1", "header2": "value2"}
queryStringParameters	リクエストに対するクエリパラメータです。例えば、リクエスト URL が https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane である場合の queryStringParameters の値は、キーに name を、値に Jane を持つ、JSON オブジェクトとなります	{"name": "Jane"}
requestContext	リクエストに関する追加的な情報 (requestId 、リクエストの発行時刻、および AWS Identity and Access Management (IAM) 経由で承認された場合は発信者の身元、その他) を含むオブジェクト。	
requestContext.accountId	関数所有者の AWS アカウント ID。	"123456789012"
requestContext.apiId	関数 URL ID。	"33anwqw8fj"
requestContext.authentication	関数 URL ではこのパラメーターを使用しません。Lambda では null がセットされます。	null

パラメータ	説明	例
requestContext.authorizer	関数 URL で AWS_IAM の認証タイプが使用されている場合の、発信者の ID に関する情報を含むオブジェクト。それ以外の場合、Lambda はこれに null をセットします。	
requestContext.authorizer.iam.accessKey	発信者 ID のアクセスキー。	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer.iam.accountId	発信者アイデンティティの AWS アカウント ID。	"111122223333"
requestContext.authorizer.iam.callerId	発信者の ID (ユーザー ID)。	"AIDACKCEVSQ6C2EXAMPLE"
requestContext.authorizer.iam.cognitoIdentity	関数 URL ではこのパラメーターを使用しません。Lambda は、この値に null を設定するか、JSON からこれを除外します。	null
requestContext.authorizer.iam.principalOrgId	発信者の身元に関連付けられているプリンシパル組織 ID。	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer.iam.userArn	発信者の身元のユーザー Amazon リソースネーム (ARN)。	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer.iam.userId	発信者の身元のユーザー ID。	"AIDACOSFODNN7EXAMPLE2"

パラメータ	説明	例
requestContext.domainName	関数 URL のドメイン名。	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	関数 URL のドメインプレフィックス。	"<url-id>"
requestContext.http	HTTP リクエストの詳細を含むオブジェクト。	
requestContext.http.method	リクエストで使用されている HTTP メソッド。有効な値には、GET、POST、PUT、HEAD、および DELETE があります。	GET
requestContext.http.path	リクエストパス。例えば、リクエスト URL が https://{url-id}.lambda-url.{region}.on.aws/example/test/demo であれば、パス値は /example/test/demo となります。	/example/test/demo
requestContext.http.protocol	リクエストのプロトコル。	HTTP/1.1
requestContext.http.sourceIp	リクエストを発行した即時 TCP 接続のソース IP アドレス。	123.123.123.123
requestContext.http.userAgent	User-Agent リクエストヘッダー値。	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0

パラメータ	説明	例
requestContext.requestId	呼び出しリクエストの ID。この ID は、関数に関連する呼び出しログをトレースするために使用できます。	e1506fd5-9e7b-434f-bd42-4f8fa224b599
requestContext.routeKey	関数 URL ではこのパラメーターを使用しません。Lambda は、プレースホルダーとしてこの値に $default を設定します。	$default
requestContext.stage	関数 URL ではこのパラメーターを使用しません。Lambda は、プレースホルダーとしてこの値に $default を設定します。	$default
requestContext.time	リクエストのタイムスタンプです。	"07/Sep/2021:22:50:22 +0000"
requestContext.timeEpoch	リクエストのタイムスタンプ (Unix エポック秒)。	"1631055022677"
body	リクエスト本文。リクエストのコンテンツタイプがバイナリの場合、本文は base64 でエンコードされます。	{"key1": "value1", "key2": "value2"}
pathParameters	関数 URL ではこのパラメーターを使用しません。Lambda は、この値に null を設定するか、JSON からこれを除外します。	null

パラメータ	説明	例
isBase64Encoded	ボディがバイナリペイロードで base64 でエンコードされている場合は TRUE、それ以外の場合は FALSE です。	FALSE
stageVariables	関数 URL ではこのパラメーターを使用しません。Lambda は、この値に null を設定するか、JSON からこれを除外します。	null

レスポンスペイロードの形式

関数からレスポンスが返されると、Lambda はそのレスポンスを解析し HTTP の形式に変換します。関数レスポンスペイロードの形式は以下のとおりです。

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda は自動的にレスポンス形式を推定します。関数が有効な JSON を返し、かつ statusCode を返さない場合、Lambda は以下のような想定を立てます。

- statusCode は 200。
- content-type は application/json。
- body は関数のレスポンス。

- `isBase64Encoded` は `false`。

次の例は、Lambda 関数の出力がレスポンスペイロードにどのようにマッピングされるか、また、レスポンスペイロードが最終的な HTTP レスポンスにどのようにマッピングされるかを示しています。関数の URL を呼び出したクライアントには、HTTP 応答が表示されます。

文字列による応答の出力例

Lambda 関数出力	インタプリティングされたレスポンス出力	HTTP レスポンス (クライアントに表示されるもの)
`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json content-length: 15 "Hello, world!"`

JSON によるレスポンスの出力例

Lambda 関数出力	インタプリティングされたレスポンス出力	HTTP レスポンス (クライアントに表示されるもの)
`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json" }, "isBase64Encoded": false`	`HTTP/2 200 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json content-length: 34 { "message": "Hello, world!" }`

Lambda 関数出力	インタプリティングされたレスポンス出力	HTTP レスポンス (クライアントに表示されるもの)
	} 	

カスタムレスポンスでの出力例

Lambda 関数出力	インタプリティングされたレスポンス出力	HTTP レスポンス (クライアントに表示されるもの)

{
"statusCode": 201,
"headers": {
"Content-Type":
"application/json",
"My-Custom-
Header": "Custom Value"
},
"body": JSON.stri
ngify({
"message":
"Hello, world!"
}),
"isBase64Encoded":
false
}
	{
	"statusCode": 201,
	"headers": {
	"Content-Type":
	"application/json",
	"My-Custom-
	Header": "Custom Value"
	},
	"body": JSON.stri
	ngify({
	"message":
	"Hello, world!"
	}),
	"isBase64Encoded":
	false
	}
		HTTP/2 201
		date: Wed, 08 Sep 2021
		18:02:24 GMT
		content-type: applicati
		on/json
		content-length: 27
		my-custom-header:
		Custom Value
		
		{
		"message": "Hello,
		world!"
		}

cookie

関数から Cookie を返す場合、手動で `set-cookie` ヘッダーを追加しないでください。代わりに、レスポンスペイロードオブジェクトに Cookie を含めます。Lambda はこれを自動的に解釈し、次の例のように HTTP レスポンスの `set-cookie` ヘッダーとして追加します。

クッキーを返すレスポンスの出力例

Lambda 関数出力	HTTP レスポンス (クライアントに表示されるもの)
{	HTTP/2 201

Lambda 関数出力	HTTP レスポンス (クライアントに表示されるもの)

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type": "application/json",
  "My-Custom-Header": "Custom Value"
  },
  "body": JSON.stringify({
  "message": "Hello, world!"
  }),
  "cookies": [
  "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
  "Cookie_2=Value2; Max-Age=78000"
  ],
  "isBase64Encoded": false
}</pre>
	<pre>date: Wed, 08 Sep 2021 18:02:24 GMT
	content-type: application/json
	content-length: 27
	my-custom-header: Custom Value
	set-cookie: Cookie_1=Value2; Expires=21 Oct 2021 07:48 GMT
	set-cookie: Cookie_2=Value2; Max-Age=78000
	
	{
	  "message": "Hello, world!"
	}</pre>

Lambda 関数 URL のモニタリング

関数 URL のモニタリングには、AWS CloudTrail および Amazon CloudWatch が使用できます。

トピック

CloudTrail による関数 URL のモニタリング

Lambda は、以下の関数 URL 用 API オペレーションのログを、CloudTrail ログファイル内のイベントとして自動的に記録します。

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

各ログエントリには、発信者 ID、リクエストが発行された日時、およびその他の詳細に関する情報が含まれます。CloudTrail で [Event history] (イベント履歴) を表示することで、過去 90 日以内のすべてのイベントを確認できます。90 日より前のレコードを保持するには、追跡を作成します。

CloudTrail のデフォルトでは、データイベントと見なされる `InvokeFunctionUrl` リクエストのログ記録は行われません。ただし、CloudTrail でデータイベントのログ記録を有効化することは可能です 詳細については、「AWS CloudTrail ユーザーガイド」の「証跡のデータイベントの記録」を参照してください。

関数 URL 用の CloudWatch メトリクス

Lambda は、関数 URL リクエストに関して集計されたメトリクスを、CloudWatch に送信します。CloudWatch コンソールでは、これらのメトリクスを使用して、関数 URL のモニタリング、ダッシュボードの作成、アラームの設定を行うことができます。

関数 URL では、呼び出しに関する以下のメトリクスをサポートしています。これらのメトリクスの表示には、`Sum` 統計を使用することをお勧めします。

- UrlRequestCount – この関数 URL に対し発行されたリクエストの数。
- Url4xxCount – 4xx HTTP ステータスコードを返したリクエストの数。4xx シリーズのコードは、不正なリクエストなど、クライアント側で発生したエラーを表します。
- Url5xxCount – 5xx HTTP ステータスコードを返したリクエストの数。5xx シリーズのコードは、機能エラーやタイムアウトなど、サーバー側で発生したエラーを表します。

関数 URL では、パフォーマンスに関する以下のメトリクスもサポートしています。このメトリクスの表示には、Average または Max 統計を使用することをお勧めします。

- UrlRequestLatency – 関数 URL がリクエストを受信してから関数 URL がレスポンスを返すまでの時間。

呼び出しとパフォーマンスに関するこれらのメトリクスは、それぞれ以下のディメンションをサポートしています。

- FunctionName – 未公開な関数の $LATEST バージョン (あるいは関数の任意のエイリアス) に割り当てられている、関数 URL の集計メトリクスを表示します。例えば、hello-world-function と指定します。
- Resource – 特定の関数 URL のメトリクスを表示します。この値は、未公開な関数の $LATEST バージョン (または関数のいずれかエイリアス) とならんで、関数の名前により定義されます 例えば、hello-world-function:$LATEST と指定します。
- ExecutedVersion – 特定の関数 URL のメトリクスを、実行されたバージョンに基づいて表示します。このディメンションは、主に、未公開な $LATEST バージョンに割り当てられた関数 URL を追跡するために使用します。

チュートリアル: 関数 URL を使用する Lambda 関数の作成

このチュートリアルでは、パブリック関数 URL エンドポイントから 2 つの数値の積を返す Lambda 関数を、.zip ファイルアーカイブとして定義することで作成します。関数 URL の設定の詳細については、「関数 URL の作成と管理」を参照してください。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。初めての方は、コンソールで Lambda の関数の作成 の手順に従って最初の Lambda 関数を作成してください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する

AWS リソースにアクセスするためのアクセス権限をLambda 関数に付与する実行ロールを作成します。

実行ロールを作成するには

1. AWS Identity and Access Management (IAM) コンソールの ロールページを開きます。
2. [ロールの作成] を選択します。
3. [信頼できるエンティティタイプ] で [AWS サービス] を選択し、[ユースケース] で [Lambda] を選択します。
4. [Next] を選択します。
5. [許可ポリシー] ペインの検索ボックスに「**AWSLambdaBasicExecutionRole**」と入力します。
6. AWS マネージドポリシー AWSLambdaBasicExecutionRole の横にあるチェックボックスをオンにしてから、[次へ] を選択します。
7. [ロール名] に「**lambda-url-role**」と入力して、[ロールの作成] をクリックします。

AWSLambdaBasicExecutionRole ポリシーには、ログを Amazon CloudWatch Logs に書き込むために関数が必要とするアクセス許可が含まれます。このチュートリアルの後半で、Lambda 関数の作成にロールの Amazon リソースネーム (ARN) が必要になります。

実行ロールの ARN を見つける方法

1. AWS Identity and Access Management (IAM) コンソールの ロールページを開きます。
2. 先ほど作成したロール (lambda-url-role) を選択します。
3. [概要] ペインで、ARN をコピーします。

関数 URL を使用する Lambda 関数を作成する (ZIP ファイルアーカイブ)

ZIP ファイルアーカイブから、関数 URL エンドポイントを使用する Lambda 関数を作成します。

関数を作成するには

1. 以下のコード例を index.js という名前のファイルにコピーします。

   Example index.js

```
exports.handler = async (event) => {
    let body = JSON.parse(event.body);
    const product = body.num1 * body.num2;
    const response = {
```

```
        statusCode: 200,
        body: "The product of " + body.num1 + " and " + body.num2 + " is " +
 product,
    };
    return response;
};
```

2. デプロイパッケージを作成します。

```
zip function.zip index.js
```

3. `create-function` コマンドを使用して Lambda 関数を作成します。ロール ARN は、チュートリアルの前半でコピーした独自の実行ロールの ARN に置き換えてください。

```
aws lambda create-function \
    --function-name my-url-function \
    --runtime nodejs18.x \
    --zip-file fileb://function.zip \
    --handler index.handler \
    --role arn:aws:iam::123456789012:role/lambda-url-role
```

4. 関数に、関数の URL へのパブリックアクセスを許可する権限を付与するリソースベースのポリシーを追加します。

```
aws lambda add-permission \
    --function-name my-url-function \
    --action lambda:InvokeFunctionUrl \
    --principal "*" \
    --function-url-auth-type "NONE" \
    --statement-id url
```

5. `create-function-url-config` コマンドを使用して、関数の URL エンドポイントを作成します。

```
aws lambda create-function-url-config \
    --function-name my-url-function \
    --auth-type NONE
```

関数 URL エンドポイントをテストする

curl や Postman などの HTTP クライアントから関数 URL エンドポイントをコールすることで、Lambda 関数を呼び出します。

```
curl 'https://abcdefg.lambda-url.us-east-1.on.aws/' \
-H 'Content-Type: application/json' \
-d '{"num1": "10", "num2": "10"}'
```

以下の出力が表示されます。

```
The product of 10 and 10 is 100
```

関数 URL を使用する Lambda 関数を作成する (CloudFormation)

また、AWS CloudFormation タイプに `AWS::Lambda::Url` を指定することで、関数 URL エンドポイントを使用する Lambda 関数を作成することもできます。

```
Resources:
  MyUrlFunction:
    Type: AWS::Lambda::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs18.x
      Role: arn:aws:iam::123456789012:role/lambda-url-role
      Code:
        ZipFile: |
          exports.handler = async (event) => {
              let body = JSON.parse(event.body);
              const product = body.num1 * body.num2;
              const response = {
                  statusCode: 200,
                  body: "The product of " + body.num1 + " and " + body.num2 + " is " +
 product,
              };
              return response;
          };
      Description: Create a function with a URL.
  MyUrlFunctionPermissions:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyUrlFunction
```

```
      Action: lambda:InvokeFunctionUrl
      Principal: "*"
      FunctionUrlAuthType: NONE
  MyFunctionUrl:
    Type: AWS::Lambda::Url
    Properties:
      TargetFunctionArn: !Ref MyUrlFunction
      AuthType: NONE
```

関数 URL を使用する Lambda 関数の作成 (AWS SAM)

また、AWS Serverless Application Model (AWS SAM) により、関数 URL が設定された Lambda 関数を作成することもできます。

```
ProductFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/.
    Handler: index.handler
    Runtime: nodejs18.x
    AutoPublishAlias: live
    FunctionUrlConfig:
      AuthType: NONE
```

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。

2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[Delete] (削除) を選択します。

AWS Lambda 関数の管理

Lambda API またはコンソールを使用して、Lambda 関数に関連付けられたリソースを調整し、セキュア化する方法を学びます。

AWS CLI で Lambda を使用する

AWS Command Line Interface を使用して関数と他の AWS Lambda リソースを管理できます。AWS CLI は、AWS SDK for Python (Boto) を使用して Lambda API を操作します。このチュートリアルでは、AWS CLI を使用して Lambda 関数を管理して呼び出します。

関数スケーリング

予約された同時実行とプロビジョニングされた同時実行の 2 つの関数レベルでの同時実行コントロールを設定できます。同時実行とは、アクティブな関数のインスタンスの数で、重要な関数がスロットリングを回避することを確実にするために設定できます。

コード署名

Lambda のコード署名により、信頼性と整合性を管理できます。それにより、承認されたデベロッパーによって公開された未変更のコードのみが、Lambda 関数にデプロイされていることを確認できます。

タグで整理

Lambda 関数にタグ付けして属性ベースのアクセス制御 (ABAC) をアクティブ化し、それらを所有者、プロジェクト、または部門別に整理することができます。

レイヤーの使用

ビジネスロジックの記述をより迅速にイテレートできるように、以前に作成したレイヤーを適用してデプロイパッケージのサイズを縮小し、コードの共有と責任の分離を推進することができます。

AWS CLI での Lambda の使用

AWS Command Line Interface を使用して関数と他の AWS Lambda リソースを管理できます。AWS CLI は、AWS SDK for Python (Boto) を使用して Lambda API を操作します。これを使用して API について学び、AWS SDK で Lambda を使用するアプリケーションの構築にその知識を適用できます。

このチュートリアルでは、AWS CLI を使用して Lambda 関数を管理して呼び出します。詳細については、AWS Command Line Interface ユーザーガイドの「AWS CLI とは」を参照してください。

前提条件

このチュートリアルでは、基本的な Lambda オペレーションと Lambda コンソールについてある程度の知識があることを前提としています。まだ知識がない場合は、「the section called “コンソールで Lambda の関数の作成”」の指示に従ってください。

以下の手順を完了するには、「AWS Command Line Interface (AWS CLI) バージョン 2」が必要です。コマンドと予想される出力は、別々のブロックにリストされます。

```
aws --version
```

次のような出力が表示されます。

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

コマンドが長い場合、コマンドを複数行に分割するためにエスケープ文字 (\) が使用されます。

Linux および macOS では、任意のシェルとパッケージマネージャーを使用します。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (zip など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。このガイドの CLI コマンドの例では、Linux フォーマットを使用しています。Windows CLI を使用している場合、インライン JSON ドキュメントを含むコマンドを再フォーマットする必要があります。

実行ロールを作成する

AWS リソースにアクセスするためのアクセス許可を関数に付与する実行ロールを作成します。AWS CLI を使用して実行ロールを作成するには、`create-role` コマンドを使用します。

次の例では、信頼ポリシーをインラインで指定しています。JSON 文字列で引用符をエスケープするための要件は、シェルに応じて異なります。

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '{"Version":
 "2012-10-17","Statement": [{ "Effect": "Allow", "Principal": {"Service":
 "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]}'
```

また、JSON ファイルを使用してロールの信頼ポリシーを定義することもできます。次の例では、`trust-policy.json` は現在のディレクトリにあるファイルです。この信頼ポリシーは、AWS Security Token Service (AWS STS) `AssumeRole` アクションを呼び出すサービスプリンシパルの `lambda.amazonaws.com` アクセス許可を付与することで、Lambda がロールのアクセス許可を使用できるようにします。

Example trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-
policy.json
```

以下の出力が表示されます。

```
{
```

```
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
        "CreateDate": "2020-01-17T23:19:12Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

ロールにアクセス許可を追加するには、attach-policy-to-role コマンドを使用します。AWSLambdaBasicExecutionRole マネージドポリシーを追加して開始します。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

このAWSLambdaBasicExecutionRoleポリシーには、関数が CloudWatch ログにログを書き込むために必要なアクセス許可があります。

関数を作成する

次の例では、環境変数の値とイベントオブジェクトをログに記録します。

Example index.js

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.log("EVENT\n" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

関数を作成するには

1. サンプルコードを index.js という名前のファイルにコピーします。
2. デプロイパッケージを作成します。

```
zip function.zip index.js
```

3. create-function コマンドを使用して Lambda 関数を作成します。ロール ARN 内の強調表示されたテキストをアカウント ID に置き換えます。

```
aws lambda create-function --function-name my-function \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-ex
```

次のような出力が表示されます。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-ex",
    "Handler": "index.handler",
    "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",
    ...
}
```

コマンドラインから呼び出しのログを取得するには、--log-type オプションを使用します。レスポンスには、LogResult フィールドが含まれます。このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

このログをデコードするには、base64 ユーティリティを使用します。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text |  base64 -d
```

次のような出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
  "AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64 は、Linux、macOS、および Ubuntu on Windows で使用できます。macOS の場合、コマンドは base64 -D です。

コマンドラインから完全なログイベントを取得するには、前の例に示すように、関数の出力にログストリーム名を含めることができます。次の例のスクリプトは、my-function という名前の関数を呼び出し、最後の 5 つのログイベントをダウンロードします。

Example get-logs.sh スクリプト

この例では、my-function がログストリーム ID を返す必要があります。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-name
 $(cat out) --limit 5
```

このスクリプトは sed を使用して出力ファイルから引用符を削除し、15 秒間スリープ状態にすることにより、ログを使用できるようにします。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

```
./get-logs.sh
```

以下の出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
```

```
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

関数の更新

関数を作成したら、トリガー、ネットワークアクセス、ファイルシステムアクセスなどの、関数の追加機能を設定できます。また、メモリや並行処理など、関数に関連のあるリソースも調整できます。これらの設定は、.zip ファイルアーカイブとして定義された関数と、コンテナイメージとして定義された関数に適用されます。

update-function-configuration コマンドを使用して 関数を設定します。次の例では、関数メモリを 256 MB に設定します。

Example update-function-configuration コマンド

```
aws lambda update-function-configuration \
--function-name my-function \
--memory-size 256
```

アカウントの Lambda 関数のリスト化

以下の AWS CLI コマンド `list-functions` を実行して、作成した関数のリストを取得します。

```
aws lambda list-functions --max-items 10
```

次のような出力が表示されます。

```
{
    "Functions": [
        {
            "FunctionName": "cli",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Runtime": "nodejs20.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-ex",
            "Handler": "index.handler",
```

```
            ...
        },
        {
            "FunctionName": "random-error",
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:random-
error",
            "Runtime": "nodejs20.x",
            "Role": "arn:aws:iam::123456789012:role/lambda-role",
            "Handler": "index.handler",
            ...
        },
        ...
      ],
      "NextToken": "eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0="
}
```

応答の際、Lambda は最大 10 個の関数のリストを返します。取得できる関数がさらにある場合、NextToken は、次の list-functions リクエストで使用できるマーカーを提供します。次の list-functions AWS CLI コマンドは、--starting-token パラメータを示す例です。

```
aws lambda list-functions --max-items 10 --starting-
token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0=
```

Lambda 関数の取得

Lambda CLI get-function コマンドは、関数のデプロイパッケージをダウンロードするために使用できる、Lambda 関数のメタデータと署名付き URL を返します。

```
aws lambda get-function --function-name my-function
```

以下の出力が表示されます。

```
{
    "Configuration": {
        "FunctionName": "my-function",
        "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
        "Runtime": "nodejs20.x",
        "Role": "arn:aws:iam::123456789012:role/lambda-ex",
        "CodeSha256": "FpFMvUhayLkOoVBpNuNiIVML/tuGv2iJQ7t0yWVTU8c=",
        "Version": "$LATEST",
        "TracingConfig": {
```

```
            "Mode": "PassThrough"
        },
        "RevisionId": "88ebe1e1-bfdf-4dc3-84de-3017268fa1ff",
        ...
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-east-2-tasks.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-function-4203078a-b7c9-4f35-..."
    }
}
```

詳細については、「」を参照してくださいGetFunction。

クリーンアップ

以下の delete-function コマンドを実行して、my-function 関数を削除します。

```
aws lambda delete-function --function-name my-function
```

IAM コンソールで作成した IAM ロールを削除します。ロールの削除に関する詳細については、IAM ユーザーガイドの「ロールおよびインスタンスプロファイルを削除する」を参照してください。

Lambda 関数のスケーリングについて

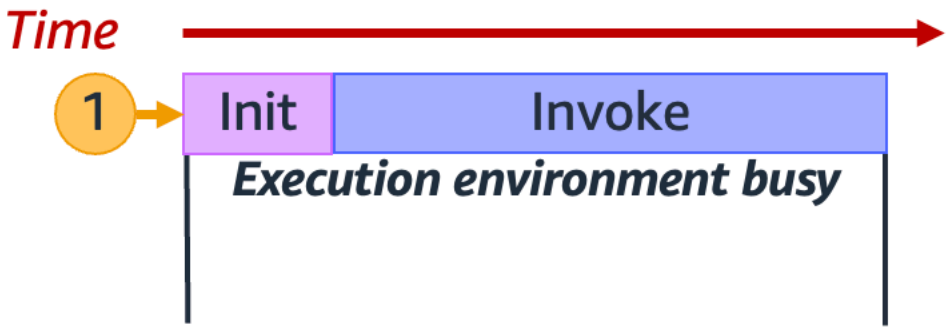
同時実行数とは、AWS Lambda 関数が同時に処理できる未完了のリクエストの数のことです。Lambda は、同時実行リクエストごとに、実行環境の個別のインスタンスをプロビジョニングします。関数が受け取るリクエストが増えると、Lambda が実行環境数のスケーリングを自動的に処理し、これはアカウントの同時実行上限に達するまで行われます。Lambda はアカウントに対し、1 つの AWS リージョン内のすべての関数全体での合計数 1,000 を上限とした同時実行をデフォルトで提供しています。特定のアカウントニーズをサポートするため、クォータの引き上げをリクエストしたり、関数レベルでの同時実行コントロールを設定したりして、重要な関数でスロットリングが発生しないようにすることができます。

このトピックでは、Lambda での同時実行数と関数スケーリングについて説明します。このトピックを読み終える頃には、同時実行を計算する、2 つの主な同時実行コントロールオプション (予約された同時実行とプロビジョニングされた同時実行) を視覚化する、適切な同時実行コントロール設定を見積もる、およびさらなる最適化のためのメトリクスを表示する方法を理解できるようになります。

セクション

同時実行の概要と視覚化

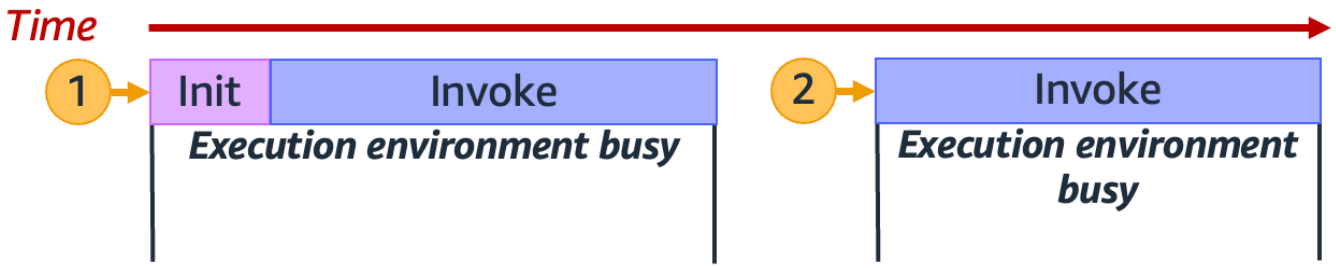
Lambda は、セキュアで分離された実行環境で関数を呼び出します。リクエストを処理するため、Lambda はまず実行環境を初期化 (初期化フェーズ) してから、それを使用して関数を呼び出す (呼び出しフェーズ) 必要があります。



> **Note**
>
> 実際の初期化と呼び出しの所要時間は、選択したランタイムや Lambda 関数コードなど、さまざまな要因に応じて異なります。上記の図は、初期化フェーズと呼び出しフェーズの所要時間の正確な割合を表すものではありません。

上記の図では、長方形を使用して単一の実行環境を表しています。関数が最初のリクエスト (ラベル 1 が付いた黄色い円) を受け取ると、Lambda が初期化フェーズ中に新しい実行環境を作成し、メインハンドラー外でコードを実行します。次に、Lambda は呼び出しフェーズ中に関数のメインハンドラーコードを実行します。この実行環境は、このプロセス全体を通じてビジー状態になり、他のリクエストを処理することはできません。

この実行環境は、Lambda が最初のリクエストの処理を終了した時点で、同じ関数に対する追加のリクエストを処理できるようになります。Lambda が、後続のリクエストのために実行環境を再度初期化する必要はありません。

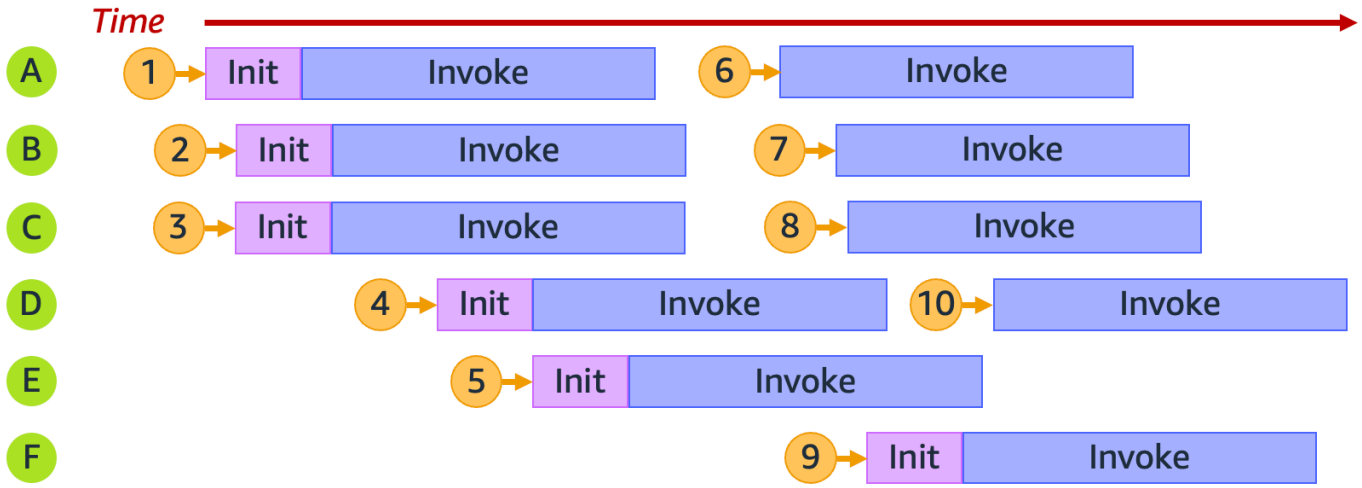


上記の図では、Lambda が実行環境を再利用して 2 番目のリクエスト (ラベル 2 が付いた黄色の円) を処理します。

これまでは、実行環境の単一のインスタンス (つまり、1 個の同時実行) のみに焦点を当ててきました。実際には、すべての受信リクエストを処理するために、Lambda は複数の実行環境インスタンスを並行してプロビジョニングする必要がある場合があります。関数が新しいリクエストを受け取ると、以下の 2 つのいずれかが行われる可能性があります。

- 事前に初期化された実行環境インスタンスが利用できる場合は、Lambda がそれを使用してリクエストを処理する。
- 利用できない場合は、Lambda が新しい実行環境インスタンスを作成してリクエストを処理する。

例として、関数が 10 個のリクエストを受け取った場合について検証してみましょう。



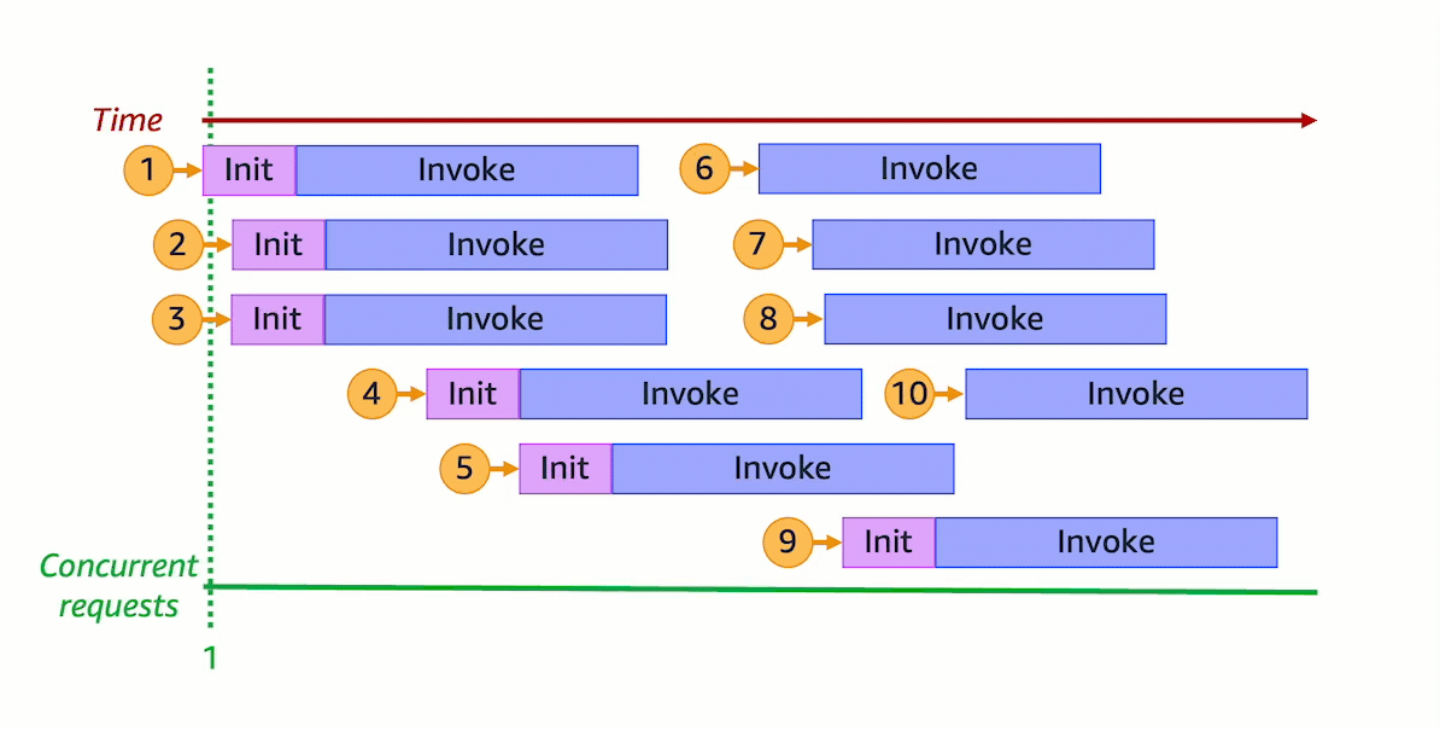
上記の図では、各横棒が単一の実行環境インスタンス (A から F でラベル付けされたもの) を表しています。Lambda はこのように各リクエストを処理します。

リクエスト 1 ~ 10 に対する Lambda の動作

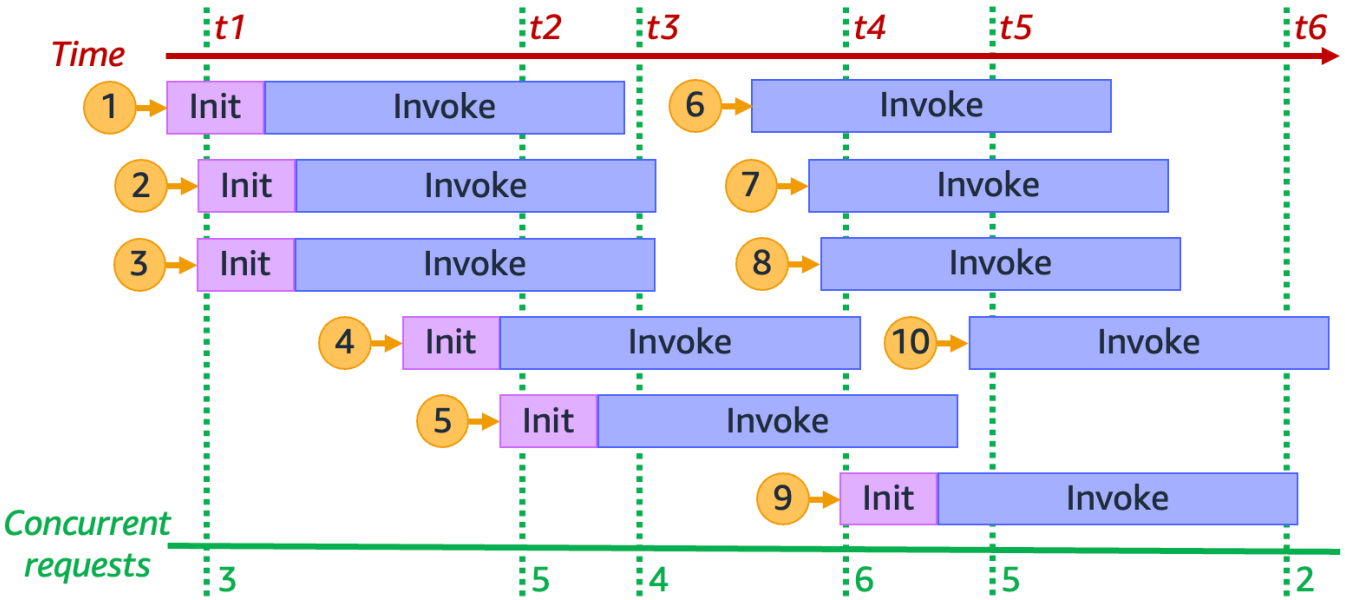
リクエスト	Lambda の動作	理由
1	新しい環境 A をプロビジョニング	これは最初のリクエストで、利用できる実行環境インスタンスはありません。
2	新しい環境 B をプロビジョニング	既存の実行環境インスタンス A がビジー状態。

リクエスト	Lambda の動作	理由
3	新しい環境 C をプロビジョニング	既存の実行環境インスタンス A と B がどちらもビジー状態。
4	新しい環境 D をプロビジョニング	既存の実行環境インスタンス A、B、および C のすべてがビジー状態。
5	新しい環境 E をプロビジョニング	既存の実行環境インスタンス A、B、C、および D のすべてがビジー状態。
6	環境 A を再利用	実行環境インスタンス A がリクエスト 1 の処理を完了し、利用可能になっている。
7	環境 B を再利用	実行環境インスタンス B がリクエスト 2 の処理を完了し、利用可能になっている。
8	環境 C を再利用	実行環境インスタンス C がリクエスト 3 の処理を完了し、利用可能になっている。
9	新しい環境 F をプロビジョニング	既存の実行環境インスタンス A、B、C、D、および E のすべてがビジー状態。
10	環境 D を再利用	実行環境インスタンス D がリクエスト 4 の処理を完了し、利用可能になっている。

関数が受け取る同時リクエストが増えると、Lambda はそれに応じて実行環境インスタンスの数をスケールアップします。以下のアニメーションは、同時リクエストの数を経時的に追跡するものです。



上記のアニメーションを 6 つの異なる時点で停止すると、以下の図が得られます。



上記の図では、任意の時点で垂直の線を引いて、その線に交差する環境の数を数えることができます。そうすることで、その時点での同時リクエストの数がわかります。例えば、t1 の時点では、3 件の同時リクエストを処理する 3 個のアクティブな環境があります。このシミュレーションで

は、t4 の時点で最大数の同時リクエストが行われており、6 個のアクティブな環境が 6 件の同時リクエストを処理しています。

要約すると、関数の同時実行とは、関数が同時に処理している同時リクエストの数になります。Lambda は、関数の同時実行の増加に対応してより多くの実行環境インスタンスをプロビジョニングし、リクエストの需要を満たします。

関数の同時実行数の計算

一般に、システムの同時実行性とは、複数のタスクを同時に処理する能力のことです。Lambda での同時実行は、関数が同時に処理している未完了のリクエストの数です。Lambda 関数の同時実行を測定するためのすばやく実用的な方法は、以下の式を使用することです。

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

同時実行の数は、1 秒あたりのリクエスト数とは異なります。例えば、関数が 1 秒あたり平均 100 件のリクエストを受け取るとします。リクエストの平均所要時間が 1 秒の場合、同時実行の数も 100 になります。

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

ただし、リクエストの平均所要時間が 500 ミリ秒の場合、同時実行の数は 50 になります。

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

同時実行の数が 50 というのは、実際に何を意味するのでしょうか？ リクエストの平均所要時間が 500 ミリ秒の場合、関数のインスタンスは 1 秒あたり 2 件のリクエストを処理できると考えられます。その場合、1 秒あたり 100 リクエストの負荷を処理するには、関数のインスタンスが 50 個必要です。50 の同時実行は、このワークロードをスロットリングなしで効率的に処理するには Lambda が 50 個の実行環境インスタンスをプロビジョニングする必要があることを意味します。これを方程式で表現すると、以下のようになります。

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

関数が 2 倍のリクエスト (1 秒あたり 200 件のリクエスト) を受け取り、各リクエストの処理に半分の時間 (250 ミリ秒) しか必要ない場合でも、同時実行の数は 50 のままです。

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

同時実行に関する理解度をテストする

実行に平均 200 ミリ秒かかる関数があるとします。負荷のピーク時には、1 秒あたり 5,000 件のリクエストがあります。負荷のピーク時における関数の同時実行の数を計算してください。

回答

関数の平均所要時間は 200 ミリ秒、つまり 0.2 秒です。同時実行の式を使用し、これらの数値を代入すると、同時実行の数は 1,000 になります。

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

言い換えると、関数の平均所要時間が 200 ミリ秒の場合、その関数は 1 秒あたり 5 件のリクエストを処理できることになります。1 秒あたり 5,000 リクエストのワークロードを処理するには、1,000 個の実行環境インスタンスが必要です。したがって、同時実行の数は 1,000 になります。

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

同時実行数と 1 秒あたりのリクエスト数の区別

前のセクションで述べたように、同時実行の数は、秒あたりのリクエスト数とは異なります。この違いは、平均リクエスト時間が 100 ミリ秒未満の関数を扱う場合に特に重要になります。

一般に、実行環境の各インスタンスでは、1 秒あたりに処理できるのは多くて 10 リクエスト程度です。この制限は、同期オンデマンド関数だけでなく、プロビジョニングされた同時実行機能を使用する関数にも適用されます。この制限を十分理解していない場合は、なぜそのような関数が特定のシナリオでスロットリングを受けるのかわからないかもしれません。

例えば、平均リクエスト時間が 50 ミリ秒の関数を考えてみましょう。1 秒あたり 200 リクエストでこの関数の同時実行を行います。

```
Concurrency = (200 requests/second) * (0.05 second/request) = 10
```

この結果から、この負荷の処理に必要な実行環境インスタンスは 10 個のみのように見えるかも知れません。ところが、各実行環境は 1 秒あたり 10 回の実行しか処理できません。つまり、実行環境が 10 個の場合、関数は合計 200 件のリクエストのうち、1 秒あたり 100 リクエストしか処理できません。この関数ではスロットリングが発生します。

ここで重要なのは、関数の同時実行設定を行う際には、同時実行と 1 秒あたりのリクエスト数の両方を考慮する必要があるということです。この場合、同時実行数が 10 個のみの場合でも、関数には 20 個の実行環境が必要です。

同時実行 (100 ミリ秒未満の関数) についての理解度をテスト

実行に平均 20 ミリ秒かかる関数があるとします。負荷のピーク時には、1 秒あたり 3,000 件のリクエストがあります。負荷のピーク時における関数の同時実行の数を計算してください。

回答

関数の平均所要時間は 20 ミリ秒、つまり 0.02 秒です。同時実行の式を使用し、これらの数値を代入すると、同時実行の数は 60 になります。

```
Concurrency = (3,000 requests/second) * (0.02 seconds/request) = 60
```

ところが、各実行環境は 1 秒あたり 10 回のリクエストしか処理できません。実行環境が 60 個ある場合、関数は 1 秒あたり最大 600 件のリクエストを処理できます。3,000 件のリクエストに完全に対応するには、関数に少なくとも 300 個の実行環境インスタンスが必要です。

予約済み同時実行数とプロビジョニングされた同時実行数について

デフォルトで、アカウントの同時実行はリージョン内のすべての関数全体で 1,000 に制限されています。関数は、この 1,000 の同時実行のプールをオンデマンドで共有します。利用できる同時実行が不足すると、関数でスロットリングが発生します (つまり、リクエストがドロップされ始めます)。

関数の中には、他の関数よりも重要なものがあります。そのため、重要な関数が必要な同時実行を利用できるように、同時実行を設定することをお勧めします。同時実行コントロールには、予約された同時実行とプロビジョニングされた同時実行の 2 つのタイプのコントロールがあります。

- 予約された同時実行は、関数のためにアカウントの同時実行の一部を予約するために使用します。これは、他の関数が利用可能な予約されていない同時実行のすべてを使い切ってしまわないようにするために役立ちます。
- プロビジョニングされた同時実行は、関数用の多数の環境インスタンスを事前に初期化するために使用します。これはコールドスタートレイテンシーの削減に役立ちます。

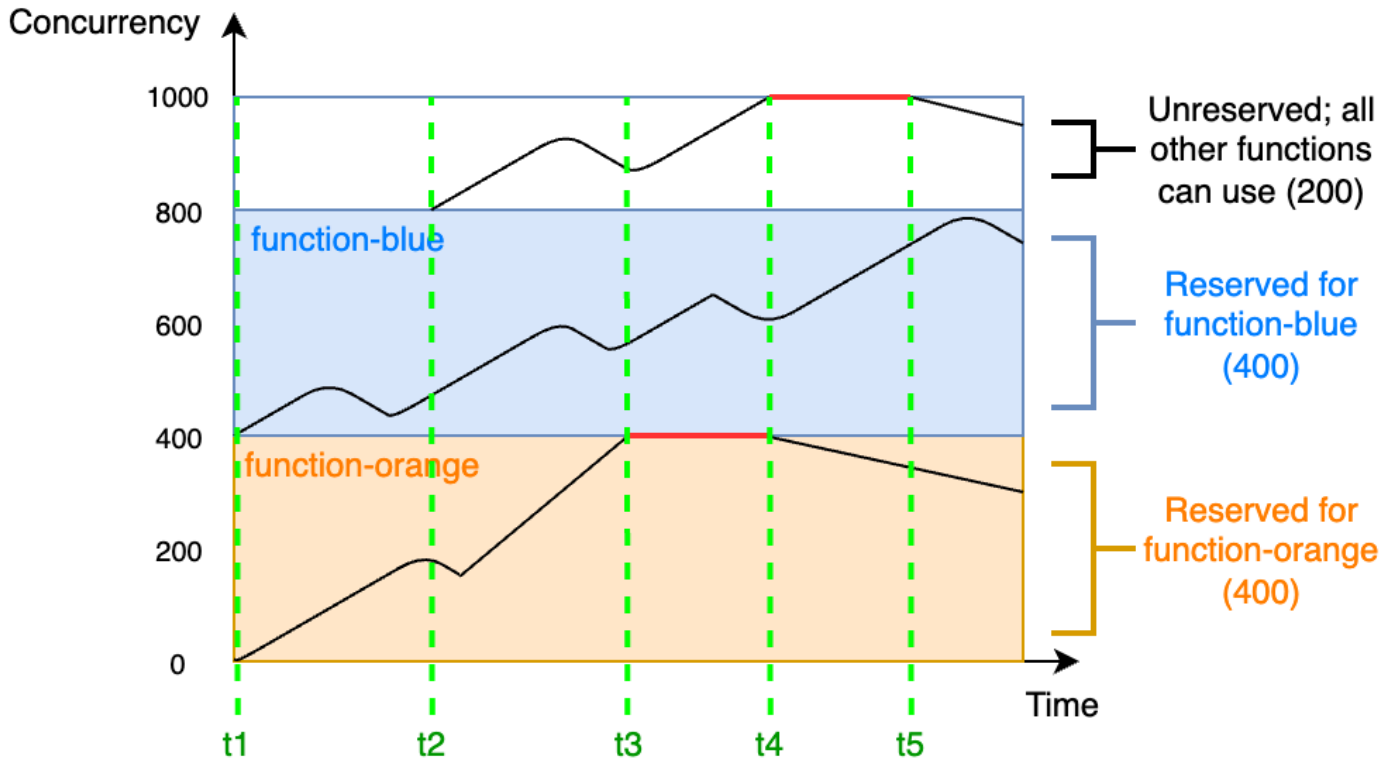
予約された同時実行

関数のために一定数の同時実行が常に利用可能であることを確実にしたい場合は、予約された同時実行を使用します。

予約された同時実行は、関数に割り当てる同時インスタンスの最大数です。予約された同時実行を 1 つの関数専用の同時実行にすると、他の関数がその同時実行を使用することはできません。つまり、予約された同時実行の設定は、他の関数が利用できる同時実行のプールに影響を与える可能性があります。予約された同時実行がない関数は、残りの予約されていない同時実行のプールを共有します。

予約された同時実行の設定は、アカウント全体の同時実行上限にカウントされます。関数に対して予約済み同時実行を設定する場合、料金はかかりません。

予約された同時実行に関する理解を深めるため、以下の図を検証しましょう。



この図では、このリージョン内のすべての関数に対するアカウントの同時実行上限が、デフォルト上限の 1,000 になっています。function-blue と function-orange の 2 つの重要な関数があり、大量の呼び出しが定期的に行われることが見込まれているとします。function-blue に予約された同時実行を 400 ユニット、function-orange にも予約された同時実行を 400 ユニット割り当てることにしました。この例では、アカウント内のその他すべての関数が、残りの予約されていない同時実行 200 ユニットを共有する必要があります。

この図には、以下の 5 つの注目点があります。

- t1 で、function-orange と function-blue の両方がリクエストの受け取りを開始します。各関数は、それぞれに割り当てられている予約された同時実行ユニットを使用し始めます。

- t2 では、function-orange と function-blue が受け取るリクエスト数が着実に増加します。それと同時に他の Lambda 関数がいくつかデプロイされ、それらもリクエストの受け取りを開始します。これらの他の関数には、予約された同時実行を割り当てません。これらの関数は、残りの予約されていない同時実行 200 ユニットの使用を開始します。
- t3 で、function-orange が最大同時実行数の 400 に到達します。アカウントのどこかに未使用の同時実行はあるものの、function-orange はそれらにアクセスできません。赤い線は、function-orange でスロットリングが発生していることを示し、Lambda はリクエストをドロップする可能性があります。
- t4 では、function-orange が受け取るリクエストの数が減り始め、スロットルされなくなります。しかし、他の関数ではトラフィックが急増し、スロットルされ始めます。アカウントのどこかに未使用の同時実行はあるものの、これらの他の関数はそれらにアクセスできません。赤い線は、他の関数でスロットリングが発生していることを示します。
- t5 では、他の関数が受け取るリクエストの数が減り始め、スロットルされなくなります。

この例から、予約された同時実行に以下の効果があることがわかります。

- 関数は、アカウント内の他の関数とは別個にスケーリングできる。同じリージョン内にあるアカウントの関数で、予約された同時実行がないすべての関数は、予約されていない同時実行のプールを共有します。予約された同時実行がないと、他の関数が利用可能なすべての同時実行を使い切る可能性があります。これは、重要な関数が必要に応じてスケールアップすることを妨げます。
- 関数は、際限なくスケールアウトできない。予約された同時実行は、関数の同時実行に上限を設定します。つまり、関数は、他の関数用に予約されている同時実行や、予約されていない同時実行のプールを使用できません。同時実行は、関数がアカウント内の利用可能な同時実行のすべてを使用したり、ダウンストリームリソースを過負荷状態にしたりすることがないように予約できます。
- アカウントで利用可能な同時実行のすべてを使用できない場合がある。同時実行の予約は、アカウントの同時実行上限にカウントされますが、他の関数が予約された同時実行の部分を使用できないことも意味します。関数が予約された同時実行を使い切らない場合は、実質的にその同時実行を無駄にすることになります。アカウント内の他の関数が無駄になった同時実行からメリットを得られるならば、これは問題にはなりません。

関数用に予約された同時実行の設定を管理する方法については、「関数に対する予約済み同時実行数の設定」を参照してください。

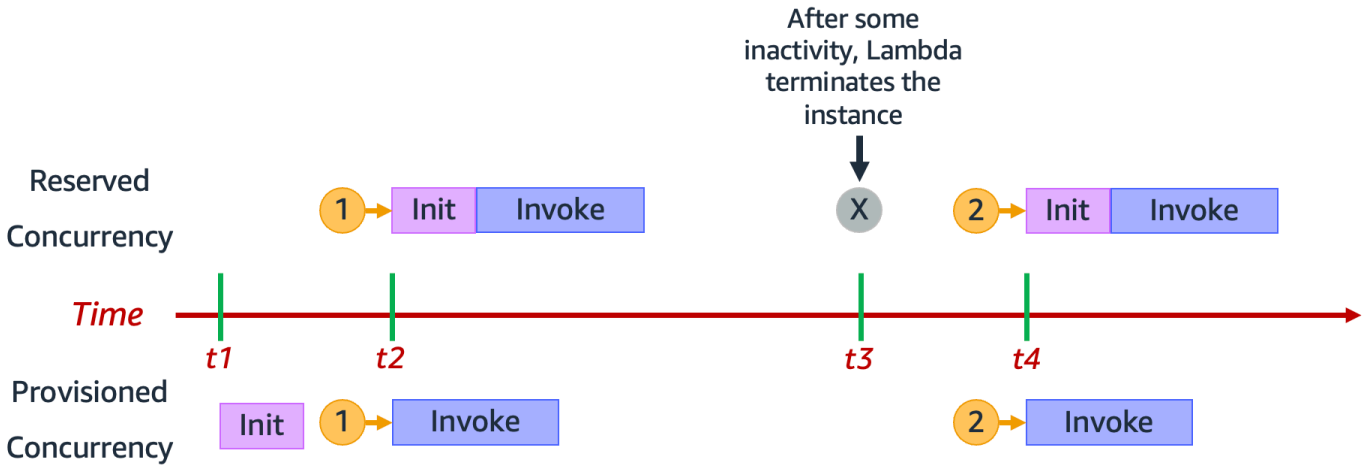
プロビジョニングされた同時実行

予約された同時実行は、Lambda 関数用に予約されている実行環境の最大数を定義するために使用しますが、これらの環境は、いずれも事前に初期化されていません。その結果、関数の呼び出し時間が長くなる可能性があります。Lambda は、最初に新しい環境を初期化してから、それを使って関数を呼び出す必要があるためです。Lambda が呼び出しを実行するために新しい環境を初期化しなければならない状況は、コールドスタートとして知られています。コールドスタートを緩和するため、プロビジョニングされた同時実行を使用できます。

プロビジョニングされた同時実行は、関数に割り当てる、事前に初期化された実行環境の数です。関数にプロビジョニングされた同時実行を設定すると、Lambda はその数だけ実行環境を初期化して、関数リクエストに即座に応答できるようにしておきます。

> **Note**
>
> プロビジョニングされた同時実行を使用すると、アカウントに追加料金が請求されます。Java 11 または Java 17 のランタイムを使用している場合は、Lambda SnapStart を使用して、追加費用なしでコールドスタートの問題を軽減することもできます。SnapStart は、実行環境のキャッシュされたスナップショットを使用して、起動時のパフォーマンスを大幅に向上させます。SnapStart とプロビジョニングされた同時実行の両方を同じ関数バージョンで使用することはできません。SnapStart の機能、制限事項、サポートされているリージョンの詳細については、「Lambda SnapStart による起動パフォーマンスの向上」を参照してください。

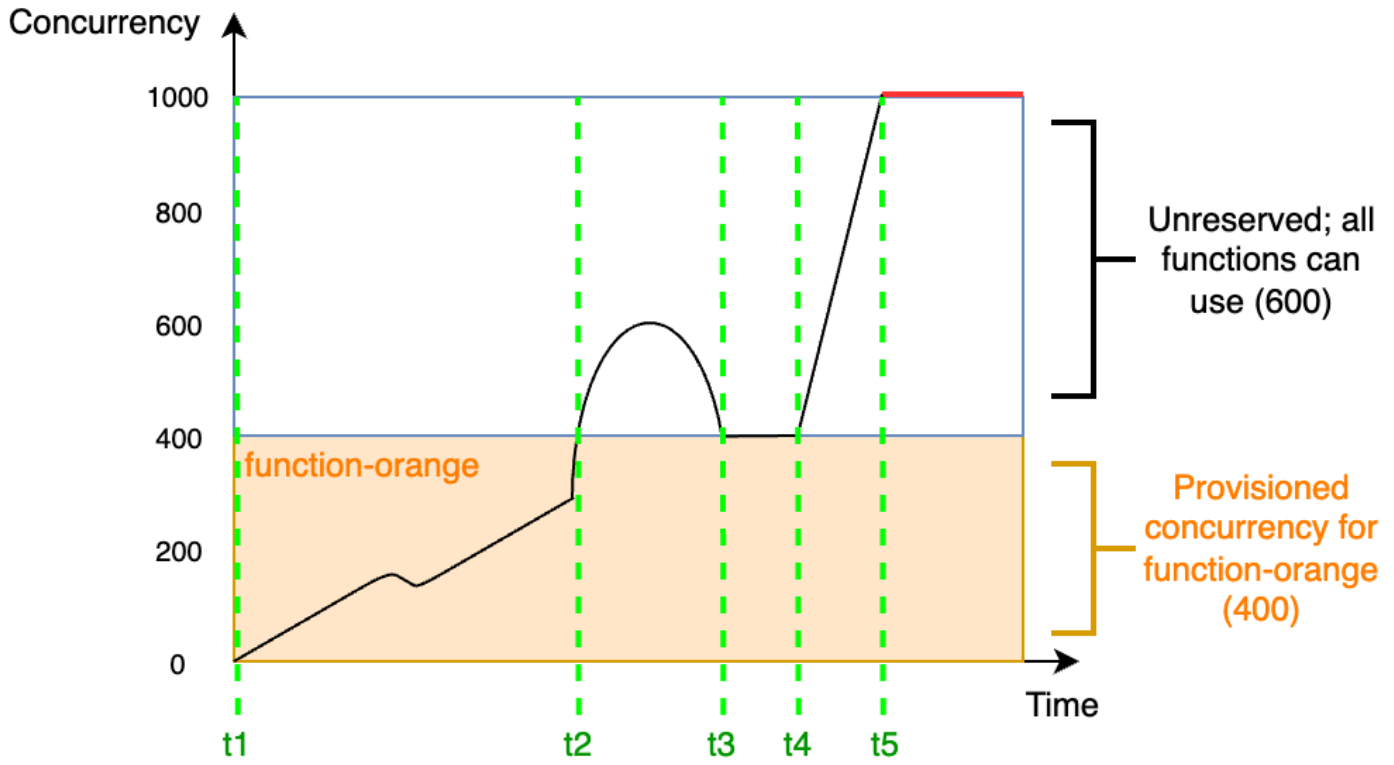
プロビジョニングされた同時実行を使用するときも、Lambda は引き続き実行環境をバックグラウンドでリサイクルしますが、Lambda は常に、事前に初期化された環境の数が、関数のプロビジョニングされた同時実行設定の値と等しくなることを確実にします。この動作は、アイドル状態が続いた後で Lambda が環境を完全に終了する場合がある、予約された同時実行とは異なります。以下の図は、予約された同時実行を使用して関数を設定する場合と、プロビジョニングされた同時実行を使用して関数を設定する場合における、単一の実行環境のライフサイクルを比較することで、これを説明しています。



この図には、次の 4 つの注目点があります。

時間	予約された同時実行	プロビジョニングされた同時実行
t1	何も実行されません。	Lambda が実行環境インスタンスを事前に初期化します。
t2	リクエスト 1 を受け取ります。Lambda は新しい実行環境インスタンスを初期化する必要があります。	リクエスト 1 を受け取ります。Lambda は事前に初期化された環境インスタンスを使用します。
t3	アイドル時間がしばらく続くと、Lambda がアクティブな環境インスタンスを終了します。	何も実行されません。
t4	リクエスト 2 を受け取ります。Lambda は新しい実行環境インスタンスを初期化する必要があります。	リクエスト 2 を受け取ります。Lambda は事前に初期化された環境インスタンスを使用します。

プロビジョニングされた同時実行に関する理解を深めるために、以下の図を検証しましょう。



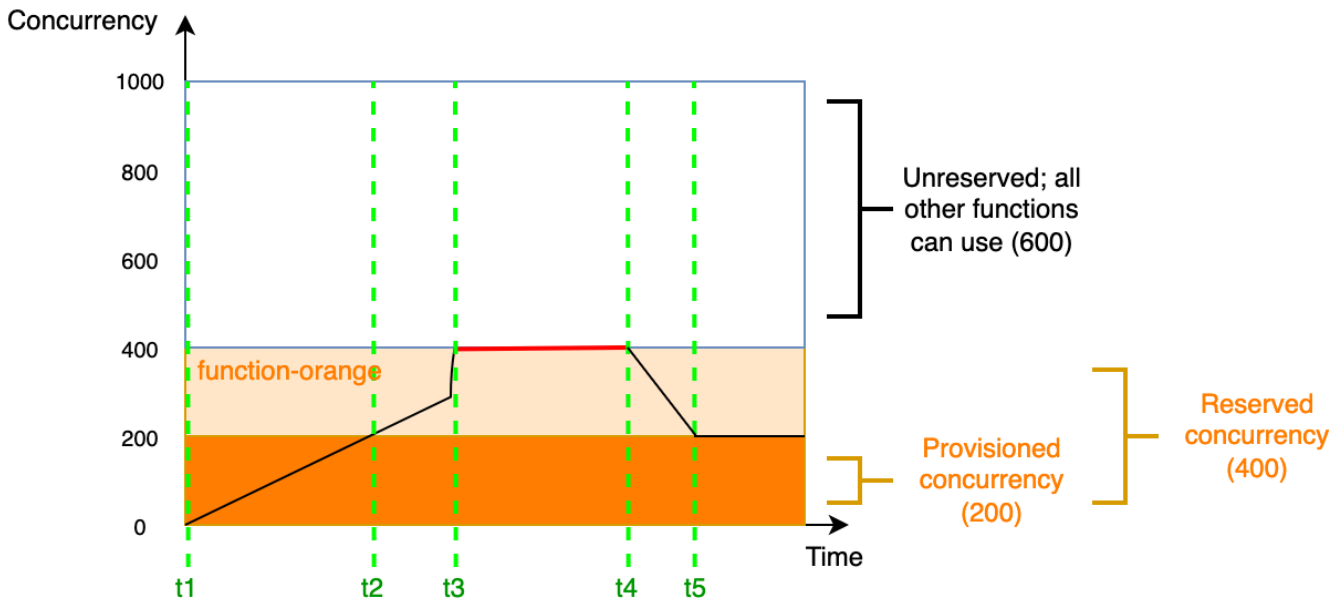
この図では、アカウントの同時実行上限が 1,000 になっています。function-orange にプロビジョニングされた同時実行を 400 ユニット設定することにしました。function-orange を含めたアカウント内のすべての関数が、残りの予約されていない同時実行 600 ユニットを使用できます。

この図には、以下の 5 つの注目点があります。

- t1 で、function-orange がリクエストの受け取りを開始します。Lambda には事前に初期化された実行環境インスタンスが 400 個あるので、function-orange は即時呼び出しに対応できます。
- t2 では、function-orange の同時リクエストが 400 件に到達します。その結果、function-orange はプロビジョニングされた同時実行を使い切ってしまいます。しかし、予約されていない同時実行がまだ残っているので、Lambda はこれを使用して function-orange への追加のリクエストを処理できます (スロットリングは行われません)。Lambda はこれらのリクエストを処理するために新しいインスタンスを作成する必要があり、関数でコールドスタートレイテンシーが発生する可能性があります。

- t3 では、トラフィックの一時的な急増後に function-orange の同時リクエスト数が 400 件に戻ります。Lambda は再び、コールドスタートレイテンシーなしですべてのリクエストを処理できるようになります。
- t4 で、アカウント内の関数のトラフィックが急増します。この急増は、function-orange、またはアカウント内の他の関数で発生したものになり得ます。Lambda は、予約されていない同時実行を使用して、これらのリクエストを処理します。
- t5 では、アカウント内の関数が同時実行上限の 1,000 に到達し、スロットリングが発生します。

上記の例では、プロビジョニングされた同時実行のみを検証しました。実際には、プロビジョニングされた同時実行と予約された同時実行の両方を関数に設定できます。これは、平日には一定量の呼び出しを処理する関数があるが、週末にはトラフィックの急増が定期的に発生するという場合に実行できます。この場合、プロビジョニングされた同時実行を使用して平日のリクエストを処理するための基本数の環境を設定し、予約された同時実行を使用して週末の急増を処理することができます。以下の図を検証しましょう。



この図では、function-orange のために、プロビジョニングされた同時実行を 200 ユニット、および予約された同時実行を 400 ユニット設定するとします。予約された同時実行を設定したので、function-orange が 600 ユニットの予約されていない同時実行を使用することはできません。

この図には、以下の 5 つの注目点があります。

- t1 で、function-orange がリクエストの受け取りを開始します。Lambda には事前に初期化された実行環境インスタンスが 200 個あるので、function-orange は即時呼び出しに対応できます。
- t2 で、function-orange はプロビジョニングされた同時実行のすべてを使い切ります。function-orange は予約された同時実行を使用してリクエストの処理を継続できますが、これらのリクエストではコールドスタートレイテンシーが発生する可能性があります。
- t3 では、function-orange の同時リクエストが 400 件に到達します。その結果、function-orange は予約された同時実行のすべてを使い切ります。function-orange は予約されていない同時実行を使用できないため、リクエストがスロットルされ始めます。
- t4 では、function-orange が受け取るリクエストの数が減り始め、スロットルされなくなります。
- t5 では、function-orange の同時リクエスト数が 200 件に減少するため、すべてのリクエストがプロビジョニングされた同時実行を再び使用できるようになります (つまり、コールドスタートレイテンシーは発生しません)。

予約された同時実行とプロビジョニングされた同時実行の両方が、アカウントの同時実行上限とリージョン別のクォータにカウントされます。つまり、予約された同時実行とプロビジョニングされた同時実行の割り当ては、他の関数が利用できる同時実行のプールに影響を与える可能性があります。プロビジョニング済み同時実行を設定すると、AWS アカウントに料金が請求されます。

> **Note**
>
> 関数の Versions and Aliases に割り当てたプロビジョニングされた同時実行数が、関数の予約された同時実行数に達すると、すべての呼び出しはプロビジョニングされた同時実行数を使用して実行されます。この設定には、非公開バージョンの関数 ($LATEST) をスロットリングする効果もあるため、その関数は実行されません。関数に対し、予約済同時実行数よりも多くのプロビジョニングされた同時実行を割り当てることはできません。

関数用にプロビジョニングされた同時実行の設定を管理するには、「関数に対するプロビジョニングされた同時実行数の設定」を参照してください。スケジュールまたはアプリケーションの使用状況に基づいてプロビジョニングされた同時実行のスケーリングを自動化するには、「Application Auto Scaling を使用してプロビジョニングされた同時実行数の管理を自動化する」を参照してください。

Lambda がプロビジョニングされた同時実行性を割り当てる方法

プロビジョニング済み同時実行は、設定後すぐにはオンラインになりません。Lambda は、1～2 分の準備後に、プロビジョニング済み同時実行の割り当てをスタートします。AWS リージョン とは関係なく、Lambda は、関数ごとに 1 分あたり最大 6,000 個の実行環境をプロビジョニングできます。これは関数の同時実行スケーリングレートとまったく同じです。

プロビジョニングされた同時実行を割り当てるリクエストを送信すると、Lambda が割り当てを完全に完了するまで、それらの環境にはアクセスできません。例えば、5,000 個のプロビジョニングされた同時実行をリクエストした場合、Lambda が 5,000 個の実行環境の配分を完全に終了するまで、リクエストでプロビジョニングされた同時実行を使用することはできません。

予約された同時実行とプロビジョニングされた同時実行の比較

以下の表は、予約された同時実行とプロビジョニングされた同時実行を要約し、比較したものです。

トピック	予約された同時実行	プロビジョニングされた同時実行
定義	関数の実行環境インスタンスの最大数。	関数用に事前にプロビジョニングされた、一定数の実行環境インスタンス。
プロビジョニング動作	Lambda が新しいインスタンスをオンデマンドベースでプロビジョニングします。	Lambda がインスタンスを事前に (つまり、関数がリクエストの受け取りを開始する前に) プロビジョニングします 。
コールドスタート動作	Lambda はオンデマンドで新しいインスタンスを作成する必要があることから、コールドスタートレイテンシーが発生する可能性があります。	Lambda はオンデマンドでインスタンスを作成する必要がないため、コールドスタートレイテンシーが発生することはありません。
スロットリング動作	予約された同時実行の上限に達すると、関数がスロットルされます。	予約された同時実行が設定されていない場合: プロビジョニングされた同時実行の上限に達すると、関数は予約されて

トピック	予約された同時実行	プロビジョニングされた同時実行
		いない同時実行を使用します。 予約された同時実行数が設定されている場合: 予約された同時実行の上限に達すると、関数がスロットルされます。
設定されていない場合のデフォルト動作	関数は、アカウントで利用できる、予約されていない同時実行を使用します。	Lambda はインスタンスを事前にプロビジョニングしません。その代わり、予約された同時実行が設定されていない場合、関数はアカウントで利用できる、予約されていない同時実行を使用します。 予約された同時実行が設定されている場合: 関数は予約された同時実行を使用します。
料金	追加料金はありません。	追加料金が発生します。

同時実行のクォータ

Lambda は、リージョン内のすべての関数全体で使用できる同時実行の合計数に対するクォータを設定しています。これらのクォータは、次の 2 つのレベルで設定されています。

- アカウントレベルでは、関数がデフォルトで最大 1,000 ユニットの同時実行を使用できます。クォータの引き上げをリクエストするには、「Service Quotas User Guide」(Service Quotas ユーザーガイド) の「Requesting a quota increase」(クォータ引き上げのリクエスト) を参照してください。
- 関数レベルでは、すべての関数全体で最大 900 ユニットの同時実行をデフォルトで予約できます。アカウントの同時実行数の合計制限に関係なく、Lambda は同時実行を明示的に予約しない関数に対して常に 100 ユニットの同時実行を予約します。例えば、アカウントの同時実行上限を 2,000 に引き上げた場合、関数レベルでは最大 1,900 ユニットの同時実行を予約できます。

現在のアカウントレベルの同時実行クォータを確認するには、AWS Command Line Interface (AWS CLI) を使用して以下のコマンドを実行します。

```
aws lambda get-account-settings
```

次のような出力が表示されます。

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

ConcurrentExecutions はアカウントレベルの同時実行クォータの合計です。UnreservedConcurrentExecutions は、まだ関数に割り当て可能な予約された同時実行の量です。

関数が受け取るリクエストが増えると、Lambda が実行環境数を自動的にスケールアップしてこれらのリクエストを処理します。これはアカウントが同時実行クォータに達するまで行われます。ただし、突然のトラフィック急増によるオーバースケーリングを防ぐために、Lambda では関数がスケールできる速度を制限しています。この同時実行のスケーリングレートは、アカウントの関数がリクエストの増加に応じてスケールインできる最大レートです。(つまり、Lambda がどれだけ速く新しい実行環境を作成できるかということです。) 同時実行のスケーリングレートは、アカウントレベルの同時実行数の上限 (関数で利用できる同時実行の合計量) とは異なります。

各 AWS リージョンおよび各関数の同時実行スケーリングレートは 10 秒ごとに 1,000 件の実行環境インスタンスです。つまり、10 秒ごとに、Lambda は各関数に最大 1,000 の追加実行環境インスタンスを割り当てることができます。

通常、この制限について心配する必要はありません。ほとんどのユースケースでは、Lambda のスケーリングレートで十分です。

重要なのは、同時実行のスケーリングレートは関数レベルの上限であることです。つまり、アカウント内の各関数は、他の関数とは別個にスケールできます。

スケーリング動作の詳細については、「Lambda のスケーリング動作」を参照してください。

関数に対する予約済み同時実行数の設定

Lambda での同時実行は、関数が現在処理している未完了のリクエストの数です。利用できる同時実行コントロールには、次の 2 種類があります。

- 予約された同時実行 — これは、関数に割り当てられた同時インスタンスの最大数です。ある関数が予約済み同時実行を使用している場合、他の関数はその同時実行を使用できません。予約済み同時実行数は、最も重要な関数が受信リクエストを処理するのに十分な同時実行数を常に確保するのに役立ちます。関数に対して予約される同時実行を設定する場合には追加料金がかかりません。
- プロビジョニングされた同時実行 — これは、関数に割り当てる、事前に初期化された実行環境の数です。これらの実行環境は、受信した関数リクエストに即座に対応できます。プロビジョニングされた同時実行数は、関数のコールドスタートレイテンシーを削減するのに役立ちます。プロビジョニングされた同時実行を設定すると、AWS アカウント に追加料金が請求されます。

このトピックでは、予約済み同時実行数を管理および設定する方法について説明します。これら 2 種類の同時実行制御の概念的な概要については、「予約同時実行とプロビジョニング同時実行」を参照してください。プロビジョニングされた同時実行数の設定の詳細については、the section called “プロビジョニング済み同時実行の設定” を参照してください。

> **Note**
>
> Amazon MQ イベントソースマッピングにリンクされた Lambda 関数には、デフォルトの最大同時実行数があります。Apache ActiveMQ の場合、同時インスタンスの最大数は 5 です。RabbitMQ の場合、同時インスタンスの最大数は 1 です。関数に予約またはプロビジョニングされる同時実行数を設定しても、これらの制限は変わりません。Amazon MQ を使用する場合のデフォルトの最大同時実行数の増加をリクエストするには、AWS Support にお問い合わせください。

セクション

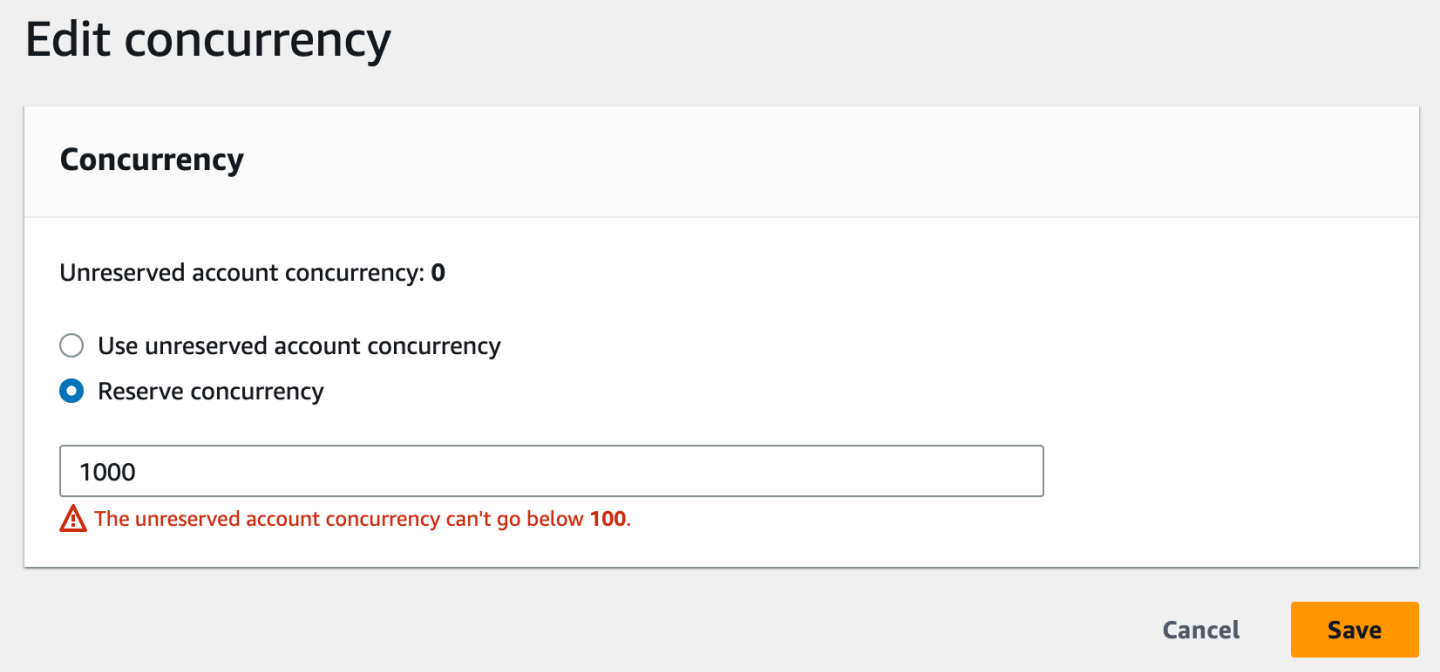
予約済同時実行数の設定

Lambda コンソールまたは Lambda API を使用して、関数の予約済同時実行の設定を行うことができます。

関数の同時実行数を予約するには (コンソール)

1. Lambda コンソールの [関数] ページを開きます。
2. 同時実行を予約する関数を選択します。
3. [設定]、[同時実行] の順にクリックします。
4. [同時実行数] で、[編集] をクリックします。
5. [同時実行の予約] をクリックします。関数用に予約する同時実行の量を入力します。
6. [保存] をクリックします。

[予約されていないアカウントの同時実行] の値から 100 を引いた値まで予約できます。残りの 100 単位の同時実行は、予約された同時実行を使用しない関数用です。例えば、アカウントの同時実行の上限が 1,000 の場合、1 つの関数で同時実行の 1,000 ユニットすべては予約できません。



関数の同時実行数を予約すると、他の関数で使用できる同時実行数のプールに影響を与えることがあります。たとえば、`function-a` の同時実行に 100 ユニットを予約した場合、`function-a` が予約された同時実行の 100 ユニットすべてを使用しなくても、アカウント内の他の関数は残りの 900 ユニットの同時実行を共有する必要があります。

意図的に関数をスロットリングするには、予約済同時実行数をゼロに設定します。これにより、制限を削除するまで、関数のイベント処理が停止します。

以下の API オペレーションを使用して、Lambda API で予約された同時実行を設定するには

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

例えば、AWS Command Line Interface (CLI) を使用して予約された同時実行を設定するには、`put-function-concurrency` コマンドを使用します。以下のコマンドでは、`my-function` という名前の関数に 100 の同時実行数を予約します。

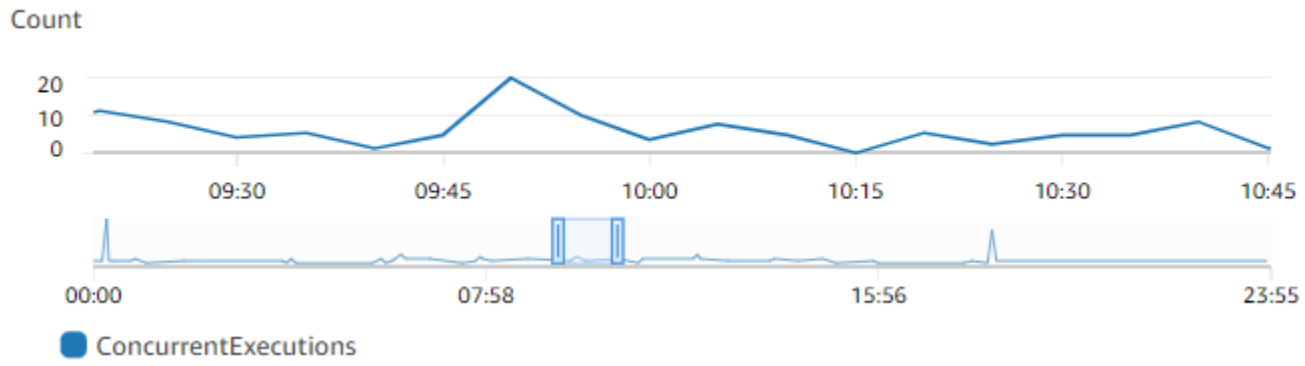
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

次のような出力が表示されます。

```
{
    "ReservedConcurrentExecutions": 100
}
```

関数に必要な予約済み同時実行数を正確に見積もる

関数が現在トラフィックを処理している場合は、CloudWatch メトリクスを使用して、その同時実行メトリクスを簡単に確認できます。特に、`ConcurrentExecutions` メトリクスはアカウント内の各関数の同時呼び出し数を表示します。



グラフによると、この関数は通常、1 日平均 5 ~ 10 件、最大で 20 件の同時リクエストを処理します。アカウントに、他にも多くの関数があるとします。この関数がアプリケーションにとって重要な関数で、リクエストを一切ドロップしたくない場合は、予約された同時実行の設定として 20 以上の数を使用します。

または、以下の式を使用して同時実行を計算できることを思い出してください。

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

1 秒あたりの平均リクエスト数に平均リクエスト時間 (秒単位) を掛けると、予約する必要がある同時実行数の大まかな見積もりが得られます。1 秒あたりの平均リクエスト数は `Invocation` メトリクスを、平均リクエスト時間 (秒単位) は `Duration` メトリクスを使用して見積もることができます。詳細については、「Lambda 関数のメトリクスの使用」を参照してください。

関数に対するプロビジョニングされた同時実行数の設定

Lambda での同時実行は、関数が現在処理している未完了のリクエストの数です。利用できる同時実行コントロールには、次の 2 種類があります。

- 予約された同時実行 — これは、関数に割り当てられた同時インスタンスの最大数です。ある関数が予約済み同時実行を使用している場合、他の関数はその同時実行を使用できません。予約済み同時実行数は、最も重要な関数が受信リクエストを処理するのに十分な同時実行数を常に確保するのに役立ちます。関数に対して予約される同時実行を設定する場合には追加料金がかかりません。
- プロビジョニングされた同時実行 — これは、関数に割り当てる、事前に初期化された実行環境の数です。これらの実行環境は、受信した関数リクエストに即座に対応できます。プロビジョニングされた同時実行数は、関数のコールドスタートレイテンシーを削減するのに役立ちます。プロビジョニングされた同時実行を設定すると、AWS アカウント に追加料金が請求されます。

このトピックでは、プロビジョニング済み同時実行を管理および設定する方法について説明します。これら 2 種類の同時実行制御の概念的な概要については、「予約済み同時実行とプロビジョニング済み同時実行」を参照してください。予約済み同時実行の設定の詳細については、「the section called “予約済同時実行数の設定”」を参照してください。

> **Note**
>
> Amazon MQ イベントソースマッピングにリンクされた Lambda 関数には、デフォルトの最大同時実行数があります。Apache ActiveMQ の場合、同時インスタンスの最大数は 5 です。RabbitMQ の場合、同時インスタンスの最大数は 1 です。関数に予約またはプロビジョニングされる同時実行数を設定しても、これらの制限は変わりません。Amazon MQ を使用する場合のデフォルトの最大同時実行数の増加をリクエストするには、AWS Support にお問い合わせください。

セクション

- Application Auto Scaling を使用してプロビジョニングされた同時実行数の管理を自動化する

プロビジョニング済み同時実行の設定

Lambda コンソールまたは Lambda API を使用して、関数のプロビジョニング済み同時実行の設定を行うことができます。

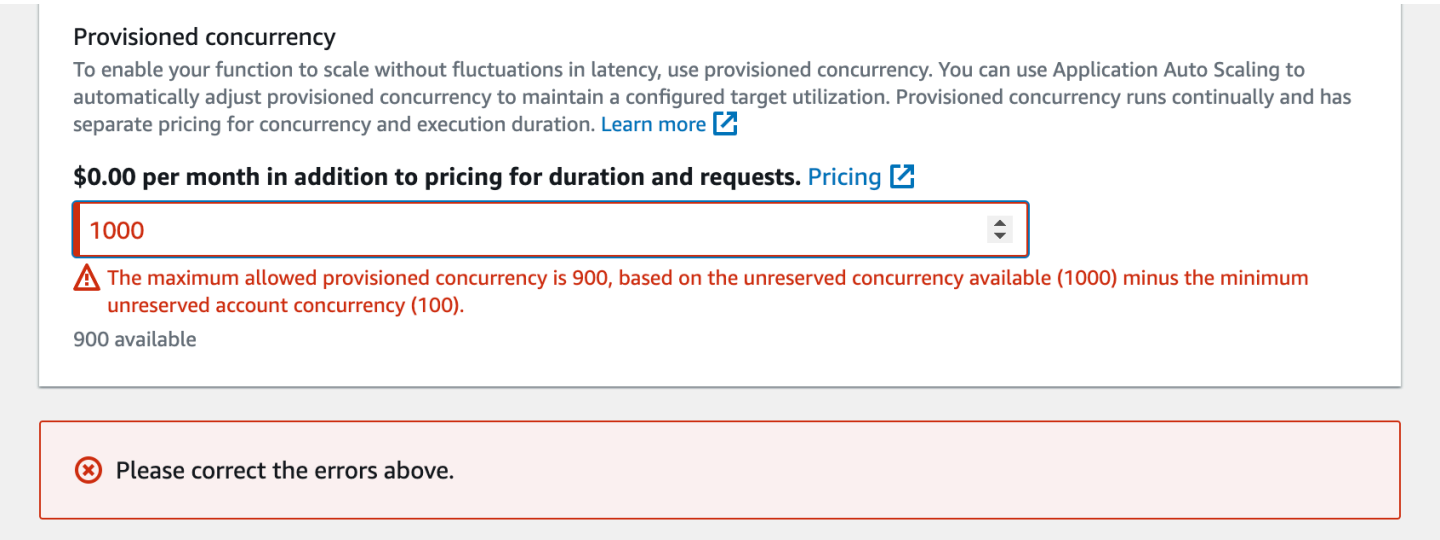
プロビジョニング済み同時実行を関数に割り当てるには (コンソール)

1. Lambda コンソールの「関数」ページを開きます。
2. プロビジョニング済み同時実行を割り当てる関数を選択します。
3. [設定]、[同時実行] の順にクリックします。
4. [プロビジョニング済み同時実行の設定] で、[設定を追加] をクリックします。
5. 修飾子のタイプ、エイリアスまたはバージョンを選択します。

> **Note**
>
> プロビジョニング済み同時実行は、どの関数の $LATEST バージョンでも使用できません。
> 関数にイベントソースがある場合は、そのイベントソースが正しい関数エイリアスまたはバージョンを指していることを確認してください。正しくない場合、関数はプロビジョニング済み同時実行環境を使用しません。

6. [プロビジョニング済み同時実行] に数値を入力します。Lambda は毎月の費用の見積もりを提供します。
7. [保存] を選択します。

アカウントで設定できるのは、予約なしアカウントの同時実行数から 100 を引いた数までです。残りの 100 単位の同時実行数は、予約済み同時実行を使用しない関数用です。たとえば、アカウントの同時実行数の制限が 1,000 で、他の関数に予約済みまたはプロビジョニング済み同時実行を割り当てていない場合、1 つの関数に対して最大 900 のプロビジョニング済み同時実行単位を設定できます。



ある関数にプロビジョニングされた同時実行を設定すると、他の関数で使用できる同時実行のプールに影響が生じます。例えば、`function-a` のプロビジョニングされた同時実行を 100 単位に設定すると、アカウント内の他の関数は残りの 900 単位の同時実行を共有する必要があります。これは、`function-a` が 100 ユニットすべてを使用しない場合にも当てはまります。

予約された同時実行とプロビジョニングされた同時実行の両方を同じ関数に割り当てることができます。このような場合、プロビジョニングされた同時実行は予約された同時実行を超えることはありません。

この制限は関数バージョンにも適用されます。特定の関数バージョンに割り当てることができるプロビジョニングされた同時実行の最大数は、その関数の予約された同時実行数から他の関数バージョンのプロビジョニングされた同時実行数を引いたものです。

Lambda API でプロビジョニング済み同時実行を設定するには、以下の API オペレーションを使用します。

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

たとえば、AWS Command Line Interface (CLI) でプロビジョニング済み同時実行を設定するには、`put-provisioned-concurrency-config` コマンドを使用します。以下のコマンドにより、`my-function` という名前の関数の BLUE エイリアスに、100 単位のプロビジョニング済み同時実行を割り当てます。

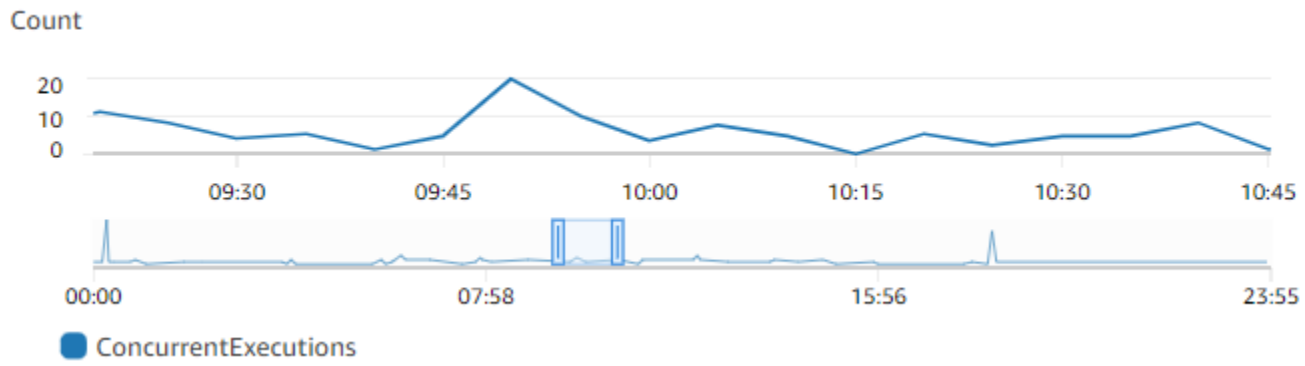
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

次のような出力が表示されます。

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

関数に必要なプロビジョニングされた同時実行数を正確に見積もる

CloudWatch メトリクスを使用して、アクティブな関数の同時実行メトリクスを表示できます。具体的には、ConcurrentExecutions メトリクスはアカウント内の関数の同時呼び出し数を表示します。



グラフによると、この関数は平均 5 ~ 10 件、最大で 20 件の同時リクエストを処理します。アカウントに、他にも多くの関数があるとします。この関数がアプリケーションにとって重要で、呼び出しのたびに低レイテンシーの応答が必要な場合は、プロビジョニングされた同時実行を 20 以上に設定します。

以下の式を使用して同時実行を計算することもできます。

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

必要な同時実行数を見積もるには、秒ごとの平均リクエスト数に平均リクエスト時間 (秒単位) を掛けます。1 秒あたりの平均リクエスト数は `Invocation` メトリクスを、平均リクエスト時間 (秒単位) は `Duration` メトリクスを使用して見積もることができます。

プロビジョニングされた同時実行を設定する際、Lambda では、関数が通常必要とする同時実行数の 10% を余分に含めることを推奨します。例えば、関数の同時リクエスト数が通常 200 件まで増加する場合は、プロビジョニングされた同時実行を 220 (200 件の同時リクエスト + 10% = 220 のプロビジョニングされた同時実行) に設定してください。

プロビジョニングされた同時実行数を使用する場合の関数コードの最適化

プロビジョニングされた同時実行数を使用する場合は、低レイテンシーを最適化するために関数コードを再編成することを検討してください。プロビジョニングされた同時実行数を使用する関数の場合、Lambda は割り当て時に初期化コード (ライブラリのロードやクライアントのインスタンス化など) を実行します。このため、実際の関数呼び出し中のレイテンシーへの影響を避けるために、メインの関数ハンドラーの外部で、できるだけ多くの初期化を行うことをお勧めします。対照的に、メインハンドラーコード内でライブラリを初期化したり、クライアントをインスタンス化したりすると、関数は呼び出しのたびにこれを実行する必要があります (これは、プロビジョニングされた同時実行数を使用しているかどうかに関らず発生します)。

オンデマンド呼び出しの場合、関数がコールドスタートするたびに、Lambda は初期化コードを再実行する必要がある場合があります。このような関数では、特定の機能の初期化処理を、関数が必要とするまで延期することができます。たとえば、次の Lambda ハンドラーの制御フローを考えてみます。

```
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

前の例では、開発者は、メインハンドラーの外部で `CLIENT_A` を初期化する代わりに、`if` ステートメント内で初期化を行いました。これにより、Lambda は `some_condition` の条件を満たした場合にのみこのコードを実行します。メインハンドラーの外部で `CLIENT_A` を初期化した場合、Lambda はコールドスタートのたびにそのコードを実行します。これにより、全体のレイテンシーが長くなる可能性があります。

環境変数を使用してプロビジョニングされた同時実行数の動作を表示および制御する

これにより、関数がプロビジョニング済み同時実行をすべて使い果たしてしまう可能性があります。Lambda はオンデマンドインスタンスを使用して過剰なトラフィックを処理します。Lambda が特定の環境で使用した初期化のタイプを特定するには、AWS_LAMBDA_INITIALIZATION_TYPE 環境変数の値を確認してください。この変数は、provisioned-concurrency または on-demand の値を取る可能性があります。AWS_LAMBDA_INITIALIZATION_TYPE の値は不変で、環境の存続期間を通じて一定に保たれます。関数コード内の環境変数の値を確認するには、「???」を参照してください。

.NET 6 または .NET 7 ランタイムを使用する場合は、プロビジョニングされた同時実行を使用しない場合でも、AWS_LAMBDA_DOTNET_PREJIT 環境変数を設定して、関数のレイテンシーを改善できます。.NET ランタイムでは、コードが初めて呼び出すライブラリごとに遅延コンパイルと初期化を行います。この結果、Lambda 関数の呼び出しが、その後の呼び出しよりも長くかかる場合があります。これを軽減するには、AWS_LAMBDA_DOTNET_PREJIT の 3 つの値から 1 つを選択します。

- ProvisionedConcurrency: Lambda は、プロビジョニング済み同時実行を使用して、すべての環境に対して事前 JIT コンパイルを実行します。これは、デフォルト値です。
- Always: Lambda は、関数がプロビジョニング済み同時実行を使用しない場合でも、すべての環境に対して事前 JIT コンパイルを実行します。
- Never: Lambda は、すべての環境に対して事前 JIT コンパイルを無効にします。

プロビジョニングされた同時実行数を使用したログ記録と請求動作について

プロビジョニングされた同時実行環境では、関数の初期化コードは、割り当て時と、Lambda が環境のインスタンスをリサイクルする際に定期的に実行されます。環境インスタンスでリクエストが処理された後、ログとトレースで初期化時間を確認できます。Lambda では、環境インスタンスがリクエストを処理しない場合でも、初期化に対して課金されることに注意してください。プロビジョニングされた同時実行は継続的に実行され、初期化コストと呼び出しコストとは別に課金されます。詳細については、「AWS Lambda の料金」を参照してください。

また、 Lambda 関数のプロビジョニングされた同時実行を設定すると、Lambda はその実行環境を事前に初期化して、関数呼び出しリクエストの前に使用できるようにします。ただし、関数が実際に呼び出された場合にのみ、関数は呼び出しログを CloudWatch に発行します。したがって、初期化が事前に行われた場合でも、Init Duration フィールドは最初の関数呼び出しの REPORT ログ行に表示されます。これは、関数でコールドスタートが発生したことを意味するものではありません。

Application Auto Scaling を使用してプロビジョニングされた同時実行数の管理を自動化する

Application Auto Scaling を使用すると、スケジュールに従って、または使用率に基づいて、プロビジョニング済み同時実行を管理できます。関数が予測可能なトラフィックパターンを取得した場合は、スケジュールされたスケーリングを使用します。関数で特定の使用率を維持したい場合は、ターゲット追跡スケーリングポリシーを使用します。

スケジュールされたスケーリング

Application Auto Scaling により、予測可能な負荷の変化に従って独自のスケーリングスケジュールを設定できます。詳細と例については、「Application Auto Scaling ユーザーガイド」の「Application Auto Scaling のスケジュールされたスケーリング」、および「AWS コンピューティングブログ」の「定期的なピーク使用量に対する AWS Lambda のプロビジョニング済み同時実行のスケジューリング」を参照してください。

ターゲット追跡

ターゲット追跡では、Application Auto Scaling がスケーリングポリシーの定義方法に基づいて、一連の CloudWatch アラームを作成および管理します。これらのアラームがアクティブになると、Application Auto Scaling はプロビジョニング済み同時実行を使用して、割り当てられる環境の量を自動的に調整します。トラフィックパターンが予測できないアプリケーションには、ターゲット追跡を使用してください。

ターゲットトラッキングを使用してプロビジョニング済み同時実行をスケールするには、`RegisterScalableTarget` および `PutScalingPolicy` の Application Auto Scaling API オペレーションを使用します。たとえば、AWS Command Line Interface (CLI) を使用している場合は、次の手順に従います。

1. 関数のエイリアスをスケーリングターゲットとして登録します。次の例では、`my-function` という名前の関数の BLUE エイリアスを登録します。

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. スケーリングポリシーをターゲットに適用します。次の例では、アプリケーションの自動スケーリングを設定し、エイリアスのプロビジョニングされた同時実行の設定を調整することにより、使用率を 70% 近くに維持します。ただし、10% から 90% までの間の値を適用できます。

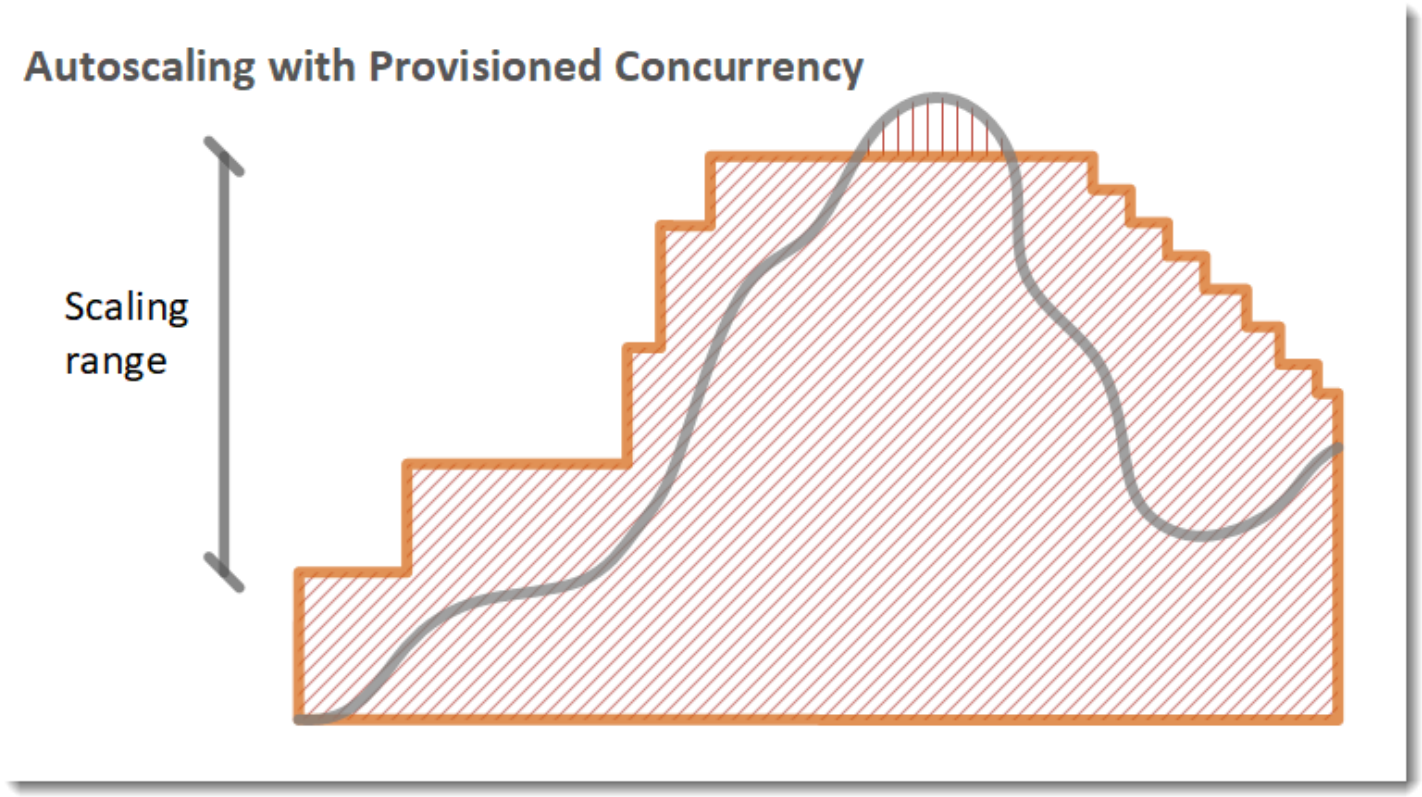
```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

次のような出力が表示されます。

```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling は CloudWatch に 2 つのアラームを作成します。最初のアラームは、プロビジョニング済み同時実行の使用率が一貫して 70% を超えたときにトリガーされます。これにより、Application Auto Scaling はプロビジョニング済み同時実行の割り当て数を増やして、使用率を下げます。2 番目のアラームは、使用率が一貫して 63% (70% ターゲットの 90%) を下回った場合にトリガーされます。これにより、Application Auto Scaling はエイリアスのプロビジョニング済み同時実行数を減らします。

次の例では、関数が使用率に基づいて、プロビジョニング済み同時実行の最小値と最大値の間でスケールします。



凡例

- 関数のインスタンス数
- 未処理のリクエスト数
- プロビジョニング済み同時実行
- 標準の同時実行数

オープンリクエストの数が増加すると、Application Auto Scaling は設定された最大値に達するまで、プロビジョニング済み同時実行数を大規模なステップで増加させます。この後、アカウントの同時実行数の上限に達していなければ、関数は引き続き標準の予約されていない同時実行でスケールできます。使用率が下がり、低いまま続く場合、Application Auto Scaling はプロビジョニングされた同時実行数を減少させるために、より小規模な定期的ステップを実行します。

Application Auto Scaling アラームはどちらも、デフォルトで平均統計を使用します。トラフィックのクイックバーストが起こる関数は、これらのアラームをトリガーしない可能性があります。例えば、Lambda 関数が瞬間的に (20 ~ 100 ミリ秒) 実行され、トラフィックがクイックバーストしたとします。この場合、バースト中はリクエスト数が割り当てられた同時実行数を超えます。ただし、Application Auto Scaling では、追加の環境をプロビジョニングするために、バーストロードを少なくとも 3 分間維持する必要があります。さらに、両方の CloudWatch アラームでは、自動スケーリングポリシーを有効にするために、ターゲット平均に達している 3 つのデータポイントが必要です。関数でトラフィックが急増する場合、平均統計の代わりに最大統計を使用する方が、プロビジョニングされた同時実行をスケーリングしてコールドスタートを最小限に抑えるのに効果的です。

ターゲット追跡スケーリングポリシーの詳細については、「Application Auto Scaling のターゲット追跡スケーリングポリシー」を参照してください。

Lambda のスケーリング動作

関数が受け取るリクエストが増えると、Lambda が実行環境数を自動的にスケールアップしてこれらのリクエストを処理します。これはアカウントが同時実行クォータに達するまで行われます。ただし、突然のトラフィック急増によるオーバースケーリングを防ぐために、Lambda では関数がスケールできる速度を制限しています。この同時実行スケーリングレートは、リクエストの増加に応じてアカウント内の関数をスケールできる最大レートです。(つまり、Lambda がどれだけ速く新しい実行環境を作成できるかということです。) 同時実行スケーリングレートは、関数で使用できる同時実行の合計量であるアカウントレベルの同時実行制限とは異なります。

同時実行スケーリングレート

各 AWS リージョンおよび各関数の同時実行スケーリングレートは 10 秒ごとに 1,000 件の実行環境インスタンスです。つまり、10 秒ごとに、Lambda は各関数に最大 1,000 の追加実行環境インスタンスを割り当てることができます。

通常、この制限について心配する必要はありません。ほとんどのユースケースでは、Lambda のスケーリングレートで十分です。

重要なのは、同時実行スケーリングレートは関数レベルの制限です。つまり、アカウント内の各関数は、他の関数とは別個にスケールできます。

> **Note**
>
> 実際には、Lambda は 10 秒ごとに 1,000 ユニットを 1 回補充するのではなく、同時実行のスケーリングレートを時間経過と共に継続的に補充するよう最大限試みます。

Lambda では、同時実行スケーリングレートの未使用分が発生しません。つまり、どの時点でも、スケーリングレートは常に、最大 1,000 ユニットの同時実行です。例えば、10 秒間隔で利用可能な 1,000 ユニットの同時実行をまったく使用しなかった場合、次の 10 秒間隔で 1,000 ユニットが追加で発生することはありません。次の 10 秒間の同時実行スケーリングレートは 1,000 のままです。

関数が受け取るリクエストの数が増え続ける限り、Lambda は利用可能な最速のレート (アカウントの同時実行数の上限まで) でスケールします。予約された同時実行数を設定することで、個々の関数が使用できる同時実行の量を制限できます。リクエストが入ってくるスピードに関数のスケールが追いつけない場合、または関数が同時実行数の最大値に達した場合は、追加リクエストはスロットルエラーで失敗します (ステータスコード 429)。

同時実行のモニタリング

Lambda は、関数の同時実行数のモニタリングに役立つ Amazon CloudWatch メトリクスを出力します。このトピックでは、これらのメトリクスとその解釈方法について説明します。

セクション

一般的な同時実行メトリクス

Lambda 関数の同時実行のモニタリングには、以下のメトリクスを使用します。各メトリクスの詳細度は 1 分です。

- ConcurrentExecutions - 特定時点のアクティブな同時実行呼び出し数。Lambda は、このメトリクスをすべての関数、バージョン、エイリアスに対して出力します。Lambda コンソールのどの関数であっても、Lambda は [モニタリング] タブの [メトリクス] に ConcurrentExecutions のグラフをネイティブに表示します。MAX を使用してこのメトリクスを表示します。
- UnreservedConcurrentExecutions – 予約されていない同時実行を使用している、アクティブな同時呼び出しの数。Lambda は、リージョン内のすべての関数にこのメトリクスを出力します。MAX を使用してこのメトリクスを表示します。
- ClaimedAccountConcurrency — オンデマンド呼び出しでは使用できない同時実行の量。ClaimedAccountConcurrency は、UnreservedConcurrentExecutions に割り当てられた同時実行数を加えたものに等しくなります (つまり、予約された同時実行数の合計にプロビジョニングされた同時実行数の合計を加えたもの)。ClaimedAccountConcurrency がアカウントの同時実行数の制限を超える場合は、アカウントの同時実行数の上限を引き上げるようにリクエストできます。MAX を使用してこのメトリクスを表示します。詳細については、「ClaimedAccountConcurrency メトリクスの処理」を参照してください。

プロビジョニングされた同時実行数のメトリクス

プロビジョニングされた同時実行を使用する Lambda 関数のモニタリングには、以下のメトリクスを使用します。各メトリクスの詳細度は 1 分です。

- ProvisionedConcurrentExecutions – プロビジョニングされた同時実行で呼び出しをアクティブに処理している実行環境インスタンスの数。Lambda は、プロビジョニングされた同時実行が設定された関数バージョンとエイリアスごとにこのメトリクスを出力します。MAX を使用してこのメトリクスを表示します。

ProvisionedConcurrentExecutions は、割り当てるプロビジョニングされた同時実行の合計数と同じではありません。例えば、100 ユニットのプロビジョニングされた同時実行を関数バージョンに割り当てるとします。任意の 1 分間で、100 個の実行環境のうち最大 50 個が同時に呼び出しを処理していた場合、MAX (ProvisionedConcurrentExecutions) の値は 50 になります。

- ProvisionedConcurrentInvocations – Lambda がプロビジョニングされた同時実行を使用して関数コードを呼び出す回数。Lambda は、プロビジョニングされた同時実行が設定された関数バージョンとエイリアスごとにこのメトリクスを出力します。SUM を使用してこのメトリクスを表示します。

ProvisionedConcurrentInvocations は呼び出しの総数をカウントしますが、ProvisionedConcurrentExecutions はアクティブな環境の数をカウントするという点で、ProvisionedConcurrentInvocations は ProvisionedConcurrentExecutions と異なります。この違いを理解するために、次のシナリオを考えてみます。

この例では、1 分あたり 1 回の呼び出しを受け取り、各呼び出しの完了までに 2 分かかるとします。オレンジ色の横棒はそれぞれ 1 つのリクエストを表します。この関数に 10 ユニットのプロビジョニングされた同時実行を割り当て、各リクエストがプロビジョニングされた同時実行で実行されるとします。

0 分と 1 分の間に Request 1 が入ります。1 分の時点では、それまでの 1 分間でアクティブだった実行環境は最大 1 つなので、MAX (ProvisionedConcurrentExecutions) の値は 1 です。それまでの 1 分間に入った新しいリクエストは 1 件だったため、SUM (ProvisionedConcurrentInvocations) の値も 1 です。

1 分から 2 分の間に Request 2 が入り、Request 1 は実行し続けます。2 分の時点では、それまでの 1 分間でアクティブだった実行環境は最大 2 つなので、MAX (ProvisionedConcurrentExecutions) の値は 2 です。ただし、それまでの 1 分間に入った新しいリクエストは 1 件のみだったため、SUM (ProvisionedConcurrentInvocations) の値は 1 です。このメトリクスの動作は、この例の終わりまで継続します。

- ProvisionedConcurrencySpilloverInvocations – プロビジョニングされた同時実行がすべて使用されているときに、Lambda が標準的な同時実行 (予約された同時実行または予約されていない同時実行) で関数を呼び出す回数。Lambda は、プロビジョニングされた同時実行が設定された関数バージョンとエイリアスごとにこのメトリクスを出力します。SUM を使用してこのメトリクスを表示します。ProvisionedConcurrentInvocations + ProvisionedConcurrencySpilloverInvocation 値は、関数呼び出しの総数 (Invocations メトリクス) と一致するはずです。

  ProvisionedConcurrencyUtilization - 使用中のプロビジョニングされた同時実行の割合 (ProvisionedConcurrentExecutions をプロビジョニングされた割り当て同時実行の合計量で割った値)。Lambda は、プロビジョニングされた同時実行が設定された関数バージョンとエイリアスごとにこのメトリクスを出力します。MAX を使用してこのメトリクスを表示します。

例えば、100 ユニットのプロビジョニングされた同時実行を関数バージョンにプロビジョニングするとします。任意の 1 分間で、100 個の実行環境のうち最大 60 個が同時に呼び出しを処理していた場合、MAX (ProvisionedConcurrentExecutions) の値は 60 に、MAX (ProvisionedConcurrentUtilization) の値は 0.6 になります。

ProvisionedConcurrencySpilloverInvocations の値が大きい場合は、関数にプロビジョニングされた同時実行を追加で割り当てる必要があることを示している可能性があります。または、事前定義されたしきい値に基づいてプロビジョニングされた同時実行の自動スケーリングを処理するように Application Auto Scaling を設定することもできます。

逆に、ProvisionedConcurrencyUtilization の値が常に低い場合は、関数にプロビジョニングされた同時実行数を過剰に割り当てている可能性があります。

ClaimedAccountConcurrency メトリクスの処理

Lambda は ClaimedAccountConcurrency メトリックスを使用して、アカウントがオンデマンド呼び出しに使用できる同時実行の量を判断します。Lambda は、以下の式を使用して ClaimedAccountConcurrency を計算します。

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

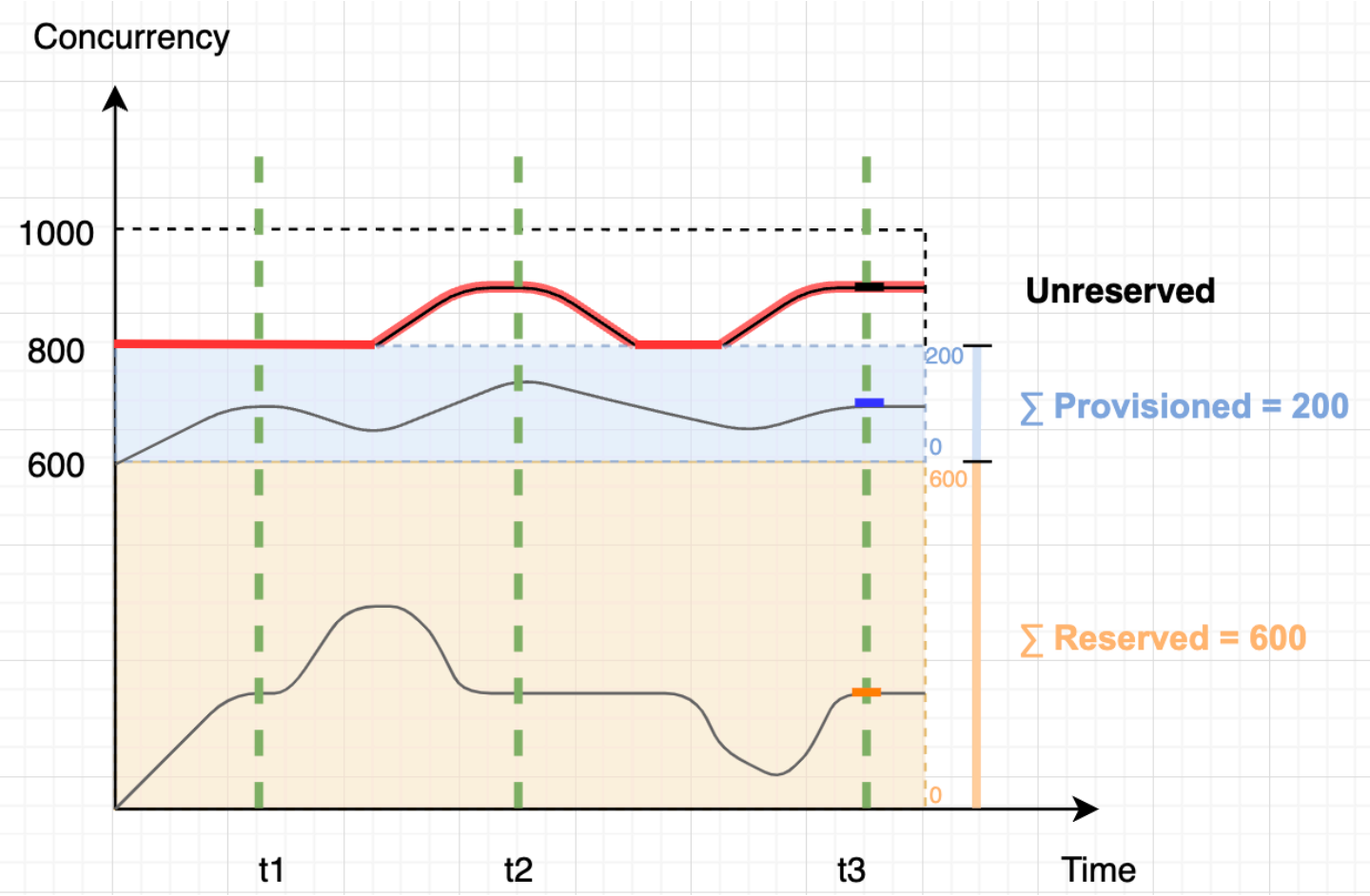
UnreservedConcurrentExecutions は、予約されていない同時実行を使用している、アクティブな同時呼び出しの数です。割り当てられた同時実行数は、次の 2 つの部分の合計です (RC は「予約された同時実行」、PC は「プロビジョニングされた同時実行」に置き換えます)。

- リージョン内のすべての関数の合計 RC。
- リージョン内で PC を使用しているすべての関数の合計 PC。RC を使用する関数は除きます。

> **Note**
>
> 1 つの関数に RC よりも多い PC を割り当てることはできません。したがって、関数の RC は常に、その関数の PC と同数かそれよりも多くなります。Lambda では、PC および RC によるこのような関数に割り当てられた同時実行性への寄与度を算出する際に、この 2 つのうちの最大値である、RC のみを考慮します。

Lambda は ConcurrentExecutions ではなく ClaimedAccountConcurrency メトリックスを使用して、アカウントがオンデマンド呼び出しに使用できる同時実行の量を判断します。ConcurrentExecutions メトリックスはアクティブな同時呼び出しの数を追跡するのに役立ちますが、実際の同時実行の可用性を必ずしも反映しているわけではありません。これは、Lambda では予約された同時実行とプロビジョニングされた同時実行も考慮して可用性を判断するためです。

ClaimedAccountConcurrency の例として、ほとんど使用されなくなっている関数全体で、多くの予約された同時実行とプロビジョニングされた同時実行を設定するシナリオを考えてみましょう。次の例では、アカウントの同時実行数の上限が 1,000 で、アカウントにとの 2 つの主要関数 function-orange、function-blue があるとします。function-orange には 600 単位の予約された同時実行を割り当てます。function-blue には 200 単位のプロビジョニングされた同時実行を割り当てます。時間の経過とともに、追加の関数をデプロイし、以下のトラフィックパターンに注目したとします。



前の図では、黒い線は時間の経過に伴う実際の同時実行使用量を示し、赤い線は時間の経過に伴う ClaimedAccountConcurrency 値を示しています。このシナリオでは、関数全体の実際の同時実行使用率は低いものの、ClaimedAccountConcurrency は最低でも 800 です。これは、function-orange と function-blue に合計 800 の同時実行ユニットを割り当てたためです。Lambda の観点からすると、ユーザーがこの同時実行性の使用を「要求」しているため、他の関数に使用できる同時実行ユニットは 200 ユニットしか残っていないことになります。

このシナリオでは、ClaimedAccountConcurrency 計算式で割り当てられる同時実行数は 800 です。これで、ダイアグラム内のさまざまなポイントでの ClaimedAccountConcurrency 値を導き出すことができます。

- t1 では、ClaimedAccountConcurrency は 800 (800 + 0 UnreservedConcurrentExecutions) です。
- t2 では、ClaimedAccountConcurrency は 900 (800 + 100 UnreservedConcurrentExecutions) です。

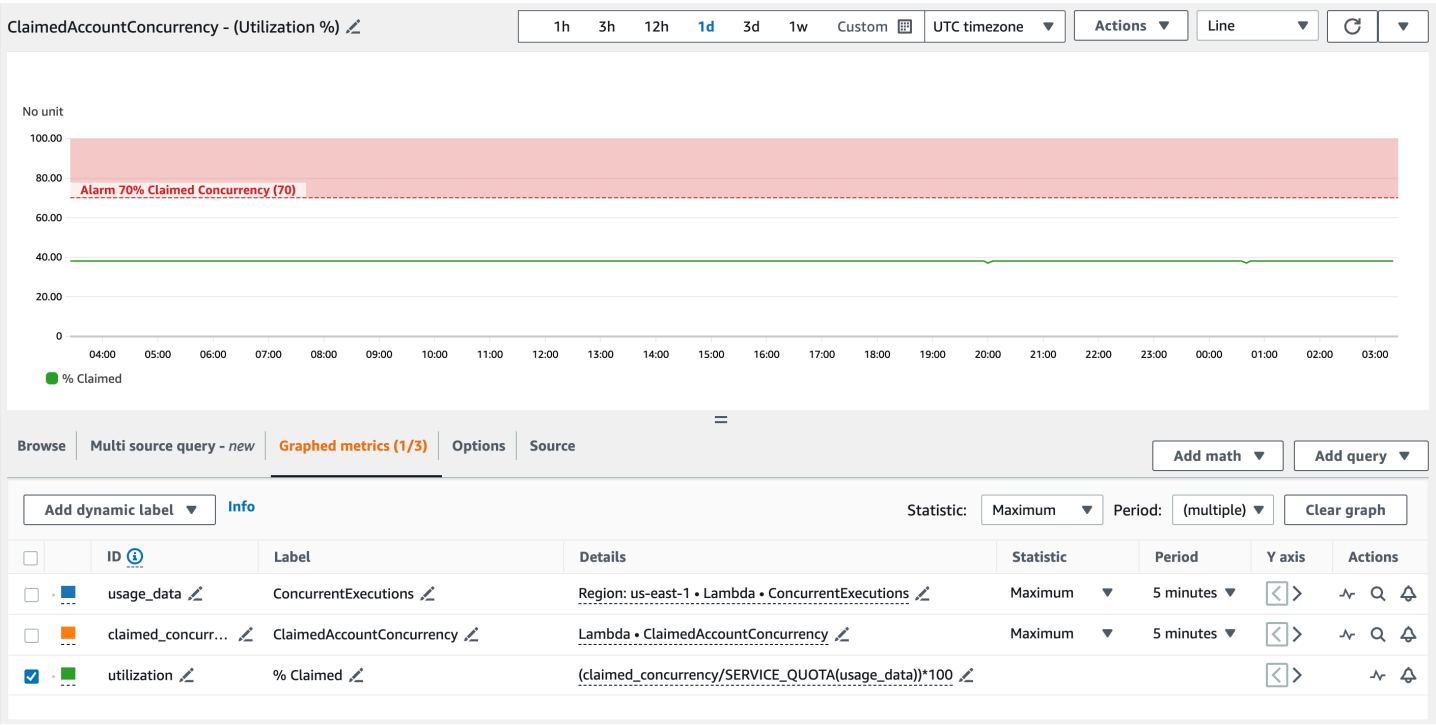
- t3 でも、ClaimedAccountConcurrency は 900 (800 + 100 UnreservedConcurrentExecutions) です。

での **ClaimedAccountConcurrency**メトリクスのセットアップ CloudWatch

Lambda は で ClaimedAccountConcurrencyメトリクスを出力します CloudWatch。次の式に示すように、このメトリクスを SERVICE_QUOTA(ConcurrentExecutions) の値とともに使用すると、アカウントの同時実行使用率を取得できます。

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

次のスクリーンショットは、 でこの式をグラフ化する方法を示しています CloudWatch。緑の claim_utilization 線は、このアカウントの同時実行使用率が約 40% であることを表しています。



前のスクリーンショットには、同時実行使用率が 70% を超えたときに ALARM状態になる CloudWatch アラームも含まれています。ClaimedAccountConcurrency メトリックスと類似のアラームを併用することで、アカウントの同時実行制限の引き上げをいつリクエストする必要があるかを事前に判断できます。

AWS Lambda でのコード署名の設定

AWS Lambda でコード署名を使用すると、信頼できるコードのみを Lambda 関数で実行するようにできます。関数のコード署名を有効にすると、デプロイされたすべてのコードが Lambda により チェックされ、コードパッケージが信頼できるソースによって署名されていることを確認できます。

> **Note**
>
> コンテナイメージとして定義された関数では、コード署名はサポートされません。

コードの整合性を検証するには、AWS Signer を使用して、関数とレイヤーのために、デジタル署名されたコードパッケージを作成します。ユーザーがコードパッケージのデプロイを試みると、そのデプロイが受け入られる前に、コードパッケージの検証チェックが Lambda により実行されます。コード署名の検証チェックはデプロイ時に実行されるため、関数の実行時には、パフォーマンスへの影響はありません。

AWS Signer を使用すると、署名プロファイルを作成することもできます。署名付きコードパッケージを作成するには、署名プロファイルを使用します。コードパッケージに署名できるユーザーを制御し、署名プロファイルを作成するには、AWS Identity and Access Management (IAM) を使用します。詳細については、AWS Signer デベロッパーガイドの「認証とアクセスコントロール」を参照してください。

関数でコード署名を有効にするには、コード署名の設定を作成し、その設定を関数にアタッチします。コード署名の設定では、許可済みの署名プロファイルのリストと、検証チェックが失敗した場合に実行するポリシーアクションを定義します。

Lambda のレイヤーは、関数のコードパッケージと同じ形式の、署名付きのコードパッケージに従います。コード署名が有効化された関数にユーザーがレイヤーを追加すると、そのレイヤーが許可された署名プロファイルによって署名されていることを Lambda がチェックします。関数に対してコード署名を有効にする場合、関数に追加されるすべてのレイヤーも、許可された署名プロファイルの 1 つによって署名されている必要があります。

IAM を使用して、コード署名の設定を作成できるユーザーを制御します。通常、このための権限は、特定の管理者ユーザーのみに付与します。さらに、コード署名が有効化された関数のみをデベロッパーが作成できるようにする IAM ポリシーを設定できます。

AWS CloudTrail への変更を記録するようにコード署名を構成できます。関数の成功したデプロイとブロックされたデプロイが、署名と検証チェックに関する情報とともに CloudTrail に記録されます。

関数のコード署名の設定には、Lambda コンソール、AWS Command Line Interface (AWS CLI)、AWS CloudFormation、AWS Serverless Application Model (AWS SAM) を使用します。

AWS での AWS Lambda Signer またはコード署名の使用には、追加料金は必要ありません。

セクション

署名の検証

ユーザーが署名付きコードパッケージを関数にデプロイすると、Lambda により次の検証チェックが実行されます。

1. 整合性 — コードパッケージが、署名されてから変更されていないことを検証します。Lambda が、パッケージのハッシュと署名からのハッシュを比較します。
2. 有効期限 – コードパッケージの署名が有効期限切れになっていないことを検証します。
3. 不一致 – Lambda 関数用に許可された署名プロファイルのうちの 1 つでコードパッケージが署名されていることを検証します。署名が存在しない場合にも不一致が発生します。
4. 失効 – コードパッケージの署名が無効でないことを検証します。

コード署名の設定で定義されている署名検証ポリシーによって、検証チェックが失敗した場合に、次のアクションのうちどれが Lambda により実行されるかが決定されます。

- 警告 — Lambda はコードパッケージのデプロイを許可しますが、警告を発行します。Lambda は、新しい Amazon CloudWatch メトリクスを発行し、警告を CloudTrail ログに保存します。

- Enforce (強制) – Lambda は (警告アクションと同じように) 警告を発行し、コードパッケージのデプロイをブロックします。

有効期限、不一致、および失効の各検証チェックのためにポリシーを設定できます。整合性検証には、ポリシーを構成できないことに注意してください。整合性のチェックが失敗した場合は、Lambda がデプロイをブロックします。

構成の前提条件

Lambda 関数用のコード署名を設定する前に、AWS Signer を使用して次の操作を実行します。

- 1 つまたは複数の署名プロファイルを作成します。
- 署名プロファイルを使用して、関数の署名付きコードパッケージを作成します。

詳細については、AWS Signer デベロッパーガイドの 「署名プロファイルの (コンソールでの) 作成」を参照してください。

コード署名の設定を作成する

コード署名の設定では、許可された署名プロファイルと署名検証ポリシーのリストを定義します。

コード署名設定を (コンソールで) 作成するには、

1. Lambda コンソールの [Code signing configurations (コード署名設定)] ページ)を開きます。
2. [Create configuration] (設定を作成) をクリックします。
3. [説明] に、その設定のための、分かりやすい名前を入力します。
4. [署名プロファイル] に、設定のための署名プロファイルを (最大 20 個まで) 追加します。
   a. [署名プロファイルバージョンの ARN] から、使用するプロファイルバージョンの Amazon リソースネーム (ARN) を選択します。あるいは、そこに ARN を入力します。
   b. 署名プロファイルを追加するには、[署名プロファイルの追加] をクリックします。
5. [署名の検証ポリシー] で、[警告] または [強制] を選択します。
6. [Create configuration] (設定を作成) をクリックします。

コード署名の設定を更新する

コード署名の設定を更新すると、その設定がアタッチされている関数で行われる、将来のデプロイに対し内容が反映されます。

コード署名の設定を (コンソールで) 更新するには、

1. Lambda コンソールの [Code signing configurations (コード署名設定)] ページ)を開きます。
2. 更新するコード署名の設定を選択し、[編集] をクリックします。
3. [説明] に、その設定のための、分かりやすい名前を入力します。
4. [署名プロファイル] に、設定のための署名プロファイルを (最大 20 個まで) 追加します。
   a. [署名プロファイルバージョンの ARN] から、使用するプロファイルバージョンの Amazon リソースネーム (ARN) を選択します。あるいは、そこに ARN を入力します。
   b. 署名プロファイルを追加するには、[署名プロファイルの追加] をクリックします。
5. [署名の検証ポリシー] で、[警告] または [強制] を選択します。
6. [Save changes] (変更の保存) をクリックします。

コード署名の設定を削除する

コード署名の設定を削除できるのは、その設定が、いずれの関数でも使用されていない場合のみです。

コード署名の設定を (コンソールで) 削除するには

1. Lambda コンソールの [Code signing configurations (コード署名設定)] ページ)を開きます。
2. 削除するコード署名の設定を選択し、[削除] をクリックします。
3. 確認のために、もう一度 [削除] をクリックします。

関数のコード署名を有効化する

関数のコード署名を有効にするには、コード署名の設定を関数に関連付けます。

コード署名の設定を (コンソールで) 関数に関連付けるには

1. Lambda コンソールの [関数ページ] を開きます。
2. コード署名を有効にする関数を選択します。

3. [Configuration] (設定) タブを開きます。
4. 下にスクロールし、[コード署名] を選択します。
5. [編集] を選択します。
6. [コード署名の編集] で、対象の関数に関連付けるコード署名の設定を選択します。
7. [Save] を選択します。

IAM ポリシーの設定

コード署名 API オペレーションにアクセスする権限をユーザーに付与するには、1 つ以上のポリシーステートメントを、そのユーザーのポリシーにアタッチします。ユーザーポリシーの詳細については、「」を参照してくださいLambda での ID ベースの IAM ポリシーの使用

次に示すポリシーステートメントの例では、コード署名の設定を作成、更新、および取得するためのアクセス権限を付与しています。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

管理者は、CodeSigningConfigArn 条件キーを使用して、デベロッパーが関数を作成または更新する際に必要となる、コード署名の設定を指定できます。

次に示すポリシーステートメントの例では、関数を作成するための権限を付与しています。ポリシーステートメントには、許可されるコード署名設定を指定する lambda:CodeSigningConfigArn 条件を含めます。CodeSigningConfigArn パラメータが欠如しているか、条件の値と一致しない場合、Lambda はすべての CreateFunction API リクエストをブロックします。

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
          "lambda:CreateFunction",
        ],
      "Resource": "*",
      "Condition": {
          "StringEquals": {
              "lambda:CodeSigningConfigArn":
                  “arn:aws:lambda:us-west-2:123456789012:code-signing-
config:csc-0d4518bd353a0a7c6”
          }
      }
    }
  ]
}
```

Lambda API を使用したコード署名の設定

AWS CLIまたは AWS SDK でコード署名の設定を管理するには、次の API オペレーションを使用します。

- ListCodeSigningConfigs
- CreateCodeSigningConfig
- GetCodeSigningConfig
- UpdateCodeSigningConfig
- DeleteCodeSigningConfig

関数のためにコード署名の設定を管理するには、次の API オペレーションを使用します。

- CreateFunction
- GetFunctionCodeSigningConfig
- PutFunctionCodeSigningConfig
- DeleteFunctionCodeSigningConfig
- ListFunctionsByCodeSigningConfig

Lambda 関数でのタグの使用

AWS Lambda 関数にタグ付けすれば、属性ベースのアクセスコントロール (ABAC) をアクティブにし、所有者やプロジェクト、あるいは部門ごとの整理が行えます。タグは自由形式の key-value ペアで、AWS のサービス全体でサポートされており、ABAC や、リソースのフィルタリング、そして請求レポートへの詳細の追加のために使用できます。

タグは、バージョンやエイリアスではなく、関数レベルで適用されます。タグは、(バージョンの公開時に Lambda がスナップショットを作成する) バージョン固有の設定には含まれません。

セクション

- タグの操作に必要なアクセス許可
- Lambda コンソールでのタグの使用
- AWS CLIでのタグの使用
- タグの要件

タグの操作に必要なアクセス許可

関数を使用するユーザーの AWS Identity and Access Management (IAM) アイデンティティ (ユーザー、グループ、ロール) に、以下の中の適切なアクセス許可を付与します。

- lambda:ListTags — 関数にタグがある場合は、その関数`ListTags`で `GetFunction`または を呼び出す必要があるすべてのユーザーにこのアクセス許可を付与します。
- lambda:TagResource – `CreateFunction`または を呼び出す必要があるすべてのユーザーにこのアクセス許可を付与します`TagResource`。

詳細については、「Lambda での ID ベースの IAM ポリシーの使用」を参照してください。

Lambda コンソールでのタグの使用

Lambda コンソールを使用すると、タグ付けをしながら関数を作成したり、既存の関数にタグを追加したり、追加したタグで関数をフィルタリングしたりできます。

関数の作成時にタグを追加するには

1. Lambda コンソールの [関数ページ] を開きます。

2. [Create function] (関数の作成) を選択します。
3. [Author from scratch] (一から作成) または [Container image] (コンテナイメージ) をクリックします。
4. [基本的な情報] で、以下を実行します。

   a. [関数名] に関数名を入力します。関数名の長さは 64 文字に制限されています。

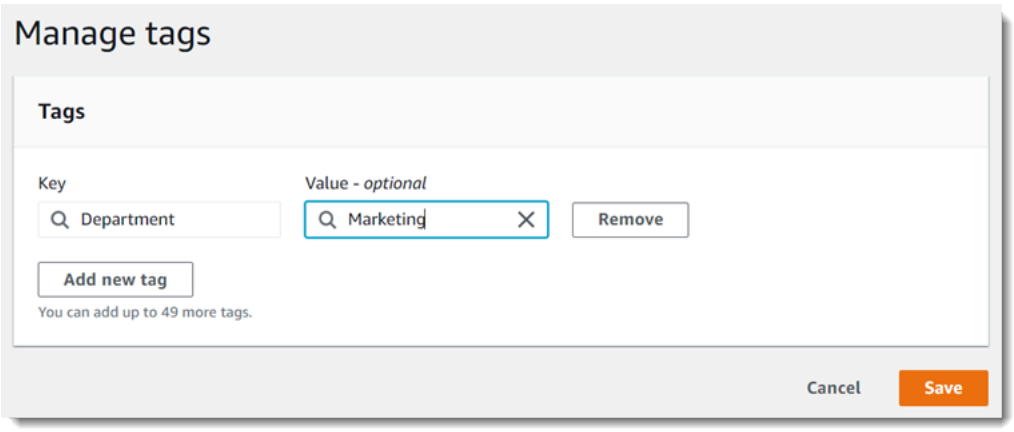
   b. [Runtime] (ランタイム) で、関数で使用する言語バージョンを選択します。

   c. (オプション) [Architecture] (アーキテクチャ) で、関数に使用する命令セットのアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用のデプロイパッケージを構築するときは、ここで選択した命令セットのアーキテクチャと互換性があることを確認してください。

5. [Advanced settings] (詳細設定) を展開した上で、[Enable tags] (タグの有効化) を選択します。
6. これを行うには、[Add new tag] (タグの追加) を選択してから、キーとオプションの値を入力します。さらにタグを追加するには、この手順を繰り返します。
7. [関数の作成] を選択します。

既存の関数にタグを追加するには

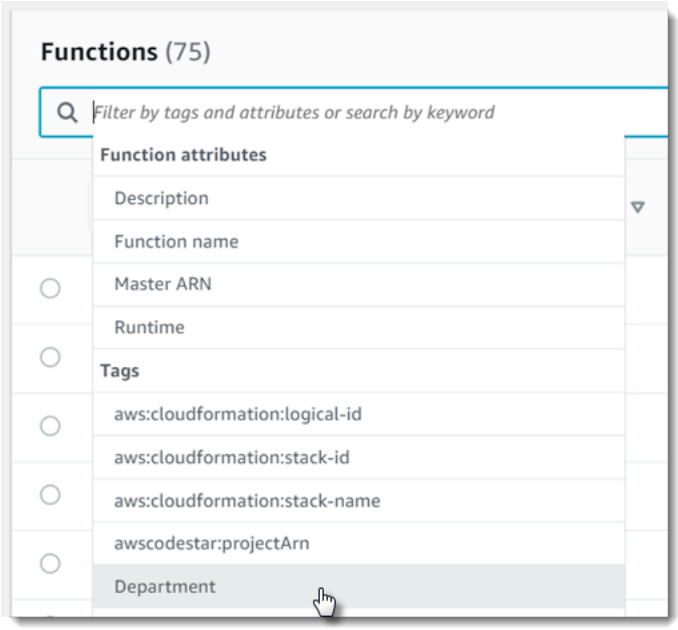
1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Configuration] (設定)、[Tags] (タグ) の順に選択します。
4. [タグ] の項目で、[タグの管理] を選択します。
5. これを行うには、[Add new tag] (タグの追加) を選択してから、キーとオプションの値を入力します。さらにタグを追加するには、この手順を繰り返します。



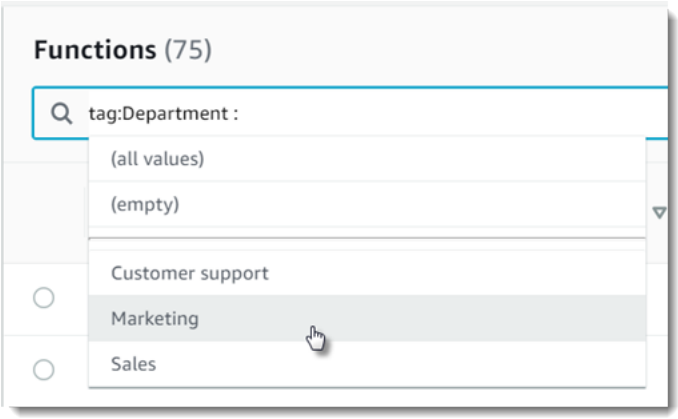
6. [Save] を選択します。

タグを使用して関数をフィルタするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 検索バーを選択すると、関数属性とタグキーのリストが表示されます。



3. タグキーを選択すると、現在の AWS リージョンで使用中の値のリストが表示されます。
4. 値を選択してその値を持つ関数を表示するか、[(all values)] を選択して、そのキーのタグを持つすべての関数を表示します。



検索バーは、タグキーの検索もサポートしています。タグキーのリストのみを表示するには `tag` と入力します。また、キーの名前を入力すると、それをリスト内で検索することもできます。

AWS CLIでのタグの使用

タグの追加と削除

-tags オプションとともに create-function コマンドを使用すると、タグを付けながら新しい Lambda 関数を作成することができます。

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

タグを既存の関数に追加するには、tag-resource コマンドを使用します。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:function:my-function \
--tags Department=Marketing,CostCenter=1234ABCD
```

タグを削除するには、untag-resource コマンドを使用します。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:function:my-
function \
--tag-keys Department
```

関数のタグの表示

特定の Lambda 関数に適用されているタグを表示する場合は、次のいずれかの AWS CLI コマンドを使用します。

- ListTags – この関数に関連付けられているタグのリストを表示するには、Lambda 関数 ARN (Amazon リソースネーム) を含めます。

  ```
  aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:function:my-
  function
  ```

- GetFunction – この関数に関連付けられているタグのリストを表示するには、Lambda 関数名を含めます。

  ```
  aws lambda get-function --function-name my-function
  ```

タグによる関数のフィルタリング

AWS Resource Groups Tagging API GetResources API オペレーションを使用して、リソースをタグでフィルタリングできます。この `GetResources` オペレーションは、それぞれにタグキーと最大 10 個のタグ値が含まれているフィルタを、最大 10 個まで受け取ります。特定のリソースタイプでフィルタリングするには、`GetResources` で `ResourceType` を指定します。

AWS Resource Groups の詳細については、「AWS Resource Groups Resource Groups とタグユーザーガイド」の「What are resource groups?」(Resource Groups とは?) を参照してください。

タグの要件

タグには、次の要件が適用されます。

- リソースあたりのタグの最大数: 50
- キーの最大長: 128 Unicode 文字 (UTF-8)
- 値の最大長: 256 Unicode文字 (UTF-8)
- タグのキーと値では、大文字と小文字が区別されます。
- タグの名前または値に `aws:` プレフィックスは使用しないでください。このプレフィックスは AWS 用に予約されています。このプレフィックスが含まれるタグの名前または値は編集または削除できません。このプレフィックスを持つタグは、リソースあたりのタグ数の制限時には計算されません。
- 複数のサービス間およびリソース間でタグ付けスキーマを使用する場合、他のサービスでは許可される文字が制限される可能性がある点にご注意ください。一般的に使用が許可される文字は、UTF-8 で表現できる文字、スペース、および数字と特殊文字 (+ - = . _ : / @) です。

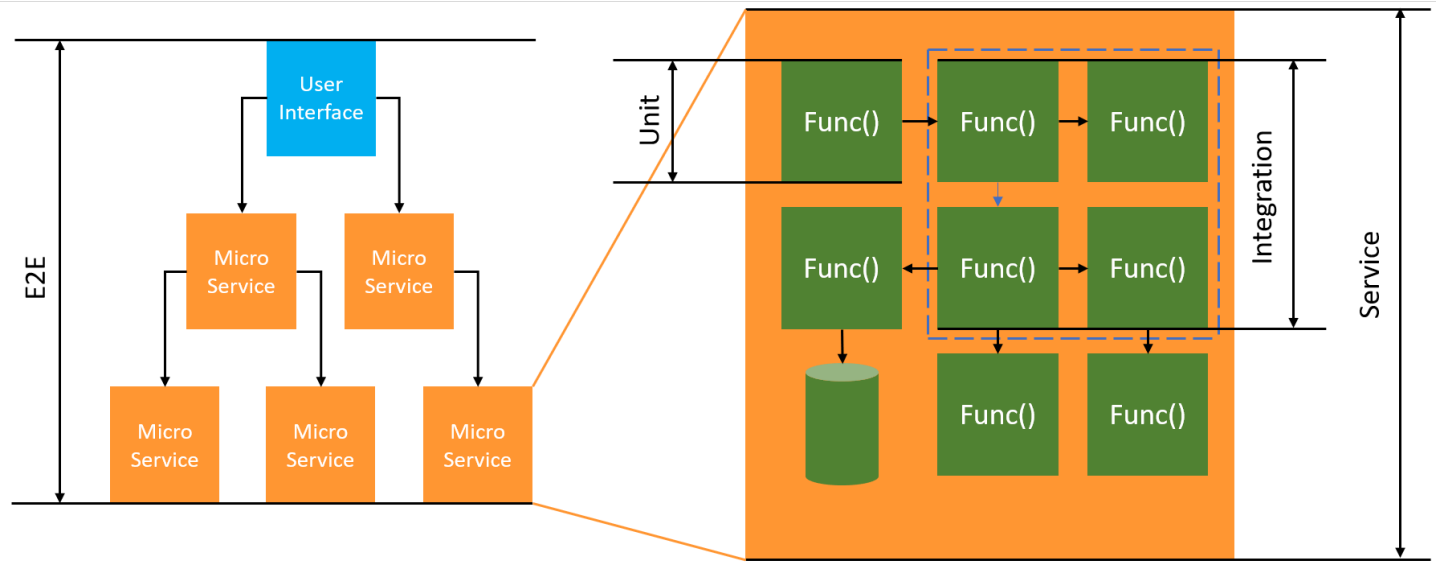
サーバーレス関数とアプリケーションをテストする方法

サーバーレス関数のテストでは、従来のテストタイプと手法を使用しますが、サーバーレスアプリケーション全体のテストも検討する必要があります。クラウドベースのテストでは、関数とサーバーレスアプリケーションの両方の品質を最も正確に測定できます。

サーバーレスアプリケーションアーキテクチャには、API 呼び出しを通じて重要なアプリケーション機能を提供するマネージドサービスが含まれます。このため、開発サイクルには、関数とサービスが相互に作用する際に機能を検証する自動テストを含める必要があります。

クラウドベースのテストを作成しない場合、ローカル環境とデプロイされた環境の違いにより問題が発生する可能性があります。継続的な統合プロセスでは、コードをQA、ステージング、本番稼働などの次のデプロイ環境に昇格する前に、クラウドにプロビジョニングされた一連のリソースに対してテストを実行する必要があります。

サーバーレスアプリケーションのテスト戦略に関する詳細については、このショートガイドを引き続きご覧ください。また、サーバーレステストサンプルリポジトリにアクセスして、選択した言語とランタイムに固有の実用的な例を調べることもできます。



サーバーレステストの場合も、ユニットテスト、統合テスト、エンドツーエンドテストを書くことになります。

- ユニットテスト - 分離されたコードブロックに対して実行されるテスト。例えば、特定の商品と配送先を指定して配送料を計算するビジネスロジックを検証する場合です。
- 統合テスト - 通常はクラウド環境で相互作用する 2 つ以上のコンポーネントまたはサービスを対象としたテスト。例えば、キューからのイベントを処理する関数を検証する場合です。

- エンドツーエンドテスト - アプリケーション全体の動作を検証するテスト。例えば、インフラストラクチャが正しく設定され、顧客の注文を記録するためにイベントがサービス間で想定どおりに流れることを確認する場合です。

ターゲットを絞ったビジネス成果

サーバーレスソリューションのテストでは、サービス間のイベント駆動型の相互作用を検証するテストの設定にさらに少し時間がかかる場合があります。このガイドを読む際には、以下のビジネス上の目的を念頭に置いてください。

- アプリケーションの品質を向上させる
- 機能の構築とバグ修正に必要な時間を短縮する

アプリケーションの品質は、さまざまなシナリオをテストして機能を検証するかどうかにかかっています。ビジネスシナリオを慎重に検討し、それらのテストをクラウドサービスに対して実行するように自動化することで、アプリケーションの品質が向上します。

ソフトウェアのバグや構成上の問題は、反復的な開発サイクルで発見されても、コストやスケジュールへの影響は最小限で済みます。開発中に問題が発見されないままになっていると、本番環境での発見と修正時により多くの人員による取り組みが必要になります。

サーバーレステスト戦略を綿密に計画すれば、Lambda 関数とアプリケーションがクラウド環境で期待どおりに動作することを検証することで、ソフトウェアの品質が向上し、反復時間が短縮されます。

テスト対象

マネージドサービスの動作、クラウド構成、セキュリティポリシー、コードとの統合をテストして、ソフトウェアの品質を向上させるテスト戦略を採用することをお勧めします。ブラックボックステストとも呼ばれる動作テストでは、内部構造のすべてを考慮せずに、システムが期待どおりに動作することを検証します。

- ユニットテストを実行して Lambda 関数内のビジネスロジックを確認します。
- 統合サービスが実際に呼び出されることと、入力パラメータが正しいことを検証します。
- イベントがワークフロー内で想定されるすべてのサービスをエンドツーエンドで通過することを確認します。

従来のサーバーベースのアーキテクチャでは、多くの場合、チームはテスト範囲にアプリケーションサーバー上で実行されるコードのみを含めるように定義していました。他のコンポーネント、サービス、依存関係はしばしば対象外とされ、テストの範囲から除外されていました。

通常、サーバーレスアプリケーションは、データベースからの製品取得、キューからの項目処理、ストレージ内の画像サイズ変更などを行う Lambda 関数のような小さな作業単位で構成されています。各コンポーネントは独自の環境で実行されます。こういった 1 つのアプリケーション内の小さなユニットの多くは、チームが担うことになるでしょう。

アプリケーション機能の中には、Amazon S3 などのマネージドサービスに完全に委任するものも、社内で開発したコードを使用せずに作成できるものもあります。これらのマネージドサービスのテストは不要ですが、これらのサービスとの統合についてはテストを実行する必要があります。

サーバーレスをテストする方法

ローカルにデプロイされたアプリケーションをテストする方法についてはご存知かと思います。つまり、デスクトップのオペレーティングシステムのみで動作するコードや、コンテナ内で動作するコードに対して実行するテストを書きます。例えば、リクエストを使用してローカルの Web サービスコンポーネントを呼び出し、そのレスポンスについてアサーションを実行できます。

サーバーレスソリューションは、関数コードと、キュー、データベース、イベントバス、メッセージングシステムなどのクラウドベースのマネージドサービスで構築されます。これらのコンポーネントはすべてイベント駆動型アーキテクチャを介して接続され、あるリソースから別のリソースにイベントと呼ばれるメッセージが流れます。このような相互作用は、同期的なアクション (Web サービスが直ちに結果を返すなど) の場合もあれば、後で完了する非同期的アクション (キューへのアイテムの配置やワークフローステップの開始など) の場合もあります。テスト戦略には両方のシナリオを含めて、サービス間の相互作用をテストする必要があります。非同期的な相互作用の場合、直ちに確認できないダウンストリームコンポーネントで発生する副作用の検出の必要が生じる可能性があります。

キュー、データベーステーブル、イベントバス、セキュリティポリシーなどを含むクラウド環境全体を複製することは現実的ではありません。ローカル環境とクラウドにデプロイされた環境の違いにより、必然的に問題が発生します。環境によって違いがあると、バグの再現と修正に要する時間が長くなります。

サーバーレスアプリケーションでは、アーキテクチャコンポーネントは通常すべてクラウドに存在するため、機能の開発やバグの修正には、クラウド内のコードとサービスに対してテストする必要があります。

テストのテクニック

実際には、テスト戦略にはソリューションの品質を向上させるためのさまざまな手法が含まれる可能性があります。簡単なインタラクティブテストを使用してコンソールで機能をデバッグする、自動ユニットテストを使用して分離されたビジネスロジックをチェックする、外部サービスへの呼び出しをモックで検証する、サービスを模倣するエミュレーターに対して時々テストを実行するなどの手法があります。

- クラウドでのテスト - インフラストラクチャとコードをデプロイして、実際のサービス、セキュリティポリシー、構成、インフラストラクチャ固有のパラメータでテストします。クラウドベースのテストは、コードの品質を最も正確に測定できます。

  コンソールで関数をデバッグすると、クラウドで簡単にテストできます。サンプルテストイベントのライブラリから選択することも、カスタムイベントを作成して関数を個別にテストすることもできます。コンソールからテストイベントをチームと共有することもできます。

  開発およびビルドライフサイクルのテストを自動化するには、コンソールの外部でテストする必要があります。自動化戦略とリソースについては、このガイドの言語固有のテストセクションを参照してください。
- モックを使ったテスト (フェイクとも呼ばれます) - モックとは、外部サービスをシミュレートして代用するコード内のオブジェクトです。モックには、サービスコールとパラメータを検証するための動作があらかじめ定義されています。フェイクとは、ショートカットを用いてパフォーマンスを単純化または改善する模擬実装のことです。例えば、フェイクのデータアクセスオブジェクトが、インメモリデータストアからデータを返す場合などです。モックは複雑な依存関係を模倣して単純化できますが、ネストされた依存関係を置き換えるためにモックが増えてしまうこともあります。
- エミュレータによるテスト - ローカル環境のクラウドサービスを模倣するアプリケーション (サードパーティ製の場合もあります) をセットアップできます。利点は高速であることですが、セットアップが必要であることと、本番環境のサービスとの同等性が課題となります。エミュレータを使いすぎないようにしてください。

クラウドでのテスト

クラウドでのテストは、ユニットテスト、統合テスト、エンドツーエンドテストなど、テストのあらゆる段階で役立ちます。クラウドベースのサービスと相互作用するクラウドベースのコードに対してテストを実行すると、コードの品質を最も正確に測定できます。

クラウドで Lambda 関数を実行する便利な方法は、AWS Management Console でテストイベントを行うことです。テストイベントとは、関数への JSON 入力のことです。関数が入力を必要としない

場合、イベントは空の JSON ドキュメント ({}) にすることができます。コンソールには、さまざまなサービス統合のサンプルイベントが用意されています。コンソールでイベントを作成したら、それをチームと共有して、テストを簡単かつ一貫性のあるものにすることもできます。

コンソールでサンプル関数をデバッグする方法を学びましょう。

> **Note**
>
> コンソールで関数を実行することはデバッグを簡単に実行する方法ですが、アプリケーションの品質と開発速度を向上させるには、テストサイクルを自動化することが不可欠です。

テスト自動化のサンプルは、サーバーレステストサンプルリポジトリで入手できます。以下のコマンドラインは、自動化された Python 統合テストの例を実行します。

```
python -m pytest -s tests/integration -v
```

テストはローカルで実行されますが、クラウドベースのリソースと相互に作用します。これらのリソースは、AWS Serverless Application Model および AWS SAM コマンドラインツールを使用してデプロイされています。テストコードは、まず API エンドポイント、関数 ARN、セキュリティロールを含む、デプロイされたスタック出力を取得します。次に、テストは API エンドポイントにリクエストを送信し、API エンドポイントは Amazon S3 バケットのリストを返します。このテストは、すべてクラウドベースのリソースに対して実行され、それらのリソースがデプロイされ、保護され、期待どおりに機能することを検証します。

```
========================== test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:

        HelloWorldApi
```

```
        = https://p7teqs3162.execute-api.us-west-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-west-2:1234567890:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::1234567890:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-west-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

クラウドネイティブアプリケーション開発の場合、クラウドでテストすることには次の利点があります。

- 利用可能なすべてのサービスをテストできます。
- 常に最新のサービス API と戻り値を使用します。
- クラウドテスト環境は、実稼働環境によく似ています。
- テストには、セキュリティポリシー、Service Quotas、構成、インフラストラクチャ固有のパラメータが含まれます。
- すべてのデベロッパーは、クラウドで 1 つ以上のテスト環境をすばやく作成できます。
- クラウドテストにより、コードが本番環境で正しく実行される精度が高まります。

クラウドでのテストにはいくつかの欠点があります。クラウドでのテストの最も明らかな欠点は、通常、クラウド環境へのデプロイはローカルデスクトップ環境へのデプロイよりも時間がかかることです。

幸い、AWSサーバーレスアプリケーションモデル (AWS SAM) Accelerate、AWSCloud Development Kit (AWS CDK) 監視モード、SST (サードパーティ) などのツールを使用すると、クラウドデプロイ

の繰り返しに伴うレイテンシーが軽減されます。これらのツールはインフラストラクチャとコードを監視し、自動的に増分アップデートをクラウド環境にデプロイできます。

> **Note**
>
> AWS Serverless Application Model、AWS CloudFormation、AWS Cloud Development Kit (AWS CDK) の詳細については、『サーバーレスデベロッパーガイド』の「インフラストラクチャをコードとして作成する方法」を参照してください。

ローカルテストとは異なり、クラウドでのテストには追加のリソースが必要で、サービスコストが発生する可能性があります。独立したテスト環境を作成すると、特にアカウントやインフラストラクチャを厳格に管理している組織では、DevOps チームの負担が増える場合があります。それでも、複雑なインフラストラクチャシナリオを扱う場合、複雑なローカル環境のセットアップと保守に要するデベロッパーの時間のコストは、Infrastructure as Code 自動化ツールで作成された使い捨てのテスト環境を使用する場合と同等 (またはそれ以上のコスト) になる可能性があります。

これらの考慮事項がありますが、サーバーレスソリューションの品質を保証するには、クラウドでのテストが依然として最善の方法です。

モックを使ったテスト

モックを使ったテストは、コード内に置換オブジェクトを作成してクラウドサービスの動作をシミュレートする手法です。

例えば、CreateObject メソッドが呼び出されるたびに特定の応答を返す Amazon S3 サービスのモックを使用するテストを作成できます。テストを実行すると、モックは Amazon S3 やその他のサービスエンドポイントを呼び出さずに、プログラムされた応答を返します。

多くの場合、モックオブジェクトは、開発の手間を減らすためにモックフレームワークによって生成されます。モックフレームワークの中には汎用的なものもあれば、AWS サービスやリソースを模倣するための Python ライブラリである Moto など、AWS SDK 専用に設計されたものもあります。

モックオブジェクトは、一般にデベロッパーがテストコードの一部として作成または設定しますが、エミュレータはエミュレート対象のシステムと同じ方法で機能を公開するスタンドアロンアプリケーションであるという点が異なることに注意してください。

モックを使用することのメリットは次のとおりです。

- モックは、API や Software as a Service (SaaS) プロバイダーなど、アプリケーションの制御が及ばないサードパーティのサービスを、それらのサービスに直接アクセスすることなくシミュレートできます。
- モックは障害状態をテストするのに便利です。特にサービス停止など、状態のシミュレートが困難である場合に役立ちます。
- モックを一度設定すれば、ローカルテストを迅速に行うことができます。
- モックは事実上あらゆる種類のオブジェクトの代替動作を提供できるため、モック戦略はエミュレータよりも幅広いサービスを対象とすることができます。
- 新しい機能や動作が利用可能になった場合、モックテストであればより迅速に対応できます。一般的なモックフレームワークを使用することで、更新された AWS SDK が利用可能になるとすぐに新機能をシミュレートできます。

モックテストには次のような欠点があります。

- モックは通常、特にレスポンスを適切に模倣するためにさまざまなサービスからの戻り値を特定しようとする際に、かなりの分量のセットアップや設定が必要になります。
- モックはデベロッパーが作成、設定し、管理しなければならないため、デベロッパーの責任が増大します。
- サービスの API と戻り値を理解するため、場合によってはクラウドにアクセスする必要があります。
- モックはメンテナンスが難しい場合があります。モックされたクラウド API シグネチャが変更されたり、戻り値のスキーマが変更されたりといった場合、モックを更新する必要があります。アプリケーションロジックを拡張して新しい API を呼び出す場合も、モックを更新する必要があります。
- モックを使用するテストは、デスクトップ環境では合格しても、クラウドでは失敗する可能性があります。結果が現在の API と一致しない可能性もあります。サービス構成とクォータはテストできません。
- モックフレームワークでは、AWS Identity and Access Management (IAM) ポリシーやクォータ制限のテストまたは検出には制限があります。認証が失敗した場合やクォータを超過した場合のシミュレーションにはモックの方が優れていますが、テストでは本番環境で実際にどのような結果になるかを判断することはできません。

エミュレーションによるテスト

エミュレータは、通常ローカルで実行される、AWS 本番環境サービスを模倣するアプリケーションです。

エミュレータには、クラウド版と同様の戻り値を提供する API があります。また、API 呼び出しによって開始される状態変化をシミュレートすることもできます。例えば、AWS SAM を使用して AWS SAM ローカルで関数を実行し、Lambda サービスをエミュレートすると、関数を迅速に呼び出すことができます。詳細については、AWS Serverless Application Model デベロッパーガイドのAWS SAM ローカルを参照してください。

エミュレータを使ったテストには、次のようなメリットがあります。

- エミュレータを使用すると、ローカルでの開発の反復やテストを迅速に行うことができます。
- エミュレータは、ローカル環境でのコード開発に慣れているデベロッパーにとって馴染みやすい環境を提供します。例えば、n 層アプリケーションの開発に慣れていれば、本番環境で実行されているものと同様のデータベースエンジンとウェブサーバーをローカルマシン上で実行して、高速かつ隔離されたテスト機能をローカルで提供できる可能性があります。
- エミュレーターはクラウドインフラストラクチャ (デベロッパーのクラウドアカウントなど) を変更する必要がないため、既存のテストパターンで簡単に実装できます。

エミュレータを使ったテストには、次のような欠点があります。

- エミュレータは、特に CI/CD パイプラインで使用する場合、セットアップや複製が難しい場合があります。このため、IT スタッフや独自のソフトウェアを管理するデベロッパーのワークロードが増大する可能性があります。
- エミュレートされた機能や API は、通常サービスの更新でラグが発生します。これにより、テストされたコードが実際の API と一致しないためにエラーが発生し、新機能の導入が妨げられる可能性があります。
- エミュレータには、サポート、更新、バグ修正、機能パリティの強化が必要です。これらはエミュレータの作成者の責任となりますが、作成者はサードパーティ企業の場合もあります。
- エミュレータを使用するテストはローカルでは成功するかもしれませんが、クラウドでは本番環境のセキュリティポリシー、サービス間の設定、Lambda クォータの超過などが原因で失敗する可能性があります。
- 多くの AWS サービスにはエミュレータがありません。エミュレーションに依存していると、アプリケーションの一部について十分なテストを行うオプションが得られない可能性があります。

ベストプラクティス

以下のセクションでは、サーバーレスアプリケーションのテストを成功させるための推奨事項について説明します。

サーバーレステストサンプルリポジトリには、テストとテスト自動化の実用的な例があります。

クラウドでのテストを優先する

クラウドでのテストは、最も信頼性が高く、正確かつ完全なテスト範囲を提供します。クラウドのコンテキストでテストを実行すると、ビジネスロジックだけではなく、セキュリティポリシー、サービス構成、クォータ、最新の API シグネチャと戻り値も包括的にテストされます。

テストしやすいようにコードを構造化する

Lambda 固有のコードを主要なビジネスロジックから分離することで、テストと Lambda 関数を簡略化できます。

Lambda 関数ハンドラーは、イベントデータを取り込み、ビジネスロジックメソッドにとって重要な詳細のみを渡すスリムなアダプターである必要があります。この方法では、Lambda 固有の詳細を気にすることなく、ビジネスロジックを中心に包括的なテストを行うことができます。AWS Lambda 関数では、テスト対象のコンポーネントを作成および初期化するために複雑な環境や大量の依存関係を設定する必要はありません。

一般には、受信したイベントとコンテキストオブジェクトからデータを抽出して検証し、その入力をビジネスロジックを実行するメソッドに送信するハンドラーを作成することになります。

開発のフィードバックループを高速化

開発フィードバックのループを高速化するためのツールやテクニックがあります。例えば、AWS SAM Accelerate と AWSCDK 監視モードは、いずれもクラウド環境の更新に要する時間を短縮します。

GitHub のサーバーレステストサンプルリポジトリにあるサンプルでは、これらのテクニックをいくつか取り上げています。

また、ソース管理にチェックインした後だけではなく、開発中のできるだけ早い段階でクラウドリソースを作成してテストすることをお勧めします。この方法により、ソリューションを開発する際の調査と実験を迅速に行うことができます。さらに、開発マシンからのデプロイを自動化することで、クラウド構成の問題をより迅速に発見し、更新やコードレビュープロセスに費やす無駄な労力を削減できます。

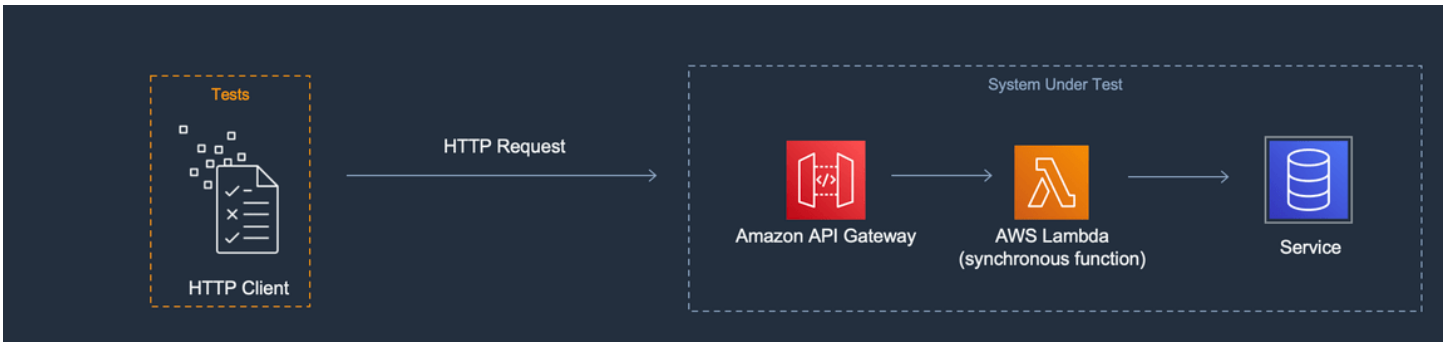
統合テストに焦点を当てる

Lambda でアプリケーションを構築する場合、コンポーネントをまとめてテストすることがベストプラクティスです。

2 つ以上のアーキテクチャコンポーネントに対して実行されるテストは統合テストと呼ばれます。統合テストの目的は、コードがコンポーネント間でどのように実行されるかだけではなく、コードをホストする環境がどのように動作するかを理解することです。エンドツーエンドテストは、アプリケーション全体の動作を検証する、特殊なタイプの統合テストです。

統合テストを作成するには、アプリケーションをクラウド環境にデプロイします。これはローカル環境から行うことも、CI/CD パイプラインを介して行うこともできます。次に、テストを作成して、テスト対象システム (SUT) を実行し、予想される動作を検証します。

例えば、テスト対象のシステムは、API Gateway、Lambda、DynamoDB を使用するアプリケーションの可能性があります。テストでは、API Gateway エンドポイントに対して合成の HTTP 呼び出しを行い、応答に期待されるペイロードが含まれていることを検証できます。このテストでは、AWS Lambda コードが正しいことと、各サービスがリクエストを処理するように正しく設定されていること (サービス間の IAM 権限を含む) を検証します。さらに、DynamoDB の最大レコードサイズなどの Service Quotas が正しく設定されていることを確認するために、さまざまなサイズのレコードを書き込むようにテストを設計することもできます。



分離テスト環境の構築

通常、クラウドでのテストには、テスト、データ、イベントが重複しないように、分離されたデベロッパー環境が必要です。

1 つの方法は、各デベロッパーに専用の AWS アカウントを提供することです。これにより、複数のデベロッパーが共有コードベースで作業する、リソースをデプロイする、API を呼び出すといった場合に発生する可能性のあるリソース命名との競合を回避できます。

自動テストプロセスでは、スタックごとに一意の名前を付けたリソースを作成する必要があります。例えば、AWS SAM CLI の sam deploy または sam sync コマンドがスタックを一意のプレフィックスで自動的に指定するように、スクリプトまたは TOML 設定ファイルを設定できます。

デベロッパーが AWS アカウントを共有するケースもあります。これは、運用やプロビジョニング、構成にコストがかかるリソースがスタック内に存在することに関係している可能性があります。例えば、データベースを共有すると、データのセットアップと適切なシードが容易になります。

デベロッパーがアカウントを共有する場合は、所有権を特定して重複を排除するための境界を設定する必要があります。これを行う 1 つの方法は、スタック名の前にデベロッパーのユーザー ID を付けることです。もう 1 つの一般的な方法は、コードブランチに基づいてスタックを設定することです。ブランチの境界があると、環境は分離されますが、デベロッパーはリレーショナルデータベースなどのリソースを共有できます。この方法は、デベロッパーが一度に複数のブランチで作業する場合のベストプラクティスです。

クラウドでのテストは、ユニットテスト、統合テスト、エンドツーエンドテストなど、テストのあらゆる段階で役立ちます。適切な分離を維持することは不可欠ですが、それでも QA 環境は本番環境にできるだけ近づける必要があります。このため、チームは QA 環境に変更管理プロセスを追加します。

本番稼働前環境と本番稼働環境では、一般にアカウントレベルで境界線が引かれます。これは、ノイズの多い近隣の問題からワークロードを隔離し、機密データを保護するための最小特権のセキュリティ制御を実装するためです。ワークロードにはクォータがあります。テストで本番環境 (ノイズの多い近隣) に割り当てられたノルマを消費したり、顧客データにアクセスしたりすることは望ましくありません。負荷テストは、本番環境のスタックから分離すべきもう 1 つのアクティビティです。

いずれの場合も、不要な支出を避けるために、環境にはアラートと制御を設定する必要があります。例えば、作成できるリソースのタイプ、階層、またはサイズを制限したり、推定コストが特定のしきい値を超えたときにメールアラートを送信したりすることができます。

分離されたビジネスロジックにはモックを使う

モックフレームワークは、高速なユニットテストを記述するための有用なツールです。特に、数学的計算や財務上の計算、シミュレーションなど、複雑な社内ビジネスロジックをテスト対象とする場合に役立ちます。テストケースや入力のバリエーションが多く、それらの入力によって他のクラウドサービスへの呼び出しのパターンや内容が変わらないユニットテストを目標とします。

モックを使ったユニットテストの対象となるコードは、クラウドでのテストでも対象とする必要があります。デベロッパーのラップトップ環境や構築用マシン環境は、クラウドの本番環境とは異なった形で構成される可能性があるため、これが推奨されます。例えば、Lambda 関数を特定の入力パラメータで実行すると、割り当てよりも多くのメモリや時間を使用する可能性があります。また、コードに同じ方法で設定されていない (またはまったく設定されていない) 環境変数が含まれており、その違いによってコードの動作が異なったり、失敗したりする可能性があります。

接続ポイントの数が増えるほど、必要なモックを実装する手間が増えるため、統合テストではモックの有用性は低下します。エンドツーエンドのテストではモックを使うべきではありません。これらのテストは一般にモックフレームワークでは簡単にシミュレートできない状態や複雑なロジックを扱うためです。

最後に、サービスコールの適切な実装を検証するためにモッククラウドサービスを使用することは避けてください。代わりに、クラウドでクラウドサービスを呼び出し、動作、設定、機能実装を検証してください。

エミュレータを使いすぎないようにしてください。

エミュレータは、開発チームの使うインターネット回線に制限がある、信頼性が低い、低速であるなど、一部のユースケースでは有用な場合があります。ただし、ほとんどの状況では、安易なエミュレータの使用を避けるべきです。

エミュレータの使用を回避することで、最新のサービス機能と最新の API を使って開発やイノベーションを行うことができます。機能の同等性を実現するために、ベンダーのリリースを待つ必要はありません。複数の開発システムや構築用マシンの購入と構成にかかる初期費用と継続的な費用を削減できます。さらに、多くのクラウドサービスではエミュレータが利用できないという問題を回避できます。エミュレーションに依存するテスト戦略では、サービスを使用することができない (その回避策が高額になる可能性もあります)、または十分テストされていないコードや構成が生成されるなどの問題が生じます。

テストにエミュレーションを使用する場合でも、構成を検証したり、エミュレートされた環境でのみシミュレートまたはモックが可能なクラウドサービスとの相互作用をテストしたりするには、やはりクラウドでテストする必要があります。

ローカルでのテストの課題

エミュレータとモック呼び出しを使用してローカルデスクトップでテストする場合、CI/CD パイプラインの環境間でコードが進行するにしたがって、テストの不整合が発生する可能性があります。デスクトップ上でアプリケーションのビジネスロジックを検証するユニットテストでは、クラウドサービスの重要な側面を正確にテストできない場合があります。

以下の例は、モックやエミュレータを使ってローカルでテストする場合に注意すべきケースを示しています。

例: Lambda 関数が S3 バケットを作成する

Lambda 関数のロジックが S3 バケットの作成に依存している場合は、Amazon S3 が呼び出され、バケットが正常に作成されたことを完全なテストで確認する必要があります。

- モックテストの設定では、成功レスポンスをモックし、失敗レスポンスを処理するテストケースを追加することもあります。
- エミュレーションテストのシナリオでは、CreateBucket API が呼び出される場合がありますが、ローカル呼び出しを行う ID は Lambda サービスから発信されたものではないことに注意する必要があります。発信者 ID は、クラウド内のようにセキュリティ上の役割を果たさないため、代わりにプレースホルダー認証が使用されます。この場合、クラウドで実行する場合と異なり、ロールやユーザー ID はそれほど厳密なものではありません。

モックとエミュレーションのセットアップでは、Lambda 関数が Amazon S3 を呼び出した場合の動作をテストしますが、これらのテストでは、Lambda 関数が設定どおりに Amazon S3 バケットを正常に作成できるかどうかは検証されません。機能に割り当てられたロールに、その機能が `s3:CreateBucket` アクションを実行できるようにするセキュリティポリシーがアタッチされていることを確認する必要があります。アタッチされていない場合、関数はクラウド環境へのデプロイ時に失敗する可能性があります。

例: Lambda 関数を使用して、Amazon SQS キューからのメッセージを処理する

Amazon SQS キューが Lambda 関数のソースである場合は、テストを完了して、メッセージがキューに入った際に Lambda 関数が正常に呼び出されることを確認する必要があります。

エミュレーションテストとモックテストは通常、Lambda 関数コードを直接実行し、関数ハンドラーの入力として JSON イベントペイロード (または逆シリアル化されたオブジェクト) を渡して Amazon SQS 統合をシミュレートするように設定されます。

Amazon SQS 統合をシミュレートするローカルテストでは、特定のペイロードで Amazon SQS から呼び出されたときに Lambda 関数がどのように実行されるかをテストしますが、クラウド環境にデプロイされたときに Amazon SQS が Lambda 関数を正常に呼び出すかどうかは検証しません。

Amazon SQS と Lambda で発生する可能性のある設定の問題として、次のような例があります。

- Amazon SQS の可視性タイムアウトが短すぎるため、意図された 1 回のみの呼び出しが複数回発生する。

- Lambda 関数の実行ロールが、キュー (`sqs:ReceiveMessage`、`sqs:DeleteMessage`、`sqs:GetQueueAttributes` を経由) からのメッセージ読み取りを許可しない。
- Lambda 関数に渡されるサンプルイベントが Amazon SQS メッセージサイズクォータを超えている。したがって、Amazon SQS ではそのサイズのメッセージは送信できず、テストが無効になる。

これらの例が示すように、ビジネスロジックは対象とするがクラウドサービス間の構成は対象としないテストでは、信頼性の低い結果が得られる可能性があります。

よくある質問

他のサービスを呼び出さずに計算を実行して結果を返す Lambda 関数があります。本当にクラウドでテストする必要がありますか？

はい。Lambda 関数には、テストの結果を変更する可能性のある設定パラメータがあります。すべての Lambda 関数コードはタイムアウトとメモリの設定に依存しているため、これらの設定が正しく設定されていないと関数が失敗する可能性があります。Lambda ポリシーでは、Amazon CloudWatch への標準出力ロギングも可能です。コードが CloudWatch を直接呼び出さない場合でも、ロギングを有効にするには権限が必要です。この必要な権限を正確にモックしたりエミュレートしたりすることはできません。

クラウドでのテストはユニットテストにどのように役立ちますか? クラウド内にあり他のリソースに接続しているとしたら、それは統合テストではありませんか?

ユニットテストとは、アーキテクチャのコンポーネントを個別に操作するテストと定義していますが、他のサービスを呼び出したり、何らかのネットワーク通信を使用するコンポーネントをテストに含めたりといった内容が除外されるわけではありません。

多くのサーバーレスアプリケーションには、クラウドでも個別にテストできるアーキテクチャコンポーネントがあります。1 つの例として、入力を受け付け、データを処理し、Amazon SQS キューにメッセージを送信する Lambda 関数があります。この関数のユニットテストは、入力値によってキューに入れられたメッセージに特定の値が存在するかどうかをテストするといったものになります。

例えば、アレンジ、アクト、アサートというパターンで書かれたテストを考えてみましょう。

- アレンジ: リソース (メッセージを受信するキュー、およびテスト対象の関数) を割り当てます。
- アクト: テスト対象の関数を呼び出します。

- アサート: 関数から送信されたメッセージを取得し、出力を検証します。

モックテストのアプローチでは、プロセス内のモックオブジェクトでキューをモックし、Lambda 関数コードを含むクラスまたはモジュールのプロセス内インスタンスを作成します。アサート段階では、キューに入れられたメッセージはモックされたオブジェクトから取得されます。

クラウドベースのアプローチでは、テスト用に Amazon SQS キューを作成し、分離された Amazon SQS キューを出力先として使用するように設定された環境変数を使用して Lambda 関数をデプロイします。Lambda 関数の実行後に、テストでメッセージを Amazon SQS キューから取得します。

クラウドベースのテストでは、同じコードを実行し、同じ動作をアサートして、アプリケーションの機能が正しいことを検証します。ただし、Lambda 関数の設定 (IAM ロール、IAM ポリシー、関数のタイムアウトとメモリの設定) を検証できるという利点もあります。

次のステップとリソース

実際のテスト例の詳細については、以下のリソースを参照してください。

実装例

GitHub の サーバーレステストサンプルリポジトリには、このガイドで説明されているパターンとベストプラクティスに従ったテストの具体例が含まれています。リポジトリには、前のセクションで説明したモック、エミュレーション、クラウドテストプロセスのサンプルコードとガイド付きウォークスルーが含まれています。このリポジトリを使用して、AWS の最新サーバーレステストガイダンスをご利用ください。

詳細情報

Serverless Land で、AWSサーバーレステクノロジーに関する最新のブログ、ビデオ、トレーニングにアクセスできます。

次の AWS ブログ記事を読むこともお勧めします。

- AWS SAM Accelerate でサーバーレス開発を高速化する (AWS ブログ記事)
- CDK 監視による開発速度の向上 (AWSブログ記事)
- AWS Step Functions Local とのサービス統合のモック (AWSブログ投稿)
- サーバーレスアプリケーションのテストを始める (AWSブログ投稿)

ツール

- AWS SAM – サーバーレスアプリケーションのテストとデバッグ
- AWS SAM – 自動化されたテストとの統合
- Lambda – Lambda コンソールでの Lambda 関数のテスト

Node.js による Lambda 関数の構築

AWS Lambda の Node.js を使用して JavaScript コードを実行できます。Lambda は Node.js のランタイムを指定して、イベントを処理するコードを実行します。コードは、管理している AWS Identity and Access Management (IAM) ロールの認証情報を使用して、AWS SDK for JavaScript を含む環境で実行されます。Node.js ランタイムに含まれている SDK バージョンの詳細については、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

Lambda は、以下の Node.js ランタイムをサポートしています。

Node.js

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日

> **Note**
>
> Node.js 18 以降のランタイムは AWS SDK for JavaScript v3 を使用しています。以前のランタイムから関数を移行するには、GitHub の「移行ワークショップ」の手順に従ってください。AWS SDK for JavaScript バージョン 3 の詳細については、「モジュラー AWS SDK for JavaScript が一般公開されました」のブログ記事を参照してください。

Node.js 関数を作成するには

1. Lambda コンソールを開きます。
2. [Create function] (関数の作成) をクリックします。
3. 以下の設定を行います。

- [Function name]: 関数名を入力します。
- [ランタイム]: [Node.js 20.x] を選択します。

4. [Create function] (関数の作成) をクリックします。
5. テストイベントを設定するには、[テスト] を選択します。
6. [イベント名] で、「**test**」と入力します。
7. [変更を保存] をクリックします。
8. [テスト] を選択して関数を呼び出します。

コンソールが、`index.js` または `index.mjs` という名前の単一のソースファイルを使用する Lambda 関数を作成します。このファイルを編集し、組み込みのコードエディタでファイルをさらに追加することができます。変更を保存するには [保存] を選択します。コードを実行するには、[Test] (テスト) を選択します。

> **Note**
>
> Lambda コンソールでは、AWS Cloud9 を使用して、ブラウザに統合開発環境を提供します。また、AWS Cloud9 を使用して、独自の環境で Lambda 関数を開発することもできます。詳細については、AWS Cloud9 ユーザーガイドの「AWS Toolkit を使用した AWS Lambda 関数の使用」を参照してください。

`index.js` または `index.mjs` ファイルは、イベントオブジェクトとコンテキストオブジェクトを取得する `handler` という名前の関数をエクスポートします。これは、関数が呼び出されるときに Lambda が呼び出すハンドラー関数です。Node.js 関数のランタイムは、Lambda から呼び出しイベントを取得し、ハンドラに渡します。関数設定で、ハンドラ値は `index.handler` です。

関数コードを保存すると、Lambda コンソールは .zip ファイルアーカイブのデプロイパッケージを作成します。コンソール外で (SDE を使用して) 関数コードを開発するときは、デプロイパッケージを作成して、Lambda 関数にコードをアップロードします。

> **Note**
>
> ローカル環境でアプリケーション開発を開始するには、このガイドの GitHub リポジトリで利用可能なサンプルアプリケーションの 1 つをデプロイします。

Node.js のサンプル Lambda アプリケーション

- blank-nodejs – ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Node.js 関数。
- nodejs-apig – API Gateway からのイベントを処理し、HTTP レスポンスを返す公開 API エンドポイントを持つ関数。
- efs-nodejs – Amazon VPC で Amazon EFS ファイルシステムを使用する関数。このサンプルには、Lambda で使用するように設定された VPC、ファイルシステム、マウントターゲット、アクセスポイントが含まれます。

関数のランタイムによって、呼び出しイベントに加えて、コンテキストオブジェクトがハンドラに渡されます。コンテキストオブジェクトには、呼び出し、関数、および実行環境に関する追加情報が含まれます。詳細情報は、環境変数から入手できます。

Lambda 関数には CloudWatch Logs ロググループが付属しています。関数のランタイムは、各呼び出しに関する詳細を CloudWatch Logs に送信します。これは呼び出し時に、任意の関数が出力するログを中継します。関数がエラーを返す場合、Lambda はエラー形式を整え、それを呼び出し元に返します。

トピック

- Node.js の初期化
- ランタイムに含まれる SDK バージョン
- TCP 接続にキープアライブを使用
- CA 証明書のロード
- Node.js の Lambda 関数ハンドラーの定義
- .zip ファイルアーカイブで Node.js Lambda 関数をデプロイする
- Node.js Lambda 関数をコンテナイメージとともにデプロイする
- Node.js の AWS Lambda context オブジェクト
- Node.js の AWS Lambda 関数ログ作成
- AWS Lambda での Node.js コードの作成

Node.js の初期化

Node.js には独特なイベントループモデルがあるため、その初期化動作が他のランタイムとは異なります。具体的に言うと、Node.js は非同期操作をサポートするノンブロッキング I/O モデルを使用します。このモデルにより、Node.js はほとんどのワークロードに対して効率的に動作できます。例えば、Node.js 関数がネットワークコールを実行する場合、そのリクエストを非同期操作として指定し、コールバックキューに置くことができます。この関数は、ネットワークコールが返されるのを待つことで、ブロックされることなくメインコールスタック内の他の操作を引き続き処理することができます。ネットワークコールが完了すると、そのコールバックが実行され、コールバックキューから削除されます。

一部の初期化タスクは非同期的に実行される場合があります。これらの非同期タスクは、呼び出し前に実行が完了することが保証されていません。例えば、AWS パラメータストアからパラメータを取得するためのネットワークコールを実行するコードは、Lambda がハンドラー関数を実行する時までに完了しない場合があります。その結果、呼び出し中は変数が null になることがあります。これを回避するには、関数のコアビジネスロジックの残りの部分を続行する前に、変数とその他の非同期コードが完全に初期化されていることを確認してください。

別の手段として、関数コードを ES モジュールとして指定することもできます。これは、関数ハンドラーの範囲外で、ファイルのトップレベルで `await` を使用することを可能にします。`Promise` ごとに `await` すると、ハンドラーの呼び出し前に非同期初期化コードが完了されるので、コールドスタートレイテンシーの削減における[プロビジョニングされた同時実行]の効果を最大限に引き出すことができます。詳細情報と例については、「[AWS Lambda で Node.js ES モジュールと Top-Level Await を使用する]」を参照してください。

ES モジュールとしての関数ハンドラーの指定

デフォルトで、Lambda は `.js` サフィックスが付いたファイルを CommonJS モジュールとして扱います。オプションで、コードを ES モジュールとして指定できます。これは、関数の `package.json` ファイルで `type` を `module` として指定する方法と、`.mjs` のファイル名拡張子を使用する方法の 2 つの方法で実行できます。最初のアプローチでは、関数コードがすべての `.js` ファイルを ES モジュールとして扱い、2 番目のシナリオでは、`.mjs` で指定したファイルのみが ES モジュールになります。`.mjs` ファイルは常に ES モジュールであり、`.cjs` ファイルは常に CommonJS モジュールであることから、ES モジュールと CommonJS モジュールをそれぞれ `.mjs` および `.cjs` として命名することで、それらを混在させることができます。

Lambda は、ES モジュールのロード時に NODE_PATH 環境変数内のフォルダを検索します。ES モジュールの import ステートメントを使用して、ランタイムに含まれている AWS SDK をロードできます。レイヤーから ES モジュールをロードすることも可能です。

ランタイムに含まれる SDK バージョン

Node.js ランタイムに含まれる AWS SDK のバージョンは、ランタイムバージョンと AWS リージョン によって異なります。使用しているランタイムに含まれている SDK のバージョンを確認するには、次のコードを使用して Lambda 関数を作成します。

> **Note**
>
> Node.js バージョン 18 以降に対応する以下のコード例は、CommonJS 形式を使用しています。Lambda コンソールで関数を作成する場合は、コードを含むファイル名を index.js に変更してください。

Example Node.js 18 およびそれ以降

```
const { version } = require("@aws-sdk/client-s3/package.json");

exports.handler = async () => ({ version });
```

これにより、次の形式で応答が返されます。

```
{
  "version": "3.462.0"
}
```

TCP 接続にキープアライブを使用

デフォルトの Node.js HTTP/HTTPS エージェントは新しいリクエストがあるたびに新しい TCP 接続を作成します。新しい接続を確立するコストを回避するために、AWS SDK for JavaScript を使用して、関数が行う接続を再利用するために keepAlive: true を使用することができます。キープアライブは、SDK を使用して複数の API 呼び出しを行う Lambda 関数のリクエスト時間を短縮することができます。

nodejs18.x 以降の Lambda ランタイムに含まれている AWS SDK for JavaScript 3.x では、キープアライブがデフォルトで有効になっています。キープアライブを無効にするには、AWS SDK

for JavaScript 3.x 開発者ガイドの「Node.js でのキープアライブによる接続の再利用」を参照してください。キープアライブの使用の詳細については、AWS 開発者ツールブログの「HTTP キープアライブはモジュラー AWS SDK for JavaScript でデフォルトでオンになっています」を参照してください。

CA 証明書のロード

Node.js 18 までのバージョンの Node.js ランタイムでは、他の AWS サービスと相互作用する関数を簡単に作成できるように、Lambda が Amazon 固有の CA (認証局) 証明書を自動的にロードします。例えば、Lambda には、Amazon RDS データベースにインストールされているサーバー ID 証明書を検証するために必要な Amazon RDS 証明書が含まれています。この動作は、コールドスタート時のパフォーマンスに影響を与える可能性があります。

Node.js 20 以降では、Lambda がデフォルトで追加の CA 証明書をロードすることはありません。Node.js 20 ランタイムには、`/var/runtime/ca-cert.pem` にあるすべての Amazon CA 証明書を含む証明書ファイルが含まれています。Node.js 18 以前のランタイムと同じ動作を復元するには、`NODE_EXTRA_CA_CERTS` 環境変数を `/var/runtime/ca-cert.pem` に設定します。

最適なパフォーマンスを得るには、必要な証明書のみをデプロイパッケージにバンドルし、`NODE_EXTRA_CA_CERTS` 環境変数を使用してロードすることをお勧めします。証明書ファイルには、1 つ以上の信頼できるルート CA 証明書または中間 CA 証明書が PEM 形式で含まれている必要があります。例えば RDS の場合は、必要な証明書をコードと共に `certificates/rds.pem` として含めてください。次に、`NODE_EXTRA_CA_CERTS` を `/var/task/certificates/rds.pem` に設定して証明書をロードします。

Node.js の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

次の例の関数では、イベントオブジェクトの内容をログに記録して、その口グの場所を返します。

> **Note**
>
> このページは、CommonJS と ES モジュールハンドラーの両方の例を示します。これらの 2 つのハンドラータイプの違いについては、「ES モジュールとしての関数ハンドラーの指定」を参照してください。

ES module handler

Example

```
export const handler = async (event, context) => {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

CommonJS module handler

Example

```
exports.handler = async function (event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2));
  return context.logStreamName;
};
```

関数を設定すると、ハンドラー設定の値はファイル名とエクスポートしたハンドラーメソッドの名前をドットで区切ったものになります。コンソールのデフォルトと、このガイドの例では、`index.handler` です。これは、`handler` ファイルからエクスポートされた `index.js` メソッドを示します。

ランタイムでは、ハンドラーメソッドに引数を渡します。最初の引数は、呼び出し元からの情報を含む `event` オブジェクトです。呼び出し元は、Invoke を呼び出してこの情報を JSON 形式の文字列

として渡し、ランタイムはそれをオブジェクトに変換します。AWS のサービスで関数を呼び出す場合、そのイベント構造はサービスによって異なります。

2 番目の引数は、コンテキストオブジェクトです。この引数には、呼び出し、関数、および実行環境に関する情報が含まれます。前述の例では、関数は、コンテキストオブジェクトからログストリームの名前を取得し、それを呼び出し元に返します。

コールバック引数を使用することもできます。これは、非同期ではないハンドラーで呼び出してレスポンスを送信できる関数です。コールバックの代わりに async/await を使用することをお勧めします。async/await により、読みやすさ、エラー処理、および効率が向上します。async/await とコールバックの違いの詳細については、「コールバックの使用」を参照してください。

命名

関数を設定すると、ハンドラー設定の値はファイル名とエクスポートしたハンドラーメソッドの名前をドットで区切ったものになります。コンソールで作成された関数のデフォルトと、このガイドの例では、`index.handler` です。これは、`index.js` または `index.mjs` ファイルからエクスポートされた `handler` メソッドを示します。

異なるファイル名または関数ハンドラー名を使用してコンソールで関数を作成する場合は、デフォルトのハンドラー名を編集する必要があります。

関数ハンドラー名を変更するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、関数を選択します。
2. [コード] タブを選択します。
3. [ランタイム設定] ペインまでスクロールして、[編集] を選択します。
4. [ハンドラー] には、関数ハンドラーの新しい名前を入力します。
5. [Save] を選択します。

async/await の使用

コードが非同期タスクを実行する場合は、async/await パターンを使用して、ハンドラーが実行を確実に終了するようにします。async/await は、Node.js で非同期コードを記述するための簡潔で読みやすい方法であり、ネストされたコールバックや連鎖する promise を必要としません。async/await を使用すると、非同期かつノンブロッキングでありながら、同期コードのように読み取るコードを記述できます。

async キーワードは関数を非同期としてマークし、await キーワードは Promise が解決されるまで関数の実行を一時停止します。

> **Note**
>
> 非同期イベントが完了するまでお待ちください。非同期イベントが完了する前に関数が戻る場合、関数が失敗したり、アプリケーションで予期しない動作が発生したりする可能性があります。これは、forEach ループに非同期イベントが含まれている場合に発生します。forEach ループは同期呼び出しを想定しています。詳細については、Mozilla ドキュメントの「Array.prototype.forEach()」を参照してください。

ES module handler

Example – async/await を使用した HTTP リクエスト

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        // fetch is available in Node.js 18 and later runtimes
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module handler

Example – async/await を使用した HTTP リクエスト

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
```

```
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

次の例では、async/await を使用して Amazon Simple Storage Service バケットを一覧表示します。

> **Note**
>
> この例を使用する前に、関数の実行ロールに Amazon S3 の読み取り許可があることを確認してください。

ES module handler

Example – async/await を使用した AWS SDK v3

この例では、nodejs18.x 以降のランタイムで利用できる「AWS SDK for JavaScript v3」を使用します。

```
import {S3Client, ListBucketsCommand} from '@aws-sdk/client-s3';
const s3 = new S3Client({region: 'us-east-1'});

export const handler = async(event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;

};
```

CommonJS module handler

Example – async/await を使用した AWS SDK v3

この例では、nodejs18.x 以降のランタイムで利用できる「AWS SDK for JavaScript v3」を使用します。

```
const { S3Client, ListBucketsCommand } = require('@aws-sdk/client-s3');
```

```
const s3 = new S3Client({ region: 'us-east-1' });

exports.handler = async (event) => {
  const data = await s3.send(new ListBucketsCommand({}));
  return data.Buckets;
};
```

コールバックの使用

コールバックを使用する代わりに、async/await を使用して関数ハンドラーを宣言することをお勧めします。いくつかの理由により、async/await の方が適しています。

- 読みやすさ: async/await コードは、コールバックコードよりも読みやすく理解しやすいです。コールバックコードは、すぐに理解するのが難しくなり、コールバック地獄に陥る可能性があります。
- デバッグとエラー処理: コールバックベースのコードのデバッグは難しい場合があります。コールスタックを追跡するのが難しくなり、エラーが簡単に隠れてしまう可能性があります。async/await では、try/catch ブロックを使用してエラーを処理できます。
- 効率: コールバックでは、多くの場合、コードのさまざまな部分を切り替える必要があります。async/await を使用すると、コンテキストスイッチの回数を減らすことができるため、コードがより効率的になります。

ハンドラーでコールバックを使用すると、関数は、イベントループが空になるか、関数がタイムアウトするまで実行を続けます。レスポンスは、すべてのイベントループタスクが完了するまで、呼び出し元に送信されません。関数がタイムアウトした場合は、エラーが返ります。すぐにレスポンスが返るようにランタイムを設定するには、context.callbackWaitsForEmptyEventLoop を false に設定します。

コールバック関数は、`Error` とレスポンスの 2 つの引数を取ります。応答オブジェクトは、`JSON.stringify` と互換性がある必要があります。

次の例の関数では、URL をチェックし、ステータスコードを呼び出し元に返します。

ES module handler

Example – callback を使用した HTTP リクエスト

```
import https from "https";
```

```
let url = "https://aws.amazon.com/";

export function handler(event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
}
```

CommonJS module handler

Example – callback を使用した HTTP リクエスト

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = function (event, context, callback) {
  https.get(url, (res) => {
      callback(null, res.statusCode);
    }).on("error", (e) => {
      callback(Error(e));
    });
};
```

次の例では、Amazon S3 のレスポンスは、有効になるとすぐに呼び出し元に返ります。イベントループで実行されているタイムアウトは停止し、次に関数が呼び出されたときに実行を継続します。

> **Note**
>
> この例を使用する前に、関数の実行ロールに Amazon S3 の読み取り許可があることを確認してください。

ES module handler

Example – callbackWaitsForEmptyEventLoop を使用した AWS SDK v3

この例では、`nodejs18.x` 以降のランタイムで利用できる「AWS SDK for JavaScript v3」を使用します。

```
import AWS from "@aws-sdk/client-s3";
const s3 = new AWS.S3({});

export const handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

CommonJS module handler

Example – callbackWaitsForEmptyEventLoop を使用した AWS SDK v3

この例では、nodejs18.x 以降のランタイムで利用できる「AWS SDK for JavaScript v3」を使用します。

```
const AWS = require("@aws-sdk/client-s3");
const s3 = new AWS.S3({});

exports.handler = function (event, context, callback) {
  context.callbackWaitsForEmptyEventLoop = false;
  s3.listBuckets({}, callback);
  setTimeout(function () {
    console.log("Timeout complete.");
  }, 5000);
};
```

.zip ファイルアーカイブで Node.js Lambda 関数をデプロイする

AWS Lambda 関数のコードは、関数のハンドラーコードと、そのコードが依存するその他のパッケージやモジュールを含む .jp または .mjs ファイルで構成されています。この関数コードを Lambda にデプロイするには、デプロイパッケージを使用します。このパッケージは、.zip ファイルアーカイブでもコンテナイメージでもかまいません。Node.js でコンテナイメージを使用する方法の詳細については、「コンテナイメージで Node.js Lambda 関数をデプロイする」を参照してください。

.zip ファイルのデプロイパッケージを .zip ファイルアーカイブとして作成するには、コマンドラインツール用の組み込み .zip ファイルアーカイブユーティリティ、または他の .zip ファイルユーティリティ (7zip など) を使用します。次のセクションに示す例では、Linux または macOS 環境でコマンドライン `zip` ツールを使用していることを前提としています。Windows で同じコマンドを使用するには、Windows Subsystem for Linux をインストールして、Windows 統合バージョンの Ubuntu と Bash を取得します

Lambda は POSIX ファイルアクセス許可を使用するため、.zip ファイルアーカイブを作成する前に、デプロイパッケージフォルダのアクセス許可を設定することが必要になる場合があります。

トピック

- Node.js でのランタイム依存関係
- 依存関係のない .zip デプロイパッケージを作成する
- 依存関係を含めて .zip デプロイパッケージを作成する
- 依存関係の Node.js レイヤーを作成する
- 依存関係検索パスおよびランタイムを含むライブラリ
- .zip ファイルを使用した Node.js Lambda 関数の作成と更新

Node.js でのランタイム依存関係

Node.js ランタイムを使用する Lambda 関数の場合、依存関係はどの Node.js モジュールでもかまいません。Node.js ランタイムには、多くの共通ライブラリに加えて、あるバージョンの AWS SDK for JavaScript が含まれています。`nodejs16.x` Lambda ランタイムには、SDK バージョン 2.x が含まれています。ランタイムバージョン `nodejs18.x` 以降には、SDK バージョン 3 が含まれています。ランタイムバージョン `nodejs18.x` 以降の SDK バージョン 2 を使用するには、SDK を .zip ファイルのデプロイパッケージに追加します。選択したランタイムに、使用している SDK のバージョンが含まれている場合は、.zip ファイルに SDK ライブラリを含める必要はありません。使用しているラ

ンタイムに含まれている SDK のバージョンを確認するには、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

Lambda は Node.js ランタイムの SDK ライブラリを定期的に更新して、最新機能とセキュリティアップグレードを適用します。Lambda は、ランタイムに含まれる他のライブラリにもセキュリティパッチとアップデートを適用します。パッケージ内の依存関係を完全に制御するには、ランタイムに含まれる依存関係の任意のバージョンをデプロイパッケージに追加できます。例えば、特定のバージョンの JavaScript 用 SDK を使用する場合は、そのバージョンを依存関係として .zip ファイルに含めることができます。ランタイムに含まれる依存関係を .zip ファイルに追加する方法の詳細については、依存関係検索パスおよびランタイムを含むライブラリ を参照してください。

AWS 責任分担モデルでは、関数のデプロイパッケージに含まれる依存関係を管理する責任があります。これには、更新とセキュリティパッチの適用が含まれます。関数のデプロイパッケージ内の依存関係を更新するには、まず新しい .zip ファイルを作成し、そのファイルを Lambda にアップロードします。詳細については、「依存関係を含めて .zip デプロイパッケージを作成する」と「.zip ファイルを使用した Node.js Lambda 関数の作成と更新」を参照してください。

依存関係のない .zip デプロイパッケージを作成する

関数コードに、Lambda ランタイムに含まれるライブラリ以外の依存関係がない場合、.zip ファイルには関数のハンドラーコードを含む `index.js` または `index.mjs` ファイルのみが含まれます。任意の zip ユーティリティを使用して、`index.js` または `index.mjs` ファイルをルートに置く .zip ファイルを作成します。関数のハンドラーコードを含むファイルが .zip ファイルのルートにない場合、Lambda はコードを実行できません。

.zip ファイルをデプロイして新しい Lambda 関数を作成する方法の詳細、既存の Lambda 関数を更新する方法の詳細については、「.zip ファイルを使用した Node.js Lambda 関数の作成と更新」を参照してください。

依存関係を含めて .zip デプロイパッケージを作成する

関数コードが Lambda Node.js ランタイムに含まれていないパッケージやモジュールに依存している場合、これらの依存関係を関数コードとともに .zip ファイルに追加するか、Lambda レイヤーを使用できます。このセクションでは、依存関係を .zip デプロイパッケージに含める方法について説明します。依存関係をレイヤーに含める方法については、「the section called “依存関係の Node.js レイヤーを作成する”」を参照してください。

次の CLI コマンドの例では、関数のハンドラーコードとその依存関係を含む `index.js` または `index.mjs` ファイルを格納する `my_deployment_package.zip` という名前の .zip ファイル

を作成します。この例では、npm パッケージマネージャーを使用して依存関係をインストールします。

デプロイパッケージを作成するには

1. index.js または index.mjs ソースコードファイルを含むプロジェクトディレクトリに移動します。この例では、ディレクトリ名は my_function です。

```
cd my_function
```

2. npm install コマンドを使用して、node_modules ディレクトリに関数に必要なライブラリをインストールします。この例では、AWS X-Ray SDK for Node.js をインストールします。

```
npm install aws-xray-sdk
```

次のようなフォルダ構造が作成されます。

```
~/my_function
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

自分で作成したカスタムモジュールをデプロイパッケージに追加することもできます。モジュールの名前を使用して node_modules にディレクトリを作成し、そこにカスタムで作成したパッケージを保存します。

3. プロジェクトフォルダの内容を含む .zip ファイルをルートに作成します。r (再帰的) オプションを使用して、zip がサブフォルダを確実に圧縮するようにします。

```
zip -r my_deployment_package.zip .
```

依存関係の Node.js レイヤーを作成する

このセクションでは、依存関係をレイヤーに含める方法について説明します。依存関係をデプロイパッケージに含める方法については、「the section called “依存関係を含めて .zip デプロイパッケージを作成する”」を参照してください。

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の /opt ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には /opt ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

- nodejs/node_modules
- nodejs/node16/node_modules (NODE_PATH)
- nodejs/node18/node_modules (NODE_PATH)
- nodejs/node20/node_modules (NODE_PATH)

例えば、レイヤーの.zip ファイルの構造は次のようになります。

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

さらに、Lambda は /opt/lib ディレクトリ内のライブラリ、および /opt/bin ディレクトリ内のバイナリを自動的に検出します。Lambda がレイヤーのコンテンツを正しく検出できるように、次の構造でレイヤーを作成することもできます。

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

レイヤーをパッケージ化したら、「the section called “レイヤーの作成と削除”」および「the section called “レイヤーの追加”」を参照してレイヤーの設定を完了してください。

依存関係検索パスおよびランタイムを含むライブラリ

Node.js ランタイムには、多くの共通ライブラリに加えて、あるバージョンの AWS SDK for JavaScript が含まれています。ランタイムに含まれるライブラリの別のバージョンを使用する場合は、それを関数にバンドルするか、デプロイパッケージに依存関係として追加することで対応できます。例えば、別のバージョンの SDK を .zip デプロイパッケージに追加して使用できます。関数の Lambda レイヤーに含めることもできます。

コードで `import` または `require` ステートメントを使用すると、Node.js ランタイムはモジュールが見つかるまで `NODE_PATH` パス内のディレクトリを検索します。デフォルトでは、ランタイムが最初に検索する場所は、.zip デプロイパッケージを解凍してマウントするディレクトリ (`/var/task`) です。ランタイムに含まれるライブラリのバージョンをデプロイパッケージに含める場合、そのバージョンが、ランタイムに含まれるバージョンよりも優先されます。デプロイパッケージ内の依存関係も、レイヤー内の依存関係よりも優先されます。

レイヤーに依存関係を追加すると、Lambda はこれを `/opt/nodejs/nodexx/node_modules` に抽出します。ここで `nodexx` は使用しているランタイムのバージョンを表します。検索パスでは、このディレクトリはランタイムに含まれるライブラリを含むディレクトリ (`/var/lang/lib/node_modules`) よりも優先されます。このため、関数レイヤー内のライブラリは、ランタイムに含まれるバージョンよりも優先されます。

次のコード行を追加すると、Lambda 関数の完全な検索パスを確認できます。

```
console.log(process.env.NODE_PATH)
```

.zip パッケージ内の個別のフォルダに依存関係を追加することもできます。例えば、カスタムモジュールを .zip パッケージ内の `common` というフォルダに追加できます。.zip パッケージを解凍してマウントすると、このフォルダは `/var/task` ディレクトリ内に配置されます。コード内の.zip デプロイパッケージにあるフォルダの依存関係を使用するには、CJS または ESM のどちらのモジュール解決を使用しているかに応じて、`import { } from` か `const { } = require()` ステートメントを使用します。例:

```
import { myModule } from './common'
```

コードを `esbuild`、`rollup` または同様のものでバンドルすると、関数で使用される依存関係が 1 つ以上のファイルにバンドルされます。可能な限り、この方法を使用して依存関係を提供することをお勧めします。デプロイパッケージに依存関係を追加する場合と比較して、コードをバンドルすると I/O 操作が減るため、パフォーマンスが向上します。

.zip ファイルを使用した Node.js Lambda 関数の作成と更新

.zip デプロイパッケージを作成すると、このパッケージを使用して新しい Lambda 関数を作成するか、既存の関数を更新できます。.zip パッケージをデプロイするには、Lambda コンソール、AWS Command Line Interface、Lambda API を使用します。AWS Serverless Application Model (AWS SAM) および AWS CloudFormation を使用して、Lambda 関数を作成および更新することもできます。

Lambda の .zip デプロイパッケージの最大サイズは 250 MB (解凍) です。この制限は、Lambda レイヤーを含む、更新するすべてのファイルの合計サイズに適用されることに注意してください。

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

コンソールを使用して .zip ファイルの関数を作成、更新する

新しい関数を作成するには、まずコンソールで関数を作成し、次に .zip アーカイブをアップロードする必要があります。既存の関数を更新するには、その関数のページを開き、同じ手順に従って更新した .zip ファイルを追加します。

.zip ファイルが 50 MB 未満の場合は、ローカルマシンから直接ファイルをアップロードして関数を作成または更新できます。50 MB を超える .zip ファイルの場合は、まず Amazon S3 バケットにパッケージをアップロードする必要があります。AWS Management Console を使用して Amazon S3 バケットにファイルをアップロードする手順については、「Amazon S3 の開始方法」を参照してください。AWS CLI を使用してファイルをアップロードするには、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [一から作成] を選択します。
3. [基本的な情報] で、以下を行います。
   a. [関数名] に、関数名を入力します。
   b. [ランタイム] で、使用するランタイムを選択します。
   c. (オプション) [アーキテクチャ]で、関数の命令セットアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用の .zip デプロイパッケージと選択した命令セットのアーキテクチャに互換性があることを確認してください。
4. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。新しい [実行ロール] を作成することも、既存のロールを使用することもできます。
5. [関数の作成]を選択します。Lambda は、選択したランタイムを使用して基本的な「Hello world」関数を作成します。

ローカルマシンから zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、.zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [.zip ファイル] をクリックします。
5. .zip ファイルをアップロードするには、次の操作を行います。
   a. [アップロード] をクリックし、ファイルセレクターで .zip ファイルを選択します。
   b. [開く] をクリックします。
   c. [保存] をクリックします。

Amazon S3 バケットから .zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、新しい .zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [Amazon S3 ロケーション] を選択します。
5. .zip ファイルの Amazon S3 リンク URL を貼り付けて、[保存] を選択します。

コンソールコードエディタを使用して .zip ファイル関数を更新する

.zip デプロイパッケージを使用する一部の関数では、Lambda コンソールの組み込みコードエディタを使用して、関数コードを直接更新できます。この機能を使用するには、関数が次の基準を満たしている必要があります。

- 関数が、インタープリター言語ランタイムのいずれか (Python、Node.js、Ruby) を使用する必要があります。
- 関数のデプロイパッケージが 3 MB 未満である必要があります。

コンテナイメージデプロイパッケージを含む関数の関数コードは、コンソールで直接編集することはできません。

コンソールのコードエディタを使用して関数コードを更新するには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインでソースコードファイルを選択し、統合コードエディタで編集します。
4. コードの編集が終了したら、[デプロイ] を選択して変更を保存し、関数を更新します。

AWS CLI を使用して .zip ファイルで関数を作成、更新する

AWS CLI を使用して新しい関数を作成したり、.zip ファイルを使用して既存の関数を更新したりできます。create-function コマンドと update-function-code を使用して、.zip パッケージをデプロイします。.zip ファイルが 50 MB 未満の場合は、ローカルビルドマシン上のファイルの場所から .zip パッケージをアップロードできます。サイズの大きいファイルの場合は、Amazon S3 バケットか

ら .zip パッケージをアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> AWS CLI を使用して Amazon S3 バケットから .zip ファイルをアップロードする場合、このバケットは関数と同じ AWS リージョン に配置する必要があります。

AWS CLI を含む .zip ファイルを使用して新しい関数を作成するには、以下を指定する必要があります。

- 関数の名前 (`--function-name`)
- 関数のランタイム (`--runtime`)
- 関数の実行ロール (`--role`) の Amazon リソースネーム (ARN)
- 関数コード内のハンドラーメソッド (`--handler`) の名前

.zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、`--zip-file` オプションを使用してファイルパスを指定します。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある `--code` オプションを使用します。`S3ObjectVersion` パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs20.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

CLI を使用して既存の関数を更新するには、`--function-name` パラメータを使用して関数の名前を指定します。関数コードの更新に使用する .zip ファイルの場所も指定する必要があります。.zip

ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、`--zip-file` オプションを使用してファイルパスを指定します。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある `--s3-bucket` および `--s3-key` オプションを使用します。`--s3-object-version` パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API を使用して .zip ファイルで関数を作成、更新する

.zip ファイルアーカイブを使用して関数を作成および更新するには、以下の API オペレーションを使用します。

- CreateFunction
- UpdateFunctionCode

AWS SAM を使用して .zip ファイルで関数を作成、更新する

AWS Serverless Application Model (AWS SAM) は、AWS のサーバーレスアプリケーションの構築と実行のプロセスを合理化するのに役立つツールキットです。YAML または JSON テンプレートでアプリケーションのリソースを定義し、AWS SAM コマンドラインインターフェイス (AWS SAM CLI) を使用して、アプリケーションを構築、パッケージ化、デプロイします。AWS SAM テンプレートから Lambda 関数を構築すると、AWS SAM は関数コードと指定した任意の依存関係を含む .zip デプロイパッケージまたはコンテナイメージを自動的に作成します。AWS SAM を使用して Lambda 関数を構築およびデプロイする方法の詳細については、「AWS Serverless Application Model 開発者ガイドの」の「AWS SAM の開始方法」を参照してください。

AWS SAM を使用して、既存の .zip ファイルアーカイブを使用する Lambda 関数を作成できます。AWS SAM を使用して Lambda 関数を作成するには、.zip ファイルを Amazon S3 バケットまたはビルドマシンのローカルフォルダに保存します。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS SAM テンプレートでは、Lambda 関数は `AWS::Serverless::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `CodeUri` - 関数コードの Amazon S3 URI、ローカルフォルダへのパス、または FunctionCode オブジェクトに設定
- `Runtime` - 選択したランタイムに設定

AWS SAM では、.zip ファイルが 50 MB を超える場合、この .zip ファイルを最初に Amazon S3 バケットにアップロードする必要はありません。AWS SAM では、ローカルビルドマシン上の場所から、最大許容サイズ 250 MB (解凍) の .zip パッケージをアップロードできます。

AWS SAM で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS SAM 開発者ガイド」の「AWS::Serverless::Function」を参照してください。

AWS CloudFormation を使用して .zip ファイルで関数を作成、更新する

AWS CloudFormation を使用して、.zip ファイルアーカイブを使用する Lambda 関数を作成できます。.zip ファイルから Lambda 関数を作成するには、最初にファイルを Amazon S3 バケットにアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS CloudFormation テンプレートでは、Lambda 関数は `AWS::Lambda::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `Code` - `S3Bucket` および `S3Key` フィールドに Amazon S3 バケット名と .zip ファイル名を入力
- `Runtime` - 選択したランタイムに設定

AWS CloudFormation が生成する .zip ファイルは、4 MB を超えることはできません。AWS CloudFormation で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::Lambda::Function」を参照してください。

Node.js Lambda 関数をコンテナイメージとともにデプロイする

Node.js Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- Node.js の AWS ベースイメージを使用する

  AWS ベースイメージには、言語ランタイム、Lambda と関数コード間のやり取りを管理するランタイムインターフェースクライアント、ローカルテスト用のランタイムインターフェイスエミュレーターがプリロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、Node.js のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することもできます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、Node.js のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

トピック

- Node.js の AWS ベースイメージ

- Node.js の AWS ベースイメージを使用する
- ランタイムインターフェイスクライアントで代替ベースイメージを使用する

Node.js の AWS ベースイメージ

AWS では、Node.js 用の以下のベースイメージが利用できます。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
20	Node.js 20	Amazon Linux 2023	GitHub にある Node.js 20 用の Dockerfile	
18	Node.js 18	Amazon Linux 2	GitHub にある Node.js 18 用の Dockerfile	
16	Node.js 16	Amazon Linux 2	GitHub にある Node.js 16 用の Dockerfile	2024 年 6 月 12 日

Amazon ECR リポジトリ: gallery.ecr.aws/lambda/nodejs

Node.js 20 以降のベースイメージは、Amazon Linux 2023 の最小コンテナイメージに基づいています。以前のベースイメージでは Amazon Linux 2 が使用されています。AL2023 ランタイムには、デプロイのフットプリントが小さいことや、`glibc` などのライブラリのバージョンが更新されていることなど、Amazon Linux 2 に比べていくつかの利点があります。

AL2023 ベースのイメージでは、Amazon Linux 2 のデフォルトのパッケージマネージャである `yum` の代わりに `microdnf` (`dnf` としてシンボリックリンク) がパッケージマネージャとして使用されています。`microdnf` は `dnf` のスタンドアロン実装です。AL2023 ベースのイメージに含まれるパッケージのリストについては、「Comparing packages installed on Amazon Linux 2023 Container Images」の「Minimal Container」列を参照してください。AL2023 と Amazon Linux 2 の違いの詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。

> **Note**
>
> AWS Serverless Application Model (AWS SAM) を含む AL2023 ベースのイメージをローカルで実行するには、Docker バージョン 20.10.10 以降を使用する必要があります。

Node.js の AWS ベースイメージを使用する

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker (Node.js 20 以降のベースイメージの最小バージョンは 20.10.10)
- Node.js

ベースイメージからイメージを作成する

Node.js の AWS ベースイメージからコンテナイメージを作成する方法

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

   ```
   mkdir example
   cd example
   ```

2. npm で新しい Node.js プロジェクトを作成します。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

   ```
   npm init
   ```

3. index.js という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

   Example CommonJS ハンドラー

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
           body: JSON.stringify('Hello from Lambda!'),
   ```

```
    };
    return response;
};
```

4. 関数が AWS SDK for JavaScript 以外のライブラリに依存している場合、「npm」を使用してパッケージに追加します。
5. 次の設定で新しい Dockerfile を作成します。

   - FROM プロパティを「ベースイメージの URI」に設定します。
   - COPY コマンドを使用し、関数コードおよびランタイムの依存関係を {LAMBDA_TASK_ROOT} (Lambda 定義の環境変数) にコピーします。
   - CMD 引数を Lambda 関数ハンドラーに設定します。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:20

# Copy function code
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "index.handler" ]
```

6. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

1. docker run コマンドを使用して、Docker イメージを起動します。この例では、docker-image はイメージ名、test はタグです。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、--platform linux/*amd64* の代わりに --platform linux/*arm64* オプションを使用してください。

2. 新しいターミナルウィンドウから、イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の curl コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の Invoke-WebRequest コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - `111122223333` を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、`update-function-code` コマンドを使用する必要があります。

ランタイムインターフェイスクライアントで代替ベースイメージを使用する

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間の相互作用を管理する Lambda Runtime API を拡張します。

npm パッケージマネージャーを使用して、Node.js 用のランタイムインターフェイスクライアントをインストールします。

```
npm install aws-lambda-ric
```

Node.js のランタイムインターフェイスクライアントを GitHub からダウンロードすることもできます。ランタイムインターフェイスクライアントは、次の NodeJS バージョンをサポートしています。

- 14.x
- 16.x
- 18.x
- 20.x

次の例は、非 AWS ベースイメージを使用して、 Node.js 用のコンテナイメージを構築する方法について説明しています。`buster` ベースイメージを使用した Dockerfile の例を示します。Dockerfile には、ランタイムインターフェイスクライアントが含まれています。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker
- Node.js

代替ベースイメージからイメージを作成する

非 AWS ベースイメージからコンテナイメージを作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir example
cd example
```

2. npm で新しい Node.js プロジェクトを作成します。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

```
npm init
```

3. index.js という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

   Example CommonJS ハンドラー

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 新しい Dockerfile を作成します。次の Dockerfile では、AWS ベースイメージの代わりに buster ベースイメージを使用します。Dockerfile には、イメージに Lambda との互換性を持たせるランタイムインターフェイスクライアントが含まれています。Dockerfile では、マルチステージビルドを使用します。第 1 段階では、関数の依存関係がインストールされた標準の Node.js 環境であるビルドイメージを作成します。第2段階では、関数コードとその依存関係を含むスリムなイメージを作成します。これにより、最終的なイメージサイズが小さくなります。

   - ベースイメージ識別子に FROM プロパティを設定します。

- COPY コマンドを使用して、関数コードおよびランタイムの依存関係をコピーします。
- ENTRYPOINT を、Docker コンテナの起動時に実行させるモジュールに設定します。この場合、モジュールはランタイムインターフェイスクライアントです。
- CMD 引数を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-buster as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-buster-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
```

```
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。エミュレーターはイメージに組み込むことも、次の手順を使用してローカルマシンにインストールすることもできます。

ローカルマシンにランタイムインターフェイスエミュレーターをインストールして実行するには

1. プロジェクトディレクトリから次のコマンドを実行して、GitHub からランタイムインターフェイスエミュレーター (x86-64 アーキテクチャ) をダウンロードし、ローカルマシンにインストールします。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
   runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   arm64 エミュレータをインストールするには、前のコマンドの GitHub リポジトリ URL を次のように置き換えます。

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
   download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
   releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   arm64 エミュレーターをインストールするには、$downloadLink を次のように置き換えます。

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
   download/aws-lambda-rie-arm64
   ```

2. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。

   - `docker-image` はイメージ名、`test` はタグです。

- `/usr/local/bin/npx aws-lambda-ric index.handler` は ENTRYPOINT で、その後に Dockerfile の CMD が続きます。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/npx aws-lambda-ric index.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/npx aws-lambda-ric index.handler
```

このコマンドはイメージをコンテナとして実行し、`localhost:9000/2015-03-31/functions/function/invocations` でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、`--platform linux/amd64` の代わりに `--platform linux/arm64` オプションを使用してください。

3. イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の `curl` コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の `Invoke-WebRequest` コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. コンテナ ID を取得します。

```
docker ps
```

5. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

- `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
- 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。

4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   例:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず `:latest` を含めてください。

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。

7. Lambda 関数を作成します。`ImageUri` には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず `:latest` を含めてください。

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   > **Note**
   >
   > イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

Node.js の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティを提供します。

context メソッド

- `getRemainingTimeInMillis()` — 実行がタイムアウトするまでの残り時間をミリ秒で返します。

context プロパティ

- `functionName` - Lambda 関数の名前。
- `functionVersion` - 関数のバージョン。
- `invokedFunctionArn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `memoryLimitInMB` - 関数に割り当てられたメモリの量。
- `awsRequestId` - 呼び出しリクエストの ID。
- `logGroupName` - 関数のロググループ。
- `logStreamName` — 関数インスタンスのログストリーム。
- `identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
  - `cognitoIdentityId` - 認証された Amazon Cognito ID
  - `cognitoIdentityPoolId` — 呼び出しを承認した Amazon Cognito ID プール。
- `clientContext` — (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`

  - `env.make`
  - `env.model`
  - `env.locale`
  - `Custom` - クライアントアプリケーションで設定されたカスタム値。
- `callbackWaitsForEmptyEventLoop` - false に設定すると、Node.js イベントループが空になるまで待機することなく、コールバックが実行されるとすぐにレスポンスが送信されます。これが false の場合、未完了のイベントは、次の呼び出し中に実行され続けます。

次の例の関数はコンテキスト情報をログに記録して、そのログの場所を返します。

Example index.js ファイル

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Node.js の AWS Lambda 関数ログ作成

AWS Lambda は、ユーザーに代わって Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda ランタイム環境は、各呼び出しの詳細をログストリームに送信し、関数のコードからのログやその他の出力を中継します。詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

このページでは、AWS Command Line Interface、Lambda コンソール、または CloudWatch コンソールを使用して、Lambda 関数のコードからログ出力を生成する方法、またはアクセスログを生成する方法について説明します。

セクション

ログを返す関数の作成

関数コードからログを出力するには、コンソールオブジェクトのメソッドか、stdout または stderr に書き込む任意のログ記録のライブラリを使用します。次の例では、環境変数の値とイベントオブジェクトをログに記録します。

Example index.js ファイル - ログ記録

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Example ログの形式

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
```

```
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 200 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Node.js ランタイムは、呼び出しごとに START、END、および REPORT の各行を記録します。関数によってログに記録される各エントリに、タイムスタンプ、リクエスト ID、およびログレベルを追加します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。

- サンプリング済み - トレースされたリクエストの場合、サンプリング結果。

ログは、Lambda コンソール、CloudWatch Logs コンソール、またはコマンドラインで表示することができます。

Node.js での Lambda の高度なログ記録コントロールの使用

関数のログのキャプチャ、処理、使用方法をより細かく制御できるように、サポートされている Node.js ランタイムに以下のログ記録オプションを設定できます。

- ログの形式 - 関数のログをプレーンテキスト形式と構造化された JSON 形式から選択します
- ログレベル - JSON 形式のログの場合、Lambda が Amazon CloudWatch に送信するログの詳細レベル (ERROR、DEBUG、INFO など) を選択します。
- ロググループ - 関数がログを送信する CloudWatch ロググループを選択します

これらのログ記録オプションの詳細と、それらのオプションを使用するように関数を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。

Node.js Lambda 関数でログ形式とログレベルのオプションを使用するには、以下のセクションのガイダンスを参照してください。

Node.js での構造化 JSON ログの使用

関数のログ形式として JSON を選択した場合、Lambda は `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error`、および `console.warn` のコンソールメソッドを使用してログ出力を構造化された JSON として CloudWatch に送信します。各 JSON ログオブジェクトには、以下のキーを含む少なくとも 4 つのキーと値のペアが含まれます。

- `"timestamp"` - ログメッセージが生成された時刻
- `"level"` - メッセージに割り当てられたログレベル
- `"message"` - ログメッセージの内容
- `"requestId"` - 関数呼び出しの一意のリクエスト ID

関数が使用するログ記録方法によっては、この JSON オブジェクトに追加のキーペアが含まれる場合もあります。例えば、関数が `console` メソッドを使用して複数の引

数を使用しているエラーオブジェクトをログに記録する場合、JSON オブジェクトには、errorMessage、errorType、stackTrace というキーを含む追加のキーと値のペアが含まれます。

コードで既に Powertools for AWS Lambda などの別のログ記録ライブラリを使用して JSON 構造化ログを生成している場合は、変更を加える必要はありません。Lambda は既に JSON でエンコードされたログを二重にエンコードしないため、関数のアプリケーションログは以前と同様にキャプチャされます。

Powertools for AWS Lambda ログ記録パッケージを使用して Node.js ランタイムで JSON 構造化ログを作成する方法の詳細については、「the section called “ログ記録”」を参照してください。

JSON 形式のログ出力の例

次の例は、関数のログ形式を JSON に設定したときに、単一および複数の引数を持つ console メソッドを使用して生成されたさまざまなログ出力が CloudWatch Logs にどのようにキャプチャされるかを示しています。

最初の例では、console.error メソッドを使用して単純な文字列を出力します。

Example Node.js ログ記録コード

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON ログレコード

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

console メソッドでは、1 つまたは複数の引数を使用して、より複雑な構造化されたログメッセージを出力することもできます。次の例では、console.log を使用して、1 つの引数を使用して 2 つのキーと値のペアを出力します。Lambda が CloudWatch Logs に送信する JSON オブジェクトの "message" フィールドは、文字列化されていないことに注意してください。

Example Node.js ログ記録コード

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON ログレコード

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

次の例では、console.log メソッドを再度使用してログ出力を作成します。今回このメソッドは、2 つのキーと値のペアを含むマップと 1 つの識別文字列の 2 つの引数を取得します。この場合、2 つの引数を指定したため、Lambda は "message" フィールドを文字列化することに注意してください。

Example Node.js ログ記録コード

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON ログレコード

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda は、console.log を使用して生成された出力にログレベル INFO を割り当てます。

最後の例は、console メソッドを使用してエラーオブジェクトを CloudWatch Logs に出力する方法を示しています。複数の引数を使用してエラーオブジェクトをログに記録すると、Lambda はフィールド errorMessage、errorType、stackTrace を ログ出力に追加することに注意してください。

Example Node.js ログ記録コード

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON ログレコード

```
{
    "timestamp": "2023-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2023-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
```

```
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

複数のエラータイプをログに記録する場合、追加のフィールド `errorMessage`、`errorType`、`stackTrace` は `console` メソッドに最初に渡されたエラータイプから抽出されます。

構造化された JSON ログでの埋め込みメトリクスフォーマット (EMF、Embedded Metric Format) クライアントライブラリの使用

AWS では、埋め込みメトリクスフォーマット (EMF、Embedded Metric Format) ログの作成に使用できるオープンソースのクライアントライブラリを提供しています。これらのライブラリを使用する既存の関数があり、関数のログ形式を JSON に変更すると、CloudWatch はコードによって生成されたメトリクスを認識しなくなる可能性があります。

コードが現在、`console.log` または Powertools for AWS Lambda (TypeScript) を使用して EMF ログを直接出力している場合でも、関数のログ形式を JSON に変更すると、CloudWatch はこれらを解析できなくなります。

> **Important**
>
> 関数の EMF ログが引き続き CloudWatch によって適切に解析されるようにするには、EMF および Powertools for AWS Lambda ライブラリを最新バージョンに更新してください。JSON ログ形式に切り替える場合は、関数に埋め込まれているメトリクスとの互換性を確認するためのテストを実施することもお勧めします。`console.log` を使用してコードが EMF ログを直接出力する場合は、以下のコード例のように、これらのメトリクスを `stdout` に直接出力するようにコードを変更してください。

Example 埋め込みメトリクスを **`stdout`** に出力するコード

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
```

```
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

Node.js でのログレベルフィルタリングの使用

AWS Lambda でログレベルに従ってアプリケーションログをフィルタリングするには、関数で JSON 形式のログを使用する必要があります。このためには以下の 2 つの方法があります。

- 標準のコンソールメソッドを使用してログ出力を作成し、JSON ログ形式を使用するように関数を設定します。その後、AWS Lambda は「the section called “Node.js での構造化 JSON ログの使用”」で説明されている JSON オブジェクトの「level」キー値のペアを使用してログ出力をフィルタリングします。関数のログ形式を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。
- 別のログ記録ライブラリまたはメソッドを使用して、ログ出力のレベルを定義する「level」キーと値のペアを含む JSON 構造化ログをコード内に作成する。例えば、Powertools for AWS Lambda を使用してコードから JSON 構造化されたログ出力を生成できます。Node.js ランタイムで Powertools を使用する方法の詳細については、「the section called “ログ記録”」を参照してください。

  Lambda で関数のログをフィルタリングするには、JSON ログ出力に `"timestamp"` のキーと値のペアも含める必要があります。時間は、有効な RFC 3339 タイムスタンプ形式で指定する必要があります。有効なタイムスタンプを指定しない場合、Lambda はログに INFO レベルを割り当ててタイムスタンプを追加します。

ログレベルのフィルタリングを使用するように関数を設定する場合、AWS Lambda が CloudWatch Logs に送信するログのレベルを以下のオプションから選択します。

ログレベル	標準的な使用状況
TRACE (最も詳細)	コードの実行パスを追跡するために使用される最も詳細な情報
DEBUG	システムデバッグの詳細情報
INFO	関数の通常の動作を記録するメッセージ
WARN	対処しないと予期しない動作を引き起こす可能性がある潜在的なエラーに関するメッセージ
ERROR	コードが期待どおりに動作しなくなる問題に関するメッセージ
FATAL (詳細度が最も低い)	アプリケーションの機能停止を引き起こす重大なエラーに関するメッセージ

Lambda は、選択したレベル以下のログを CloudWatch に送信します。例えば、ログレベルを WARN に設定すると、Lambda は WARN、ERROR、FATAL の各レベルに対応するログを送信します。

Lambda コンソールを使用する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/**###**)

3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。 X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
```

```
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

AWS Lambda での Node.js コードの作成

Lambda アプリケーションのトレース、デバッグ、最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレースされたデータを X-Ray に送信するには、以下の 2 つの SDK ライブラリのいずれかを使用します。

- AWS Distro for OpenTelemetry (ADOT) – 安全で、本番環境に対応し、AWS でサポートされている OpenTelemetry (OTel) SDK のディストリビューションです。
- AWS X-Ray SDK for Node.js - トレースデータを生成して X-Ray に送信するための SDK です。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。これらのソリューションのいずれかを使用して、X-Ray でエンドツーエンドのトレーシングを実装することができます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

Node.js 関数の計測での ADOT の使用

ADOT はフルマネージド型の Lambda レイヤーを提供します。これにより、OTel SDK を使用してテレメトリデータを収集するために必要なすべてがパッケージ化されます。このレイヤーを使用すると、関数コードを変更する必要はなしで、Lambda 関数を計測できます。また、このレイヤーは、OTel でのカスタムな初期化を実行するように構成することもできます。詳細については、ADOT のドキュメントにある「Lambda 上での ADOT Collector のカスタム設定」を参照してください。

Node.js ランタイムの場合は、ADOT Javascript 向け AWS マネージド Lambda レイヤー を追加して、関数を自動的に計測できます。このレイヤーを追加する方法の詳しい手順については、「ADOT ドキュメント」で「JavaScript に対する AWS Distro for OpenTelemetry Lambda のサポート」を参照してください。

X-Ray SDK を使用した Node.js 関数の計測

Lambda 関数がアプリケーション内の他のリソースに対して行う呼び出しの詳細を記録するために、AWS X-Ray SDK for Node.js を使用することもできます。SDK を取得するには、アプリケーションの依存関係に `aws-xray-sdk-core` パッケージを追加します。

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

AWS SDK クライアントを AWS SDK for JavaScript v3 で計測するには、`captureAWSv3Client` メソッドでクライアントインスタンスをラップします。

Example blank-nodejs/function/index.js - AWS SDK クライアントのトレース

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Lambda ランタイムは、 X-Ray SDK を設定するための環境変数をいくつか設定します。例えば、Lambda は、X-Ray SDK からランタイムエラーがスローされないように、LOG_ERROR に AWS_XRAY_CONTEXT_MISSING をセットします。カスタムのコンテキストミッシング戦略を設定するには、関数設定の環境変数をオーバーライドして値を持たせないようにします。その後、コンテキストミッシング戦略をプログラムで設定できます。

Example 初期化コードの例

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

詳細については、「the section called “環境変数の設定”」を参照してください。

正しい依存関係を追加し、必要なコード変更を行った後で、Lambda コンソールまたはAPIを介して、関数の設定でトレースをアクティブにします。

Lambda コンソールを使用してトレースを有効化する

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。

4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [保存] をクリックします。

Lambda API でのトレースのアクティブ化

AWS CLI または AWS SDK で Lambda 関数のトレースを設定するには、次の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation によるトレースのアクティブ化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray トレースの解釈

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

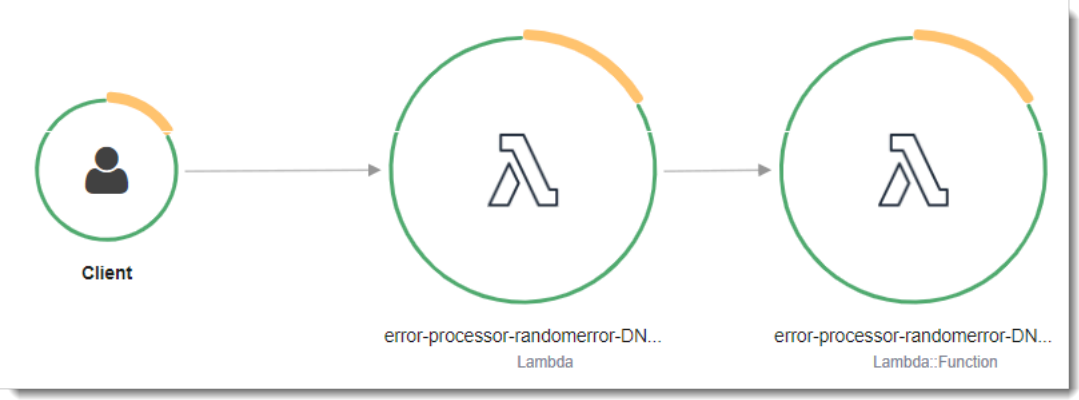
アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロググループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエ

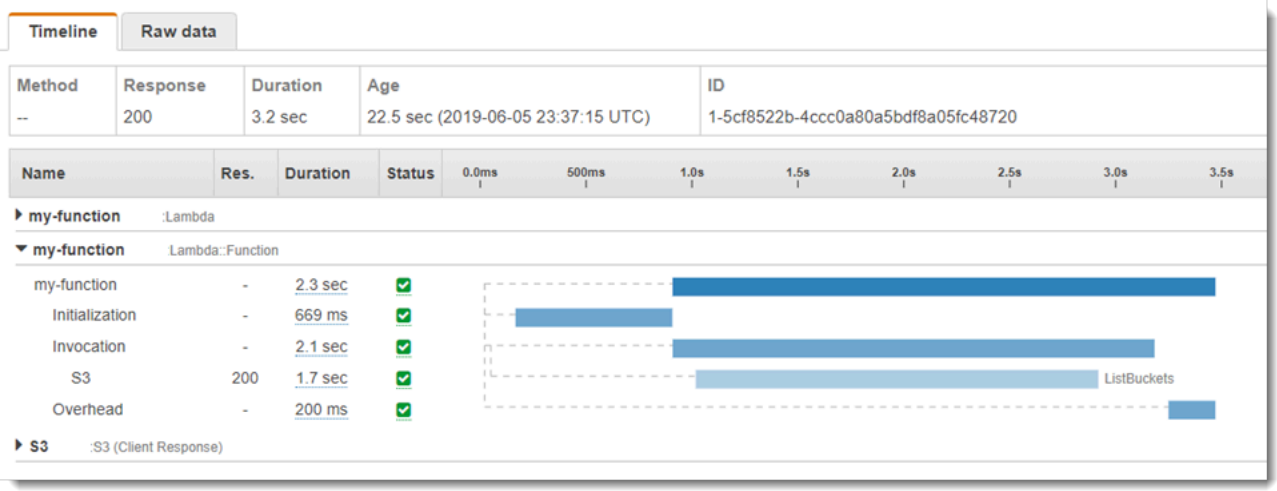
ストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは `AWS::Lambda` の起点があり、もう 1 つは `AWS::Lambda::Function` の起点があります。`AWS::Lambda` セグメントにエラーが表示される場合は、Lambda サービスに問題があります。`AWS::Lambda::Function` セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、「AWS X-Ray デベロッパーガイド」の「AWS X-Ray SDK for Node.js」を参照してください。

> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

ランタイムの依存関係をレイヤー (X-Ray SDK) に保存する

X-Ray SDK を使用して AWS SDK クライアントを関数コードに埋め込むと、デプロイパッケージが巨大になる可能性があります。関数コードを更新するたびにランタイムの依存関係がアップロードされないようにするには、X-Ray SDK を「Lambda レイヤー」にパッケージ化します。

次に、AWS X-Ray SDK for Node.js を保存している AWS::Serverless::LayerVersion リソースの例を示します。

Example template.yml - 依存関係レイヤー

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
```

```
    ...
libs:
  Type: AWS::Serverless::LayerVersion
  Properties:
    LayerName: blank-nodejs-lib
    Description: Dependencies for the blank sample app.
    ContentUri: lib/.
    CompatibleRuntimes:
      - nodejs16.x
```

この設定では、ランタイム依存関係を変更した場合にのみ、ライブラリレイヤーの更新が必要です。関数のデプロイパッケージにはユーザーのコードのみが含まれるため、アップロード時間を短縮できます。

依存関係のレイヤーを作成するには、デプロイ前にレイヤーアーカイブを生成するようにビルドを変更する必要があります。実際の例については、blank-nodejs サンプルアプリケーションを参照してください。

TypeScript による Lambda 関数の作成

Node.js ランタイムを使用すると、TypeScript コードを AWS Lambda で実行できます。Node.js は TypeScript コードをネイティブに実行しないため、最初に TypeScript から JavaScript へのコード変換 (トランスパイル) を行う必要があります。次に、JavaScript ファイルを使用して、この関数コードを Lambda にデプロイします。このコードは、ユーザーが管理する AWS Identity and Access Management (IAM) ロールの認証情報を使用することで、AWS SDK for JavaScript を含む環境内で実行できます。Node.js ランタイムに含まれている SDK バージョンの詳細については、「」を参照してくださいthe section called “ランタイムに含まれる SDK バージョン”。

Lambda は、以下の Node.js ランタイムをサポートしています。

Node.js

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024 年 6 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日

トピック

- TypeScript 開発環境のセットアップ
- TypeScript の Lambda 関数ハンドラーの定義
- トランスパイルされた TypeScript コードを .zip ファイルアーカイブで Lambda にデプロイする
- コンテナイメージを使用して、トランスパイルされた TypeScript コードを Lambda にデプロイする
- AWS Lambda の context オブジェクト TypeScript
- TypeScript での AWS Lambda 関数ログ
- での TypeScript コードのトレース AWS Lambda

TypeScript 開発環境のセットアップ

TypeScript 関数のコードを記述するには、ローカルの統合開発環境 (IDE)、テキストエディタ、または AWS Cloud9 を使用します。Lambda コンソールでは TypeScript コードを作成できません。

TypeScript コードのトランスパイルのためには、esbuild のようなコンパイラ、またはマイクロソフトの TypeScript コンパイラ (`tsc`) をセットアップします。これらは、TypeScript ディストリビューションに同梱されています。AWS Serverless Application Model (AWS SAM) または AWS Cloud Development Kit (AWS CDK) を使用すると、TypeScript コードのビルドとデプロイを簡素化できます。どちらのツールも、TypeScript コードから JavaScript へのトランスパイルに esbuild を使用します。

esbuild を使用する際は、以下を考慮してください。

- TypeScript に関する注意事項がいくつか存在します。
- TypeScript のトランスパイルには、使用する予定の Node.js ランタイムに適合する設定を行う必要があります。詳細については、esbuild ドキュメントの「Target」(ターゲット) を参照してください。Lambda でサポートされている特定の Node.js バージョンをターゲットにする場合の、tsconfig.json ファイルの例については、TypeScript GitHub リポジトリを参照してください。
- esbuild では、型チェックは行われません。型をチェックする場合は、`tsc` コンパイラを使用します。次の例に示すように、`tsc -noEmit` を実行するか、tsconfig.json ファイルに `"noEmit"` パラメータを追加します。これを設定することで、`tsc` は JavaScript ファイルを出力しなくなります。型のチェックが終了したら、esbuild を使用して TypeScript ファイルを JavaScript に変換します。

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
```

```
    "isolatedModules": true,
  },
  "exclude": ["node_modules", "**/*.test.ts"]
}
```

TypeScript の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

Example TypeScript ハンドラー

この例の関数では、イベントオブジェクトの内容をログに記録して、そのログの場所を返します。次の点に注意してください。

- このコードを Lambda 関数で使用する前に、開発環境の依存関係として @types/aws-lambda パッケージを追加する必要があります。このパッケージには Lambda の型定義が含まれています。`@types/aws-lambda` がインストールされると、`import` ステートメント (`import ... from 'aws-lambda'`) は型定義をインポートします。`aws-lambda` NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、「DefinitelyTyped GitHub リポジトリ」の「aws-lambda」を参照してください。
- この例でのハンドラーは ES モジュールであり、`package.json` ファイル内で、または `.mjs` ファイル拡張子を使用して、そのように指定する必要があります。詳細については、「ES モジュールとしての関数ハンドラーの指定」を参照してください。

```
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
    console.log('EVENT: \n' + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

ランタイムでは、ハンドラーメソッドに引数を渡します。最初の引数は、呼び出し元からの情報を含む `event` オブジェクトです。呼び出し元は、Invoke を呼び出してこの情報を JSON 形式の文字列として渡し、ランタイムはそれをオブジェクトに変換します。AWS のサービスで関数を呼び出す場合、そのイベント構造はサービスによって異なります。TypeScript では、イベントオブジェクトに型注釈を使用することをお勧めします。詳細については、「イベントオブジェクトへの型の使用」を参照してください。

2 番目の引数は、コンテキストオブジェクトです。この引数には、呼び出し、関数、および実行環境に関する情報が含まれます。前述の例では、関数は、コンテキストオブジェクトからログストリームの名前を取得し、それを呼び出し元に返します。

コールバック引数を使用することもできます。これは、非同期ではないハンドラーで呼び出してレスポンスを送信できる関数です。コールバックの代わりに async/await を使用することをお勧めします。async/await により、読みやすさ、エラー処理、および効率が向上します。async/await とコールバックの違いの詳細については、「コールバックの使用」を参照してください。

async/await の使用

コードが非同期タスクを実行する場合は、async/await パターンを使用して、ハンドラーが実行を確実に終了するようにします。async/await は、Node.js で非同期コードを記述するための簡潔で読みやすい方法であり、ネストされたコールバックや連鎖する promise を必要としません。async/await を使用すると、非同期かつノンブロッキングでありながら、同期コードのように読み取るコードを記述できます。

`async` キーワードは関数を非同期としてマークし、`await` キーワードは `Promise` が解決されるまで関数の実行を一時停止します。

Example TypeScript 関数 — 非同期

この例では、`nodejs18.x` ランタイムで使用できる `fetch` を使用します。次の点に注意してください。

- このコードを Lambda 関数で使用する前に、開発環境の依存関係として @types/aws-lambda パッケージを追加する必要があります。このパッケージには Lambda の型定義が含まれています。`@types/aws-lambda` がインストールされると、`import` ステートメント (`import ... from 'aws-lambda'`) は型定義をインポートします。`aws-lambda` NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、「DefinitelyTyped GitHub リポジトリ」の「aws-lambda」を参照してください。
- この例でのハンドラーは ES モジュールであり、`package.json` ファイル内で、または `.mjs` ファイル拡張子を使用して、そのように指定する必要があります。詳細については、「ES モジュールとしての関数ハンドラーの指定」を参照してください。

```
import { APIGatewayProxyEvent, APIGatewayProxyResult } from 'aws-lambda';
const url = 'https://aws.amazon.com/';
export const lambdaHandler = async (event: APIGatewayProxyEvent):
 Promise<APIGatewayProxyResult> => {
    try {
        // fetch is available with Node.js 18
        const res = await fetch(url);
        return {
```

```
            statusCode: res.status,
            body: JSON.stringify({
                message: await res.text(),
            }),
        };
    } catch (err) {
        console.log(err);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'some error happened',
            }),
        };
    }
};
```

コールバックの使用

コールバックを使用する代わりに、async/await を使用して関数ハンドラーを宣言することをお勧めします。いくつかの理由により、async/await の方が適しています。

- 読みやすさ: async/await コードは、コールバックコードよりも読みやすく理解しやすいです。コールバックコードは、すぐに理解するのが難しくなり、コールバック地獄に陥る可能性があります。
- デバッグとエラー処理: コールバックベースのコードのデバッグは難しい場合があります。コールスタックを追跡するのが難しくなり、エラーが簡単に隠れてしまう可能性があります。async/await では、try/catch ブロックを使用してエラーを処理できます。
- 効率: コールバックでは、多くの場合、コードのさまざまな部分を切り替える必要があります。async/await を使用すると、コンテキストスイッチの回数を減らすことができるため、コードがより効率的になります。

ハンドラーでコールバックを使用すると、関数は、イベントループが空になるか、関数がタイムアウトするまで実行を続けます。レスポンスは、すべてのイベントループタスクが完了するまで、呼び出し元に送信されません。関数がタイムアウトした場合は、エラーが返ります。すぐにレスポンスが返るようにランタイムを設定するには、context.callbackWaitsForEmptyEventLoop を false に設定します。

コールバック関数は、`Error` とレスポンスの 2 つの引数を取ります。応答オブジェクトは、`JSON.stringify` と互換性がある必要があります。

Example コールバックを持つ TypeScript 関数

このサンプル関数は Amazon API Gateway からイベントを受け取り、イベントとコンテキストオブジェクトをログに記録して、API ゲートウェイ にレスポンスを返します。次の点に注意してください。

- このコードを Lambda 関数で使用する前に、開発環境の依存関係として @types/aws-lambda パッケージを追加する必要があります。このパッケージには Lambda の型定義が含まれています。`@types/aws-lambda` がインストールされると、`import` ステートメント (`import ... from 'aws-lambda'`) は型定義をインポートします。`aws-lambda` NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、「DefinitelyTyped GitHub リポジトリ」の「aws-lambda」を参照してください。
- この例でのハンドラーは ES モジュールであり、`package.json` ファイル内で、または `.mjs` ファイル拡張子を使用して、そのように指定する必要があります。詳細については、「ES モジュールとしての関数ハンドラーの指定」を参照してください。

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

イベントオブジェクトへの型の使用

型をチェックする機能が失われるため、ハンドラー引数と return 型に any 型を使用しないことをお勧めします。代わりに、sam local generate-event AWS Serverless Application Model CLI コマンドを使用してイベントを生成するか、または @types/aws-lambda パッケージからオープンソース定義を使用します。

sam local generate-event コマンドを使用したイベントの生成

1. Amazon Simple Storage Service (Amazon S3) プロキシイベントを生成します。

```
sam local generate-event s3 put >> S3PutEvent.json
```

2. QuickType ユーティリティを使用して、S3PutEvent.json ファイルから型定義を生成します。

```
npm install -g quicktype
quicktype S3PutEvent.json -o S3PutEvent.ts
```

3. 生成された型をコードで使用します。

```
import { S3PutEvent } from './S3PutEvent';

export const lambdaHandler = async (event: S3PutEvent): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

@types/aws-lambda パッケージのオープンソース定義を使用したイベントの生成

1. 開発環境の依存関係として @types/aws-lambda パッケージを追加します。

```
npm install -D @types/aws-lambda
```

2. コードで型を使用します。

```
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

トランスパイルされた TypeScript コードを .zip ファイルアーカイブで Lambda にデプロイする

TypeScript コードを にデプロイする前にAWS Lambda、コードを にトランスパイルする必要があります JavaScript。このページでは、.zip ファイルアーカイブを使用して TypeScript コードをビルドして Lambda にデプロイする 3 つの方法について説明します。

- AWS Serverless Application Model の使用 (AWS SAM)
- AWS Cloud Development Kit (AWS CDK) の使用
- AWS Command Line Interface (AWS CLI) および esbuild の使用

AWS SAM と は、 TypeScript 関数の構築とデプロイAWS CDKを簡素化します。AWS SAM テンプレート仕様は、サーバーレスアプリケーションを構成する Lambda 関数、API、アクセス許可、設定、およびイベントを記述するためのシンプルで簡潔な構文を提供します。AWS CDK を使用すると、プログラミング言語の優れた表現力を活かして、信頼性が高く、スケーラブルで、コスト効率の高いアプリケーションをクラウドで構築できます。AWS CDK は、中級 ~ 上級レベルの AWS ユーザーを対象としています。AWS CDK と はどちらも esbuild AWS SAMを使用して TypeScript コードを にトランスパイルします JavaScript。

AWS SAM を使用して Lambda に TypeScript コードをデプロイする

以下の手順に従って、 を使用して Hello World サンプル TypeScript アプリケーションをダウンロード、ビルド、デプロイしますAWS SAM。このアプリケーションは、基本的な API バックエンドを実装し、Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数が呼び出されます。関数は `hello world` のメッセージを返します。

> **Note**
>
> AWS SAM は esbuild を使用して TypeScript コードから Node.js Lambda 関数を作成します。esbuild サポートは現在パブリックプレビュー中です。パブリックプレビュー中は、esbuild のサポートは、下位互換性のない変更の対象となる場合があります。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS CLI バージョン 2
- AWS SAM CLI バージョン 1.75 以降
- Node.js 18.x

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World TypeScript テンプレートを使用してアプリケーションを初期化します。

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. (オプション) サンプルアプリケーションには、コードの linting 用の ESLint やユニットテスト用の Jest など、一般的に使用されるツールの設定が含まれています。lint コマンドとテストコマンドを実行するには、次のコードを使用します。

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. アプリケーションを構築します。

```
cd sam-app
sam build
```

4. アプリケーションをデプロイします。

```
sam deploy --guided
```

5. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押して応答します。
6. REST API のエンドポイントが出力に表示されます。ブラウザでエンドポイントを開き、関数をテストします。次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

を使用して Lambda に TypeScript コードをAWS CDKデプロイする

を使用してサンプル TypeScript アプリケーションを構築およびデプロイするには、以下のステップに従いますAWS CDK。このアプリケーションは、基本的な API バックエンドを実装し、API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数が呼び出されます。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS CLI バージョン 2
- AWS CDK バージョン 2
- Node.js 18.x
- Docker または esbuild のどちらか

AWS CDK サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

   ```
   mkdir hello-world
   cd hello-world
   ```

2. アプリケーションを初期化します。

   ```
   cdk init app --language typescript
   ```

3. 開発環境の依存関係として @types/aws-lambda パッケージを追加します。このパッケージには Lambda の型定義が含まれています。

   ```
   npm install -D @types/aws-lambda
   ```

4. lib ディレクトリを開きます。hello-world-stack.ts というファイルが表示されます。このディレクトリに hello-world.function.ts と hello-world.ts の 2 つの新しいファイルを作成します。

5. hello-world.function.ts を開き、次のコードをファイルに追加します。これは Lambda 関数のコードです。

> **Note**
>
> import ステートメントは、@types/aws-lambda から型定義をインポートします。aws-lambda NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、 DefinitelyTyped GitHub リポジトリの「aws-lambda」を参照してください。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. hello-world.ts を開き、次のコードをファイルに追加します。これには、Lambda 関数を作成するNodejsFunction コンストラクト と、REST API を作成するLambdaRestApi コンストラクト が含まれます。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
```

```
}
```

NodejsFunction コンストラクトでは、デフォルトで次のことを前提としています。

- 関数ハンドラーは handler と呼ばれます。
- 関数コードを含む .ts ファイル (hello-world.function.ts) は、コンストラクトを含む .ts ファイル (hello-world.ts) と同じディレクトリに存在します。コンストラクトは、コンストラクトの ID (「hello-world」) と Lambda ハンドラーファイルの名前 (「function」) を使用して、関数コードを検索します。例えば、関数コードが hello-world.my-function.ts という名前のファイルに含まれている場合、hello-world.ts ファイルは、関数コードを次のようにして参照する必要があります。

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

この動作を変更したり、他の esbuild パラメータを設定したりできます。詳細については、「AWS CDK API リファレンス」の「Configuring esbuild」(esbuild の設定) を参照してください。

7. hello-world-stack.ts を開きます。これは、AWS CDK スタックを定義するコードです。このコードを、次のコードで置き換えます。

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. cdk.json ファイルが含まれる hello-world ディレクトリから、アプリケーションをデプロイします。

```
cdk deploy
```

9. AWS CDK は、esbuild を使用して Lambda 関数をビルドおよびパッケージ化し、その関数を Lambda ランタイムにデプロイします。REST API のエンドポイントが出力に表示されます。ブラウザでエンドポイントを開き、関数をテストします。次のレスポンスが表示されます。

```
{"message":"hello world"}
```

これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

AWS CLI および esbuild を使用した Lambda への TypeScript コードのデプロイ

次の例は、esbuild と を使用して TypeScript コードをトランスパイルして Lambda にデプロイする方法を示していますAWS CLI。esbuild は、すべての依存関係を持つ 1 つの JavaScript ファイルを生成します。これは、.zip アーカイブに追加する必要がある唯一のファイルです。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS CLI バージョン 2
- Node.js 18.x
- Lambda 関数の実行ロール
- Windows ユーザーの場合は、7zip などの zip ファイルユーティリティ。

サンプル関数をデプロイする

1. ローカルマシンで、新しい関数のプロジェクトディレクトリを作成します。
2. npm または任意のパッケージマネージャーを使用して、新しい Node.js プロジェクトを作成します。

```
npm init
```

3. @types/aws-lambda および esbuild のパッケージを開発環境の依存関係として追加します。`@types/aws-lambda` パッケージには Lambda の型定義が含まれています。

```
npm install -D @types/aws-lambda esbuild
```

4. index.ts という名前の新しいファイルを作成します。次のコードを新しいファイルに追加します。これは Lambda 関数のコードです。関数は `hello world` のメッセージを返します。関数は、API Gateway リソースを作成しません。

> **Note**
>
> `import` ステートメントは、@types/aws-lambda から型定義をインポートします。`aws-lambda` NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、 DefinitelyTyped GitHub リポジトリの「aws-lambda」を参照してください。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. ビルドスクリプトを package.json ファイルに追加します。これにより、esbuild が .zip デプロイパッケージを自動的に作成するように設定されます。詳細については、esbuild ドキュメントの「Build scripts」(ビルドスクリプト) を参照してください。

   Linux and MacOS

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

Windows

この例では、"postbuild" コマンドは 7zip ユーティリティを使用して .zip ファイルを作成します。お好みの Windows zip ユーティリティを使用し、必要に応じてコマンドを変更します。

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. パッケージをビルドします。

```
npm run build
```

7. .zip デプロイパッケージを使用して Lambda 関数を作成します。ハイライト表示されたテキストを、実行ロールの Amazon リソースネーム (ARN) で置き換えます。

```
aws lambda create-function --function-name hello-world --runtime "nodejs18.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. テストイベントを実行して、関数が次のレスポンスを返すことを確認します。API Gateway を使用してこの関数を呼び出す場合は、REST API を作成および設定してください。

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

コンテナイメージを使用して、トランスパイルされた TypeScript コードを Lambda にデプロイする

TypeScript コードを Node.js コンテナイメージとして AWS Lambda 関数にデプロイできます。AWS は、コンテナイメージのビルドに役立つ Node.js のベースイメージを提供します。これらのベースイメージには、Lambda でイメージを実行するために必要な言語ランタイムおよびその他のコンポーネントがプリロードされています。AWS は、コンテナイメージの構築に役立つ各ベースイメージの Dockerfile を提供します。

コミュニティベースイメージまたはプライベートエンタープライズベースイメージを使用する場合は、ベースイメージに Node.js ランタイムインターフェイスクライアント (RIC) を追加して、Lambda と互換性を持たせる必要があります。

Lambda は、ローカルでテストするためのランタイムインターフェイスエミュレータを提供します。Node.js の AWS ベースイメージには、ランタイムインターフェイスエミュレータが含まれています。Alpine Linux や Debian イメージなどの代替ベースイメージを使用する場合は、エミュレーターをイメージにビルドすることもできますし、ローカルマシンにインストールすることもできます。

Node.js ベースイメージを使用して TypeScript 関数コードをビルドおよびパッケージ化する

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker
- Node.js 18.x

ベースイメージからイメージを作成する

AWS ベースイメージから Lambda のイメージを作成するには

1. ローカルマシンで、新しい関数のプロジェクトディレクトリを作成します。
2. npm または任意のパッケージマネージャーを使用して、新しい Node.js プロジェクトを作成します。

```
npm init
```

3. @types/aws-lambda および esbuild のパッケージを開発環境の依存関係として追加します。@types/aws-lambda パッケージには Lambda の型定義が含まれています。

```
npm install -D @types/aws-lambda esbuild
```

4. ビルドスクリプトを package.json ファイルに追加します。

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. index.ts という名前のファイルを作成します。その新しいファイルに次のサンプルコードを追加します。これは Lambda 関数のコードです。関数は hello world のメッセージを返します。

> **Note**
>
> import ステートメントは、@types/aws-lambda から型定義をインポートします。aws-lambda NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、「DefinitelyTyped GitHub リポジトリ」の「aws-lambda」を参照してください。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 次の設定で新しい Dockerfile を作成します。

   - ベースイメージの URI に FROM プロパティを設定します。
   - CMD 引数を設定して、Lambda 関数ハンドラを指定します。

   Example Dockerfile

   次の Dockerfile は、マルチステージビルドを使用します。最初のステップでは、TypeScript コードを JavaScript にトランスパイルします。2 番目のステップでは、JavaScript ファイルと本番環境の依存関係のみを含むコンテナイメージを生成します。

```
FROM public.ecr.aws/lambda/nodejs:18 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:18
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

1. docker run コマンドを使用して、Docker イメージを起動します。この例では、docker-image はイメージ名、test はタグです。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、--platform linux/*amd64* の代わりに --platform linux/*arm64* オプションを使用してください。

2. 新しいターミナルウィンドウから、イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の curl コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の Invoke-WebRequest コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、`update-function-code` コマンドを使用する必要があります。

AWS Lambda の context オブジェクト TypeScript

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティを提供します。

context メソッド

- `getRemainingTimeInMillis()` — 実行がタイムアウトするまでの残り時間をミリ秒で返します。

context プロパティ

- `functionName` - Lambda 関数の名前。
- `functionVersion` - 関数のバージョン。
- `invokedFunctionArn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `memoryLimitInMB` - 関数に割り当てられたメモリの量。
- `awsRequestId` - 呼び出しリクエストの ID。
- `logGroupName` - 関数のロググループ。
- `logStreamName` — 関数インスタンスのログストリーム。
- `identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
  - `cognitoIdentityId` - 認証された Amazon Cognito ID
  - `cognitoIdentityPoolId` — 呼び出しを承認した Amazon Cognito ID プール。
- `clientContext` — (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`

  - `env.make`
  - `env.model`
  - `env.locale`
  - `Custom` - クライアントアプリケーションで設定されたカスタム値。
- `callbackWaitsForEmptyEventLoop` - false に設定すると、Node.js イベントループが空になるまで待機することなく、コールバックが実行されるとすぐにレスポンスが送信されます。これが false の場合、未完了のイベントは、次の呼び出し中に実行され続けます。

@types /aws-lambda npm パッケージを使用して、コンテキストオブジェクトを操作できます。

Example index.ts ファイル

次の例の関数はコンテキスト情報をログに記録して、そのログの場所を返します。

> **Note**
>
> このコードを Lambda 関数で使用する前に、開発環境の依存関係として @types/aws-lambda パッケージを追加する必要があります。このパッケージには Lambda の型定義が含まれています。`@types/aws-lambda` がインストールされると、`import` ステートメント (`import ... from 'aws-lambda'`) は型定義をインポートします。`aws-lambda` NPM パッケージはインポートされません。これは関連性のないサードパーティーのツールです。詳細については、 DefinitelyTyped GitHub リポジトリの「aws-lambda」を参照してください。

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

TypeScript での AWS Lambda 関数ログ

AWS Lambda は、Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログエントリを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda のランタイム環境は、各呼び出しの詳細や、関数のコードからのその他の出力をログストリームに送信します。CloudWatch Logs の詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

関数コードからログを出力するには、コンソールオブジェクトのメソッドを使用できます。より詳細なロギングを行うため、`stdout` または `stderr` に書き込みを行う任意のログ記録ライブラリを使用できます。

セクション

- ツールとライブラリ
- Powertools for AWS Lambda (TypeScript) と構造化ログ用の AWS SAM の使用
- Powertools for AWS Lambda (TypeScript) と構造化ログ用の AWS CDK の使用
- Lambda コンソールを使用する
- CloudWatch コンソールの使用

ツールとライブラリ

Powertools for AWS Lambda (TypeScript) は、サーバーレスのベストプラクティスを実装し、開発者の作業速度を向上させるための開発者ツールキットです。ロガーユーティリティは、Lambda に最適化されたロガーを提供します。このロガーは、すべての関数の関数コンテキストに関する追加情報を含み、JSON 形式で構成されて出力されます。このユーティリティを使用して次のことができます。

- Lambda の コンテキスト、コールドスタート、構造から主要キーをキャプチャし、JSON 形式でログ出力する
- 指示された場合 Lambda 呼び出しイベントをログ記録する (デフォルトでは無効)
- ログサンプリングにより、特定の割合の呼び出しにのみすべてのログを出力する (デフォルトでは無効)
- 任意のタイミングで、構造化されたログにキーを追加する
- カスタムログフォーマッター (Bring Your Own Formatter) を使用して、組織の ログ記録 RFC と互換性のある構造でログを出力する

Powertools for AWS Lambda (TypeScript) と構造化ログ用の AWS SAM の使用

AWS SAM を使用した統合 Powertools for AWS Lambda (TypeScript) モジュールを使用して、Hello World TypeScript サンプルアプリケーションをダウンロード、ビルド、およびデプロイする場合は、以下の手順に従ってください。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Node.js 18.x 以降
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World TypeScript テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、`Enter` を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name sam-app
```

ログ出力は次のようになります。

```
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.594000
 2022-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.595000
 2022-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2022-08-31T09:33:10.594Z"}
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.655000
 2022-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
```

```
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2023/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2022-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

ログ保持の管理

関数を削除しても、ロググループは自動的には削除されません。ログを無期限に保存しないようにするには、ロググループを削除するか、CloudWatch がログを自動的に削除するまでの保持期間を設定します。ログ保持を設定するには、AWS SAM テンプレートに以下を追加します。

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Powertools for AWS Lambda (TypeScript) と構造化ログ用の AWS CDK の使用

AWS CDK を使用した統合 Powertools for AWS Lambda (TypeScript) モジュールを使用して、Hello World TypeScript サンプルアプリケーションをダウンロード、ビルド、およびデプロイする場合は、以下の手順に従ってください。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを

送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Node.js 18.x 以降
- AWS CLI バージョン 2
- AWS CDK バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS CDK サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

```
mkdir hello-world
cd hello-world
```

2. アプリケーションを初期化します。

```
cdk init app --language typescript
```

3. 開発環境の依存関係として @types/aws-lambda パッケージを追加します。

```
npm install -D @types/aws-lambda
```

4. Powertools Logger ユーティリティをインストールします。

```
npm install @aws-lambda-powertools/logger
```

5. lib ディレクトリを開きます。hello-world-stack.ts という名前のファイルが表示されます。このディレクトリに hello-world.function.ts と hello-world.ts の 2 つの新しいファイルを作成します。
6. hello-world.function.ts を開き、次のコードをファイルに追加します。これは Lambda 関数のコードです。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
```

```
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. hello-world.ts を開き、次のコードをファイルに追加します。これには、Lambda 関数を作成し、Powertools の環境変数を設定し、ログ保持を 1 週間に設定する NodejsFunction コンストラクトが含まれています。また、REST API を作成する LambdaRestApi コンストラクトも含まれています。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
```

```
}
```

8. hello-world-stack.ts を開きます。これは、AWS CDK スタックを定義するコードです。このコードを、次のコードで置き換えます。

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. プロジェクトディレクトリに戻ります。

```
cd hello-world
```

10. アプリケーションをデプロイします。

```
cdk deploy
```

11. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. API エンドポイントを呼び出します。

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

13. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name HelloWorldStack
```

ログ出力は次のようになります。

```
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2022-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2023/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2022-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
cdk destroy
```

Lambda コンソールを使用する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/**###**)

3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。 X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

での TypeScript コードのトレース AWS Lambda

Lambda アプリケーションのトレース、デバッグ、最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレーシングデータを X-Ray に送信するには、以下に表示された 3 つの SDK ライブラリのいずれかを使用できます。

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK の、安全で本番環境に対応した AWSがサポートするディストリビューション。
- AWS X-Ray SDK for Node.js – トレースデータを生成して X-Ray に送信するための SDK。
- Powertools for AWS Lambda (TypeScript) — サーバーレスのベストプラクティスを実装し、開発者の作業速度を向上させるための開発者ツールキット。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。どちらのソリューションを使用しても、X-Ray で end-to-end トレースを実装できます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

- トレースに Powertools for AWS Lambda (TypeScript) と AWS SAM を使用する
- AWS Lambda (TypeScript) に Powertools を使用し、トレースに AWS CDKを使用する
- X-Ray トレースの解釈

トレースに Powertools for AWS Lambda (TypeScript) と AWS SAM を使用する

以下の手順に従って、 を使用して統合された Powertools for AWS Lambda (TypeScript) モジュールで Hello World サンプル TypeScript アプリケーションをダウンロード、構築、デプロイします AWS SAM。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数が呼び出し、埋め込みメトリクス形式を使用してログとメトリクスを に送信し CloudWatch、トレースを に送信しますAWS X-Ray。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Node.js 18.x 以降
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World TypeScript テンプレートを使用してアプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、`Enter` を押します。

> **Note**
>
> にHelloWorldFunction 認証が定義されていない可能性があります。これは問題ありませんか？ 必ず と入力してくださいy。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

AWS Lambda (TypeScript) に Powertools を使用し、トレースに AWS CDK を使用する

を使用して、統合された Powertools for AWS Lambda (TypeScript) モジュールで Hello World サンプル TypeScript アプリケーションをダウンロード、構築、デプロイするには、以下のステップに従いますAWS CDK。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数が呼び出し、埋め込みメトリクス形式を使用してログとメトリクスを に送信し CloudWatch、トレースを に送信しますAWS X-Ray。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Node.js 18.x 以降
- AWS CLI バージョン 2
- AWS CDK バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS Cloud Development Kit (AWS CDK) サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

```
mkdir hello-world
cd hello-world
```

2. アプリケーションを初期化します。

```
cdk init app --language typescript
```

3. 開発環境の依存関係として @types/aws-lambda パッケージを追加します。

```
npm install -D @types/aws-lambda
```

4. Powertools Tracer ユーティリティをインストールします。

```
npm install @aws-lambda-powertools/tracer
```

5. lib ディレクトリを開きます。hello-world-stack.ts というファイルが表示されます。このディレクトリに hello-world.function.ts と hello-world.ts の 2 つの新しいファイルを作成します。
6. hello-world.function.ts を開き、次のコードをファイルに追加します。これは Lambda 関数のコードです。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
```

```
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. hello-world.ts を開き、次のコードをファイルに追加します。これには、Lambda 関数を作成し、Powertools の環境変数を設定し、ログの保持期間を 1 週間に設定するNodejsFunction コンストラクト が含まれています。また、REST API LambdaRestApi を作成するコンストラクト も含まれています。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
```

```
  }
}
```

8. hello-world-stack.ts を開きます。これは、AWS CDK スタックを定義するコードです。このコードを、次のコードで置き換えます。

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. アプリケーションをデプロイします。

```
cd ..
cdk deploy
```

10. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. API エンドポイントを呼び出します。

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

12. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

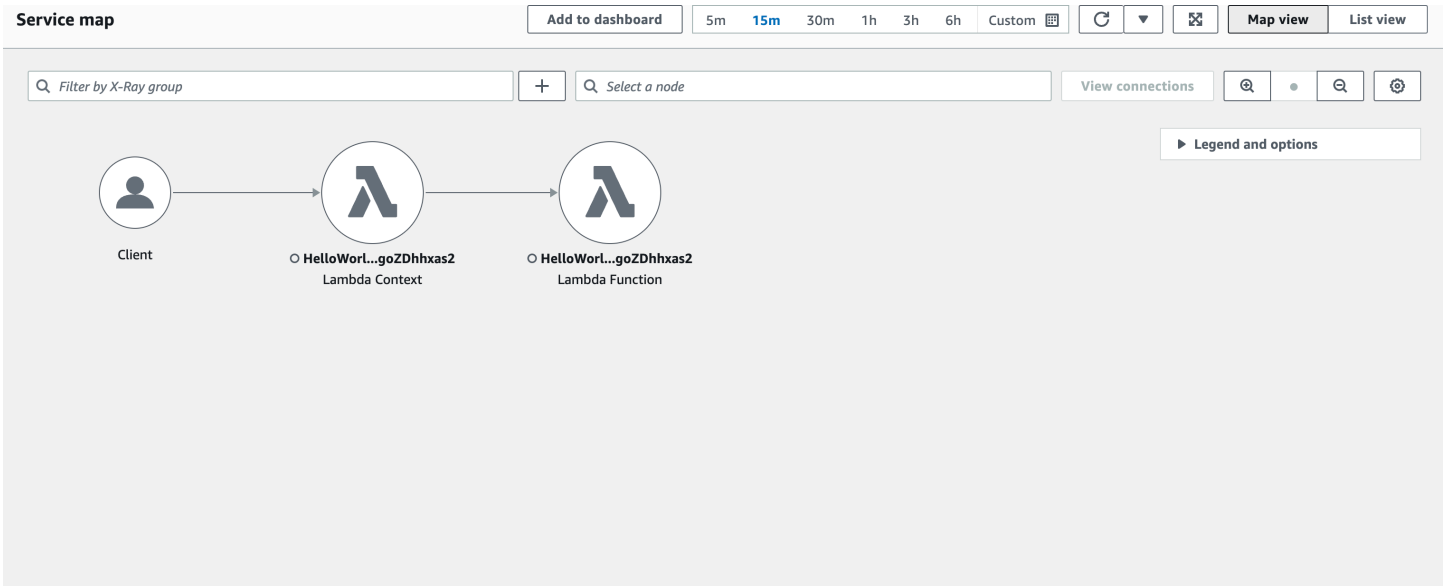
```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
cdk destroy
```

X-Ray トレースの解釈

アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。X-Ray トレースマップには、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の例はで、サンプルアプリケーションのトレースを示しています。



Python による Lambda 関数の構築

AWS Lambda で Python コードを実行できます。Lambda は、コードを実行してイベントを処理する Python 用のランタイムを提供します。コードは、管理している AWS Identity and Access Management (IAM) ロールの認証情報により、SDK for Python (Boto3) を備えた環境で実行されます。Python ランタイムに含まれている SDK バージョンの詳細については、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

Lambda は、以下の Python ランタイムをサポートしています。

Python

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Python 3.12	`python3.12`	Amazon Linux 2023			
Python 3.11	`python3.11`	Amazon Linux 2			
「Python 3.10」	`python3.10`	Amazon Linux 2			
Python 3.9	`python3.9`	Amazon Linux 2			
Python 3.8	`python3.8`	Amazon Linux 2	2024 年 10 月 14 日	2025 年 2 月 28 日	2025 年 3 月 31 日

> **Note**
>
> この表のランタイム情報は、常に更新されています。Lambda での AWS SDK の使用に関する詳細については、Serverless Land の「Lambda 関数での AWS SDK の管理」を参照してください。

Python 関数を作成するには

1. Lambda コンソールを開きます。
2. [Create function] (関数の作成) をクリックします。
3. 以下の設定を行います。
   - [Function name]: 関数名を入力します。
   - [ランタイム]: [Python 3.12] を選択します。
4. [Create function] (関数の作成) をクリックします。
5. テストイベントを設定するには、[テスト] を選択します。
6. [イベント名] で、「**test**」と入力します。
7. [変更を保存] をクリックします。
8. [テスト] を選択して関数を呼び出します。

コンソールで、`lambda_function` という名前の単一のソースファイルを含む Lambda 関数が作成されます。このファイルを編集し、組み込みのコードエディタでファイルをさらに追加することができます。変更を保存するには [保存] を選択します。コードを実行するには、[Test] (テスト) を選択します。

> **Note**
>
> Lambda コンソールでは、AWS Cloud9 を使用して、ブラウザに統合開発環境を提供します。また、AWS Cloud9 を使用して、独自の環境で Lambda 関数を開発することもできます。詳細については、AWS Cloud9 ユーザーガイドの「AWS Toolkit を使用した AWS Lambda 関数の使用」を参照してください。

> **Note**
>
> ローカル環境でアプリケーション開発を開始するには、このガイドの GitHub リポジトリで利用可能なサンプルアプリケーションの 1 つをデプロイします。
>
> Python のサンプル Lambda アプリケーション
>
> - blank-python - ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Python 関数。

Lambda 関数には CloudWatch Logs ロググループが付属しています。関数のランタイムは、各呼び出しに関する詳細を CloudWatch Logs に送信します。これは呼び出し時に、任意の関数が出力するログを中継します。関数がエラーを返す場合、Lambda はエラー形式を整え、それを呼び出し元に返します。

トピック

ランタイムに含まれる SDK バージョン

Python ランタイムに含まれる AWS SDK のバージョンは、ランタイムバージョンと AWS リージョン によって異なります。使用しているランタイムに含まれている SDK のバージョンを確認するには、次のコードを使用して Lambda 関数を作成します。

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

レスポンスの形式

Python 3.12 以降の Python ランタイムでは、関数は JSON レスポンスの一部としてユニコード文字を返します。以前の Python ランタイムは、レスポンス内の Unicode 文字のエスケープシーケンスを

返していました。例えば、Python 3.11 では、"こんにちは" のようなユニコード文字列を返すと、ユニコード文字はエスケープされ、"\u3053\u3093\u306b\u3061\u306f" が返されます。Python 3.12 ランタイムは本来の "こんにちは" を返します。

Unicode レスポンスを使用すると、Lambda レスポンスのサイズが小さくなるため、同期関数の 6 MB の最大ペイロードサイズに大きなレスポンスを簡単に収めることができます。前の例では、エスケープされたバージョンは 32 バイトですが、Unicode 文字列では 17 バイトです。

Python 3.12 にアップグレードする場合、新しい応答形式に合わせてコードを調整する必要がある場合があります。呼び出し側が Unicode のエスケープを想定している場合、戻り値関数にコードを追加して Unicode を手動でエスケープするか、Unicode の戻り値を処理するように呼び出し側を調整する必要があります。

拡張機能の正常なシャットダウン

Python 3.12 以降の Python ランタイムでは、外部拡張機能を含む関数のグレースフルシャットダウン機能が改善されています。Lambda は、実行環境をシャットダウンするときに、ランタイムに `SIGTERM` シグナルを送信してから、登録された各外部拡張機能に `SHUTDOWN` イベントを送信します。Lambda 関数で `SIGTERM` シグナルをキャッチし、その関数によって作成されたデータベース接続などのリソースをクリーンアップできます。

実行環境のライフサイクルの詳細については、「Lambda 実行環境」を参照してください。拡張機能でグレースフルシャットダウンを使用する方法の例については、「AWS Samples GitHub リポジトリ」を参照してください。

Python の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

Python で関数ハンドラーを作成するときは、次の一般的な構文が使用できます。

```
def handler_name(event, context):
    ...
    return some_value
```

命名

Lambda 関数の作成時に指定される Lambda 関数ハンドラー名は、以下から取得されます。

- Lambda ハンドラー関数が配置されているファイルの名前
- Python ハンドラー関数の名前

関数ハンドラーには任意の名前を付けることができますが、Lambda コンソールのデフォルト名は `lambda_function.lambda_handler` です。この関数ハンドラー名には、関数名 (`lambda_handler`) と、ハンドラコードが保存されているファイル (`lambda_function.py`) が反映されます。

異なるファイル名または関数ハンドラー名を使用してコンソールで関数を作成する場合は、デフォルトのハンドラー名を編集する必要があります。

関数ハンドラー名を変更するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、関数を選択します。
2. [コード] タブを選択します。
3. [ランタイム設定] ペインまでスクロールして、[編集] を選択します。
4. [ハンドラー] には、関数ハンドラーの新しい名前を入力します。
5. [Save] を選択します。

仕組み

Lambda が関数ハンドラーを呼び出すと、Lambda ランタイムが 2 つの引数を関数ハンドラーに渡します。

- 最初の引数は イベントオブジェクト です。イベントは、処理する Lambda 関数のデータを含む JSON 形式のドキュメントです。 Lambda ランタイム は、イベントをオブジェクトに変換し、それを関数コードに渡します。これは通常 Python `dict` タイプです。また `list`、`str`、`int`、`float`、または NoneType タイプを使用できます。

  イベントオブジェクトには、呼び出し元のサービスからの情報が含まれます。関数を呼び出すときは、イベントの構造とコンテンツを決定します。AWS のサービスで関数を呼び出す場合、そのイベントはサービスによって定義されます。AWS サービスからのイベントの詳細については、「他の AWS サービスからのイベントを使用した Lambda の呼び出し」を参照してください。
- 2番目の引数は コンテキストオブジェクト です。コンテキストオブジェクトは、ランタイムに Lambda によって関数に渡されます。このオブジェクトは、呼び出し、関数、およびランタイム環境に関する情報を示すメソッドおよびプロパティを提供します。

値の返し

オプションで、ハンドラーは値を返すことができます。戻り値に何が起きるかは、呼び出しの種類と呼び出したサービス に応じて変わります。以下に例を示します。

- 同期呼び出し のような `RequestResponse` 呼び出しタイプを使用すると、AWS Lambda は Python 関数の呼び出しの結果を、Lambda 関数を呼び出したクライアントに返します (呼び出しリクエストに対する HTTP レスポンスでは、JSON にシリアル化されます)。たとえば、AWS Lambda のコンソールは `RequestResponse` の呼び出しタイプを使用するため、コンソールを使用して関数を呼び出すと、コンソールに戻り値が表示されます。
- ハンドラーから `json.dumps` でシリアル化できないオブジェクトが返された場合、ランタイムはエラーを返します。
- None ステートメントを指定しなかった場合の Python 関数の暗黙の動作と同じように、ハンドラーが `return` を返した場合、ランタイムは `null` を返します。
- 「`Event`」 呼び出しタイプ (非同期呼び出し) を使用する場合、値は破棄されます。

> **Note**
>
> Python 3.9 以降のリリースでは、Lambda はエラーの応答時に呼び出しの requestId を含めます。

例

次のセクションでは、Lambda で使用できる Python 関数の例を示します。Lambda のコンソールを使用して関数を作成する場合、このセクションで関数を実行するために .zip アーカイブファイルをアタッチする必要はありません。これらの関数は、選択した Lambda ランタイムに含まれている標準の Python ライブラリを使用します。詳細については、「Lambda デプロイパッケージ」を参照してください。

メッセージの返し

以下は、`lambda_handler` と呼ばれる関数の例を示しています。この関数は、姓と名のユーザー入力を受け入れ、入力として受信したイベントからのデータを含むメッセージを返します。

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

次のイベントデータを使用して関数を呼び出すことができます。

```
{
    "first_name": "John",
    "last_name": "Smith"
}
```

レスポンスには、入力として渡されたイベントデータが表示されます。

```
{
    "message": "Hello John Smith!"
}
```

レスポンスの解析

以下は、lambda_handler と呼ばれる関数の例を示しています。この関数は、ランタイムに Lambda によって渡されたイベントデータを使用します。JSON レスポンスで返されるAWS_REGIONの 環境変数 を解析します。

```
import os
import json

def lambda_handler(event, context):
    json_region = os.environ['AWS_REGION']
    return {
        "statusCode": 200,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": json.dumps({
            "Region ": json_region
        })
    }
```

関数の呼び出し には、どのイベントデータでも使用できます。

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Lambda ランタイムは、初期化中にいくつかの環境変数を設定します。ランタイムでレスポンスで返される環境変数の詳細については、「Lambda 環境変数を使用したコードの値の設定」を参照してください。

この例の関数は、Invoke API からの正常なレスポンス (200からの) によって異なります。[API を呼び出す] ステータスの詳細については、「 Invoke Response Syntax」を参照してください。

計算の返し

以下は、lambda_handler と呼ばれる関数の例を示しています。この関数はユーザー入力を受け入れ、ユーザーに計算を返します。このサンプルの詳細については、「 aws-doc-sdk-examples GitHub リポジトリ」を参照してください。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    ...
    result = None
    action = event.get('action')
    if action == 'increment':
        result = event.get('number', 0) + 1
        logger.info('Calculated result of %s', result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {'result': result}
    return response
```

次のイベントデータを使用して 関数を呼び出す ことができます 。

```
{
    "action": "increment",
    "number": 3
}
```

Python Lambda 関数で .zip ファイルアーカイブを使用する

AWS Lambda 関数のコードは、関数のハンドラーコードと、そのコードが依存するその他のパッケージやモジュールを含む .py ファイルで構成されています。この関数コードを Lambda にデプロイするには、デプロイパッケージを使用します。このパッケージは、.zip ファイルアーカイブでもコンテナイメージでもかまいません。Python でコンテナイメージを使用する方法の詳細については、「コンテナイメージで Python Lambda 関数をデプロイする」を参照してください。

.zip ファイルのデプロイパッケージを .zip ファイルアーカイブとして作成するには、コマンドラインツール用の組み込み .zip ファイルアーカイブユーティリティ、または他の .zip ファイルユーティリティ (7zip など) を使用します。次のセクションに示す例では、Linux または macOS 環境でコマンドライン `zip` ツールを使用していることを前提としています。Windows で同じコマンドを使用するには、Windows Subsystem for Linux をインストールして、Windows 統合バージョンの Ubuntu と Bash を取得します

Lambda は POSIX ファイルアクセス許可を使用するため、.zip ファイルアーカイブを作成する前に、デプロイパッケージフォルダのアクセス許可を設定することが必要になる場合があります。

トピック

- Python でのランタイム依存関係
- 依存関係のない .zip デプロイパッケージを作成する
- 依存関係を含めて .zip デプロイパッケージを作成する
- 依存関係検索パスおよびランタイムを含むライブラリ
- __pycache__ フォルダを使用する
- ネイティブライブラリとともに .zip デプロイパッケージを作成する
- .zip ファイルを使用した Python Lambda 関数の作成と更新

Python でのランタイム依存関係

Python ランタイムを使用する Lambda 関数の場合、依存関係には任意の Python パッケージまたはモジュールを使用できます。.zip アーカイブを使用して関数をデプロイするとき、関数コードでこれらの依存関係を .zip ファイルに追加するか、Lambda レイヤー を使用できます。レイヤーは、追加のコードまたはその他のコンテンツを含むことができる個別の .zip ファイルです。Python で Lambda レイヤーの使用の詳細については、「the section called “レイヤー”」を参照してください。

Lambda Python ランタイムには、AWS SDK for Python (Boto3) とその依存関係が含まれます。独自の依存関係を追加できないデプロイシナリオでは、Lambda がランタイムに SDK を提供します。

これらのシナリオには、組み込みのコードエディタを使用してコンソールで関数を作成することや、AWS Serverless Application Model (AWS SAM) または AWS CloudFormation テンプレート内のインライン関数を使用することが含まれます。

Lambda は Python ランタイムのライブラリを定期的に更新して、最新の更新とセキュリティパッチを適用します。関数がランタイムに含まれている Boto3 SDK のバージョンを使用しているにも関わらず、デプロイパッケージに SDK の依存関係が含まれている場合、バージョンの不整合の問題が発生する可能性があります。たとえば、デプロイパッケージに SDK 依存関係 urllib3 が含まれる場合があります。Lambda がランタイムで SDK を更新すると、新しいバージョンのランタイムとデプロイパッケージの urllib3 のバージョンとの互換性の問題により、関数が失敗する可能性があります。

> **Important**
>
> 依存関係を完全に制御し、バージョンの不整合の問題を回避するには、Lambda ランタイムにそれらのバージョンが含まれていても、関数のすべての依存関係をデプロイパッケージに追加することをお勧めします。これには Boto3 SDK が含まれます。

使用しているランタイムに含まれている SDK for Python (Boto3) のバージョンを確認するには、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

AWS 責任分担モデルでは、関数のデプロイパッケージに含まれる依存関係を管理する責任があります。これには、更新とセキュリティパッチの適用が含まれます。関数のデプロイパッケージ内の依存関係を更新するには、まず新しい .zip ファイルを作成し、そのファイルを Lambda にアップロードします。詳細については、「依存関係を含めて .zip デプロイパッケージを作成する」と「.zip ファイルを使用した Python Lambda 関数の作成と更新」を参照してください。

依存関係のない .zip デプロイパッケージを作成する

関数コードに依存関係がない場合、.zip ファイルには関数のハンドラーコードを含む .py ファイルのみが含まれます。任意の zip ユーティリティを使用して、.py ファイルをルートとする .zip ファイルを作成します。.py ファイルが .zip ファイルのルートにない場合、Lambda はコードを実行できません。

.zip ファイルをデプロイして新しい Lambda 関数を作成する方法の詳細、既存の Lambda 関数を更新する方法の詳細については、「.zip ファイルを使用した Python Lambda 関数の作成と更新」を参照してください。

依存関係を含めて .zip デプロイパッケージを作成する

関数コードが追加のパッケージやモジュールに依存している場合、関数コードでこれらの依存関係を .zip ファイルに追加するか、Lambda レイヤー を使用できます。このセクションでは、依存関係を .zip デプロイパッケージに含める方法について説明します。Lambda でコードを実行するには、ハンドラーコードとすべての関数の依存関係を含む.py ファイルを.zip ファイルのルートにインストールする必要があります。

関数コードが `lambda_function.py` という名前のファイルに保存されているとします。次の CLI コマンドの例では、関数コードとその依存関係を格納している `my_deployment_package.zip` という名前の .zip ファイルを作成します。依存関係をプロジェクトディレクトリのフォルダに直接インストールすることも、Python 仮想環境を使用することもできます。

デプロイパッケージ (プロジェクトディレクトリ) を作成するには

1. `lambda_function.py` ソースコードファイルを含むプロジェクトディレクトリに移動します。この例では、ディレクトリ名は `my_function` です。

   ```
   cd my_function
   ```

2. 依存関係をインストールする「パッケージ」という名前の新しいディレクトリを作成します。

   ```
   mkdir package
   ```

   .zip デプロイパッケージの場合、Lambda では、ソースコードとその依存関係がすべて .zip ファイルのルートにあると想定されていることに注意してください。ただし、依存関係をプロジェクトディレクトリに直接インストールすると、多数の新しいファイルやフォルダが追加され、IDE 内を移動することが難しくなります。依存関係をソースコードと区別するために、ここで別の `package` ディレクトリを作成します。

3. 依存関係を `package` ディレクトリにインストールします。以下の例では、pip を使用して Python パッケージインデックスから Boto3 SDK をインストールします。関数コードに自分で作成した Python パッケージを使用している場合は、それらを `package` ディレクトリに保存します。

   ```
   pip install --target ./package boto3
   ```

4. ルートにインストール済みライブラリを含む .zip ファイルを作成します。

   ```
   cd package
   ```

```
zip -r ../my_deployment_package.zip .
```

これにより、プロジェクトディレクトリに my_deployment_package.zip ファイルが生成されます。

5. .zip ファイルのルートに Lambda_function.py ファイルを追加します。

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

.zip ファイルは、次のように関数のハンドラーコードとすべての依存関係フォルダがルートにインストールされた、フラットなディレクトリ構造である必要があります。

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

関数のハンドラーコードを含む .py ファイルが .zip ファイルのルートにない場合、Lambda はコードを実行できません。

デプロイパッケージ (仮想環境) を作成するには

1. プロジェクトディレクトリに仮想環境を作成してアクティブ化します。この例では、プロジェクトディレクトリ名は my_function です。

```
~$ cd my_function
~/my_function$ python3.12 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. pip を使用して必要なライブラリをインストールします。次の例では、Boto3 SDK をインストールします。

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. pip show を使用して、仮想環境内で pip が依存関係をインストールした場所を検索できます。

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

pip がライブラリをインストールするフォルダには、site-packages または dist-packages という名前を付けることができます。このフォルダは、lib/python3.x または lib64/python3.x ディレクトリ (python3.x は使用している Python のバージョンを表します) のどちらにあってもかまいません。

4. 仮想環境を無効化します。

```
(my_virtual_env) ~/my_function$ deactivate
```

5. pip でインストールした依存関係を含むディレクトリに移動し、インストールした依存関係をルートとする .zip ファイルをプロジェクトディレクトリに作成します。この例では、pip は依存関係を my_virtual_env/lib/python3.12/site-packages ディレクトリにインストールしています。

```
~/my_function$ cd my_virtual_env/lib/python3.12/site-packages
~/my_function/my_virtual_env/lib/python3.12/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. ハンドラーコードを含む .py ファイルがあるプロジェクトディレクトリのルートに移動し、そのファイルを .zip パッケージのルートに追加します。この例では、関数コードファイルに lambda_function.py という名前が付けられています。

```
~/my_function/my_virtual_env/lib/python3.12/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

依存関係検索パスおよびランタイムを含むライブラリ

コードで import ステートメントを使用すると、Python ランタイムはモジュールまたはパッケージが見つかるまで検索パス内のディレクトリを検索します。デフォルトでは、ランタイムが最初に検索する場所は、.zip デプロイパッケージを解凍してマウントするディレクトリ (/var/task) です。ランタイムに含まれるライブラリのバージョンをデプロイパッケージに含める場合、そのバージョンが、ランタイムに含まれるバージョンよりも優先されます。デプロイパッケージ内の依存関係も、レイヤー内の依存関係よりも優先されます。

レイヤーに依存関係を追加すると、Lambda はこれを /opt/python/lib/python3.x/site-packages (ここで python3.x は使用しているランタイムのバージョンを表します) または /opt/python に抽出します。検索パスでは、これらのディレクトリは、ランタイムに含まれるライブラリおよび pip でインストールされたライブラリを含むディレクトリ (/var/runtime および /var/lang/lib/python3.x/site-packages) よりも優先されます。このため、関数レイヤー内のライブラリは、ランタイムに含まれるバージョンよりも優先されます。

> **Note**
>
> Python 3.11 マネージドランタイムとベースイメージでは、AWS SDK とその依存関係は /var/lang/lib/python3.11/site-packages ディレクトリにインストールされます。

次のコードスニペットを追加すると、Lambda 関数の完全な検索パスを確認できます。

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> デプロイパッケージまたはレイヤーの依存関係はランタイムに含まれるライブラリよりも優先されるため、SDK を含めずに urllib3 などの SDK 依存関係をパッケージに含めると、バージョンの不整合の問題が発生する可能性があります。独自のバージョンの Boto3 依存関係をデプロイする場合は、Boto3 もデプロイパッケージ内の依存関係としてデプロイする必要があります。ランタイムに関数の依存関係のバージョンが含まれていても、それらをすべてパッケージ化することをお勧めします。

.zip パッケージ内の個別のフォルダに依存関係を追加することもできます。たとえば、Boto3 SDK のバージョンを .zip パッケージ内の common というフォルダに追加できます。.zip パッケージを解凍してマウントすると、このフォルダは /var/task ディレクトリ内に配置されます。コード内の .zip デプロイパッケージ内のフォルダの依存関係を使用するには、import from ステートメントを使用します。たとえば、.zip パッケージ内の common という名前のフォルダにある Boto3 のバージョンを使用するには、次のステートメントを使用します。

```
from common import boto3
```

__pycache__ フォルダを使用する

関数のデプロイパッケージには __pycache__ フォルダを含めないことをお勧めします。アーキテクチャやオペレーティングシステムが異なるビルドマシンでコンパイルされた Python バイトコードは、Lambda 実行環境と互換性がない場合があります。

ネイティブライブラリとともに .zip デプロイパッケージを作成する

関数で、純粋な Python パッケージとモジュールのみを使用する場合は、`pip install` コマンドを使用して任意のローカルビルドマシンに依存関係をインストールし、.zip ファイルを作成できます。NumPy や Pandas を含む一般的な Python ライブラリの多くは、純粋な Python ではなく、C または C++ で記述されたコードを含んでいます。C/C++ コードを含むライブラリをデプロイパッケージに追加するときは、パッケージを正しく構築し、Lambda 実行環境と互換性があることを確認する必要があります。

Python Package インデックス (PyPI) で入手できるほとんどのパッケージは、「ホイール」(.whl ファイル) として利用できます。.whl ファイルは、特定のオペレーティングシステムと命令セットアーキテクチャ用にプリコンパイルされたバイナリ、およびビルド済みディストリビューションを含む ZIP ファイルの一種です。デプロイパッケージに Lambda との互換性をもたせるには、Linux オペレーティングシステム用のホイールと関数の命令セットアーキテクチャをインストールします。

一部のパッケージには、ソースディストリビューションとしてしか利用できないものもあります。これらのパッケージでは、C/C++ コンポーネントを自分でコンパイルして構築する必要があります。

必要なパッケージで利用できるディストリビューションを確認するには、以下を実行します。

1. Python Package インデックスのメインページでパッケージの名前を検索します。
2. 使用するパッケージのバージョンを選択します。
3. [ファイルをダウンロード] を選択します。

ビルド済みディストリビューション (ホイール) を使用する

Lambda と互換性のあるホイールをダウンロードするには、pip `--platform` オプションを使用します。

Lambda 関数が x86_64 命令セットアーキテクチャを使用している場合は、次の `pip install` コマンドを実行して、互換性のあるホイールを `package` ディレクトリにインストールします。`--python 3.x` を、使用している Python ランタイムのバージョンに置き換えます。

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

関数が arm64 命令セットアーキテクチャを使用している場合は、次のコマンドを実行します。`--python 3.x` を、使用している Python ランタイムのバージョンに置き換えます。

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

ソースディストリビューションを使用する

パッケージをソースディストリビューションとしてしか入手できない場合は、C/C++ ライブラリを自分で構築する必要があります。パッケージに Lambda 実行環境との互換性をもたせるには、同じ Amazon Linux 2 オペレーティングシステムを使用する環境でパッケージを構築する必要があります。このためには、パッケージを Amazon EC2 Linux インスタンスで構築します。

Amazon EC2 Linux インスタンスを起動して接続する方法の詳細については、「Linux インスタンス向け Amazon EC2 ユーザーガイド」の「チュートリアル: Amazon EC2 Linux インスタンスの開始方法」を参照してください。

.zip ファイルを使用した Python Lambda 関数の作成と更新

.zip デプロイパッケージを作成すると、このパッケージを使用して新しい Lambda 関数を作成するか、既存の関数を更新できます。.zip パッケージをデプロイするには、Lambda コンソール、AWS Command Line Interface、Lambda API を使用します。AWS Serverless Application Model (AWS SAM) および AWS CloudFormation を使用して、Lambda 関数を作成および更新することもできます。

Lambda の .zip デプロイパッケージの最大サイズは 250 MB (解凍) です。この制限は、Lambda レイヤーを含む、更新するすべてのファイルの合計サイズに適用されることに注意してください。

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

コンソールを使用して .zip ファイルの関数を作成、更新する

新しい関数を作成するには、まずコンソールで関数を作成し、次に .zip アーカイブをアップロードする必要があります。既存の関数を更新するには、その関数のページを開き、同じ手順に従って更新した .zip ファイルを追加します。

.zip ファイルが 50 MB 未満の場合は、ローカルマシンから直接ファイルをアップロードして関数を作成または更新できます。50 MB を超える .zip ファイルの場合は、まず Amazon S3 バケットにパッケージをアップロードする必要があります。AWS Management Console を使用して Amazon S3 バケットにファイルをアップロードする手順については、「Amazon S3 の開始方法」を参照してください。AWS CLI を使用してファイルをアップロードするには、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [一から作成] を選択します。

3. [基本的な情報] で、以下を行います。

   a. [関数名] に、関数名を入力します。

   b. [ランタイム] で、使用するランタイムを選択します。

   c. (オプション) [アーキテクチャ]で、関数の命令セットアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用の .zip デプロイパッケージと選択した命令セットのアーキテクチャに互換性があることを確認してください。

4. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。新しい [実行ロール] を作成することも、既存のロールを使用することもできます。

5. [関数の作成]を選択します。Lambda は、選択したランタイムを使用して基本的な「Hello world」関数を作成します。

ローカルマシンから zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、.zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [.zip ファイル] をクリックします。
5. .zip ファイルをアップロードするには、次の操作を行います。

   a. [アップロード] をクリックし、ファイルセレクターで .zip ファイルを選択します。

   b. [開く] をクリックします。

   c. [保存] をクリックします。

Amazon S3 バケットから .zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、新しい .zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [Amazon S3 ロケーション] を選択します。
5. .zip ファイルの Amazon S3 リンク URL を貼り付けて、[保存] を選択します。

コンソールコードエディタを使用して .zip ファイル関数を更新する

.zip デプロイパッケージを使用する一部の関数では、Lambda コンソールの組み込みコードエディタを使用して、関数コードを直接更新できます。この機能を使用するには、関数が次の基準を満たしている必要があります。

- 関数が、インタープリター言語ランタイムのいずれか (Python、Node.js、Ruby) を使用する必要があります。
- 関数のデプロイパッケージが 3 MB 未満である必要があります。

コンテナイメージデプロイパッケージを含む関数の関数コードは、コンソールで直接編集することはできません。

コンソールのコードエディタを使用して関数コードを更新するには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインでソースコードファイルを選択し、統合コードエディタで編集します。
4. コードの編集が終了したら、[デプロイ] を選択して変更を保存し、関数を更新します。

AWS CLI を使用して .zip ファイルで関数を作成、更新する

AWS CLI を使用して新しい関数を作成したり、.zip ファイルを使用して既存の関数を更新したりできます。create-function コマンドと update-function-code を使用して、.zip パッケージをデプロイします。.zip ファイルが 50 MB 未満の場合は、ローカルビルドマシン上のファイルの場所から .zip パッケージをアップロードできます。サイズの大きいファイルの場合は、Amazon S3 バケットから .zip パッケージをアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> AWS CLI を使用して Amazon S3 バケットから .zip ファイルをアップロードする場合、このバケットは関数と同じ AWS リージョン に配置する必要があります。

AWS CLI を含む .zip ファイルを使用して新しい関数を作成するには、以下を指定する必要があります。

- 関数の名前 (--function-name)
- 関数のランタイム (--runtime)
- 関数の実行ロール (--role) の Amazon リソースネーム (ARN)
- 関数コード内のハンドラーメソッド (--handler) の名前

.zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --code オプションを使用します。S3ObjectVersion パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda create-function --function-name myFunction \
--runtime python3.12 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

CLI を使用して既存の関数を更新するには、--function-name パラメータを使用して関数の名前を指定します。関数コードの更新に使用する .zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --s3-bucket および --s3-key オプションを使用します。--s3-object-version パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API を使用して .zip ファイルで関数を作成、更新する

.zip ファイルアーカイブを使用して関数を作成および更新するには、以下の API オペレーションを使用します。

- CreateFunction
- UpdateFunctionCode

AWS SAM を使用して .zip ファイルで関数を作成、更新する

AWS Serverless Application Model (AWS SAM) は、AWS のサーバーレスアプリケーションの構築と実行のプロセスを合理化するのに役立つツールキットです。YAML または JSON テンプレートでアプリケーションのリソースを定義し、AWS SAM コマンドラインインターフェイス (AWS SAM CLI) を使用して、アプリケーションを構築、パッケージ化、デプロイします。AWS SAM テンプレートから Lambda 関数を構築すると、AWS SAM は関数コードと指定した任意の依存関係を含む .zip デプロイパッケージまたはコンテナイメージを自動的に作成します。AWS SAM を使用して Lambda 関数を構築およびデプロイする方法の詳細については、「AWS Serverless Application Model 開発者ガイドの」の「AWS SAM の開始方法」を参照してください。

AWS SAM を使用して、既存の .zip ファイルアーカイブを使用する Lambda 関数を作成できます。AWS SAM を使用して Lambda 関数を作成するには、.zip ファイルを Amazon S3 バケットまたはビルドマシンのローカルフォルダに保存します。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS SAM テンプレートでは、Lambda 関数は `AWS::Serverless::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `CodeUri` - 関数コードの Amazon S3 URI、ローカルフォルダへのパス、または FunctionCode オブジェクトに設定
- `Runtime` - 選択したランタイムに設定

AWS SAM では、.zip ファイルが 50 MB を超える場合、この .zip ファイルを最初に Amazon S3 バケットにアップロードする必要はありません。AWS SAM では、ローカルビルドマシン上の場所から、最大許容サイズ 250 MB (解凍) の .zip パッケージをアップロードできます。

AWS SAM で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS SAM 開発者ガイド」の「AWS::Serverless::Function」を参照してください。

AWS CloudFormation を使用して .zip ファイルで関数を作成、更新する

AWS CloudFormation を使用して、.zip ファイルアーカイブを使用する Lambda 関数を作成できます。.zip ファイルから Lambda 関数を作成するには、最初にファイルを Amazon S3 バケットにアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

Node.js と Python のランタイムでは、AWS CloudFormation テンプレートにインラインソースコードを提供することもできます。AWS CloudFormation は、関数を構築するときにコードを含む .zip ファイルを作成します。

既存の .zip ファイルを使用する

AWS CloudFormation テンプレートでは、Lambda 関数は `AWS::Lambda::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `Code` - `S3Bucket` および `S3Key` フィールドに Amazon S3 バケット名と .zip ファイル名を入力
- `Runtime` - 選択したランタイムに設定

インラインコードから .zip ファイルを作成する

Python で記述された単純な関数や Node.js インラインを AWS CloudFormation テンプレートで宣言できます。コードは YAML または JSON に埋め込まれているため、デプロイパッケージに外部依存関係を追加することはできません。つまり、関数はランタイムに含まれている AWS SDK のバージョンを使用する必要があります。特定の文字を エスケープする必要があるなどのテンプレートの要件も、IDE の構文チェックやコード補完機能の使用を難しくします。つまり、テンプレートに追加のテストが必要な場合があります。このような制限があるため、関数をインラインで宣言することは、頻繁に変更しない非常に単純なコードに最適です。

Node.js および Python ランタイムのインラインコードから .zip ファイルを作成するには、テンプレートの `AWS::Lambda::Function` リソースに次のプロパティを設定します。

- `PackageType` が `Zip` に設定されている場合
- `Code` - `ZipFile` フィールドに関数コードを入力します。
- `Runtime` - 選択したランタイムに設定

AWS CloudFormation が生成する .zip ファイルは、4 MB を超えることはできません。AWS CloudFormation で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::Lambda::Function」を参照してください。

コンテナイメージで Python Lambda 関数をデプロイする

Python Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- Python の AWS ベースイメージを使用する

  AWS ベースイメージには、言語ランタイム、Lambda と関数コード間のやり取りを管理するランタイムインターフェースクライアント、ローカルテスト用のランタイムインターフェイスエミュレーターがプリロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、Python のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することができます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、Python のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

トピック

- Python の AWS ベースイメージ

- Python の AWS ベースイメージを使用する
- ランタイムインターフェイスクライアントで代替ベースイメージを使用する

Python の AWS ベースイメージ

AWS は、Python 用の次のベースイメージを提供します。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
3.12	Python 3.12	Amazon Linux 2023	GitHub の Python 3.12 用 Dockerfile	
3.11	Python 3.11	Amazon Linux 2	GitHub の Python 3.11 用 Dockerfile	
3.10	「Python 3.10」	Amazon Linux 2	GitHub の Dockerfile for Python 3.10	
3.9	Python 3.9	Amazon Linux 2	GitHub の Dockerfile for Python 3.9	
3.8	Python 3.8	Amazon Linux 2	GitHub の Python 3.8 用 Dockerfile	2024 年 10 月 14 日

Amazon ECR リポジトリ: gallery.ecr.aws/lambda/python

Python 3.12 以降のベースイメージは、Amazon Linux 2023 の最小コンテナイメージに基づいています。Python 3.8 ~ 3.11 のベースイメージは、Amazon Linux 2 のイメージに基づいています。AL2023 ベースのイメージには、デプロイのフットプリントが小さいことや、`glibc` などのライブラリのバージョンが更新されていることなど、Amazon Linux 2 に比べていくつかの利点があります。

AL2023 ベースのイメージでは、Amazon Linux 2 のデフォルトのパッケージマネージャである `yum` の代わりに `microdnf` (`dnf` としてシンボリックリンク) がパッケージマネージャとして使用されています。`microdnf` は `dnf` のスタンドアロン実装です。AL2023 ベースのイメージに含まれるパッケージのリストについては、「Comparing packages installed on Amazon Linux 2023 Container

Images」の「Minimal Container」列を参照してください。AL2023 と Amazon Linux 2 の違いの詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。

> **Note**
>
> AWS Serverless Application Model (AWS SAM) を含む AL2023 ベースのイメージをローカルで実行するには、Docker バージョン 20.10.10 以降を使用する必要があります。

ベースイメージ内の依存関係の検索パス

コードで `import` ステートメントを使用すると、Python ランタイムはモジュールまたはパッケージが見つかるまで検索パス内のディレクトリを検索します。デフォルトでは、ランタイムは `{LAMBDA_TASK_ROOT}` ディレクトリを先に検索します。ランタイムに含まれるライブラリのバージョンをイメージに含める場合、そのバージョンが、ランタイムに含まれるバージョンよりも優先されます。

検索パスの他のステップは、使用している Python 用 Lambda ベースイメージのバージョンによって次のように異なります。

- Python 3.11 以降: ランタイムに含まれるライブラリと pip でインストールされるライブラリは `/var/lang/lib/python3.11/site-packages` ディレクトリにインストールされます。このディレクトリは、検索パス内で `/var/runtime` よりも優先されます。pip を使用して新しいバージョンをインストールすることで、SDK をオーバーライドできます。pip を使用して、ランタイムに含まれる SDK とその依存関係が、インストールする任意のパッケージと互換性があることを確認できます。
- Python 3.8-3.10: ランタイムに含まれるライブラリは `/var/runtime` ディレクトリにインストールされます。pip でインストールされるライブラリは `/var/lang/lib/python3.x/site-packages` ディレクトリにインストールされます。`/var/runtime` ディレクトリは検索パス内で `/var/lang/lib/python3.x/site-packages` より優先されます。

次のコードスニペットを追加すると、Lambda 関数の完全な検索パスを確認できます。

```
import sys

search_path = sys.path
print(search_path)
```

Python の AWS ベースイメージを使用する

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker (Python 3.12 以降のベースイメージの最小バージョンは 20.10.10)
- Python

ベースイメージからイメージを作成する

Python の AWS ベースイメージからコンテナイメージを作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

   ```
   mkdir example
   cd example
   ```

2. `lambda_function.py` という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

   Example Python 関数

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. `requirements.txt` という名前の新しいファイルを作成します。前のステップのサンプル関数コードを使用している場合、依存関係がないためファイルを空のままにしておくことができます。それ以外の場合は、必要なライブラリをそれぞれリストしてください。たとえば、関数で AWS SDK for Python (Boto3) を使用している場合、`requirements.txt` は次のようになります。

   Example requirements.txt

   ```
   boto3
   ```

4. 次の設定で新しい Dockerfile を作成します。

- FROM プロパティを「ベースイメージの URI」に設定します。
- COPY コマンドを使用し、関数コードおよびランタイムの依存関係を {LAMBDA_TASK_ROOT} (Lambda 定義の環境変数) にコピーします。
- CMD 引数を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12

# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

1. docker run コマンドを使用して、Docker イメージを起動します。この例では、docker-image はイメージ名、test はタグです。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、--platform linux/*amd64* の代わりに --platform linux/*arm64* オプションを使用してください。

2. 新しいターミナルウィンドウから、イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の curl コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の Invoke-WebRequest コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - `111122223333` を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、`update-function-code` コマンドを使用する必要があります。

ランタイムインターフェイスクライアントで代替ベースイメージを使用する

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間の相互作用を管理する Lambda Runtime API を拡張します。

pip パッケージマネージャーを使用して、Python 用のランタイムインターフェイスクライアントをインストールします。

```
pip install awslambdaric
```

Python ランタイムインターフェイスクライアント を GitHub からダウンロードすることもできます。

次の例は、非 AWS ベースイメージを使用して Python 用のコンテナイメージを構築する方法を示しています。サンプルの Dockerfile は公式の Python ベースイメージを使用しています。Dockerfile には、Python 用のランタイムインターフェイスクライアントが含まれます。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker
- Python

代替ベースイメージからイメージを作成する

非 AWS ベースイメージからコンテナイメージを作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir example
cd example
```

2. `lambda_function.py` という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

   Example Python 関数

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. `requirements.txt` という名前の新しいファイルを作成します。前のステップのサンプル関数コードを使用している場合、依存関係がないためファイルを空のままにしておくことができます。それ以外の場合は、必要なライブラリをそれぞれリストしてください。たとえば、関数で AWS SDK for Python (Boto3) を使用している場合、`requirements.txt` は次のようになります。

   Example requirements.txt

```
boto3
```

4. 新しい Dockerfile を作成します。次の Dockerfile は、AWS ベースイメージの代わりに公式の Python ベースイメージを使用しています。Dockerfile には、イメージに Lambda との互換性を持たせるランタイムインターフェイスクライアントが含まれています。次の Dockerfile の例では、「マルチステージビルド」が使用されます。

   - `FROM` プロパティにベースイメージを設定します。
   - `ENTRYPOINT` を、Docker コンテナの起動時に実行させるモジュールに設定します。この場合、モジュールはランタイムインターフェイスクライアントです。
   - `CMD` を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する `--platform linux/amd64` オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに `--platform linux/arm64` オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。エミュレーターはイメージに組み込むことも、次の手順を使用してローカルマシンにインストールすることもできます。

ローカルマシンにランタイムインターフェイスエミュレーターをインストールして実行するには

1. プロジェクトディレクトリから次のコマンドを実行して、GitHub からランタイムインターフェイスエミュレーター (x86-64 アーキテクチャ) をダウンロードし、ローカルマシンにインストールします。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   arm64 エミュレータをインストールするには、前のコマンドの GitHub リポジトリ URL を次のように置き換えます。

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   ```

```
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

arm64 エミュレーターをインストールするには、$downloadLink を次のように置き換えます。

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。

   - docker-image はイメージ名、test はタグです。
   - /usr/local/bin/python -m awslambdaric lambda_function.handler は ENTRYPOINT で、その後に Dockerfile の CMD が続きます。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/local/bin/python -m awslambdaric lambda_function.handler
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/local/bin/python -m awslambdaric lambda_function.handler
   ```

   このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、`--platform linux/amd64` の代わりに `--platform linux/arm64` オプションを使用してください。

3. イベントをローカルエンドポイントにポストします。

   Linux/macOS

   Linux および macOS では、次の `curl` コマンドを実行します。

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   PowerShell で次の `Invoke-WebRequest` コマンドを実行します。

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. コンテナ ID を取得します。

```
docker ps
```

5. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - --region 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
```

```
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   例：

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず `:latest` を含めてください。

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。`ImageUri` には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず `:latest` を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

Alpine ベースイメージから Python イメージを作成する方法の例については、AWS ブログの「Lambda のコンテナイメージのサポート」を参照してください。

Python Lambda 関数にレイヤーを使用する

Lambda レイヤー は、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。レイヤーの作成には、次の 3 つの一般的な手順が含まれます。

1. レイヤーコンテンツのパッケージ化。これは、関数で使用する依存関係を含む .zip ファイルアーカイブを作成することを意味します。
2. Lambda でレイヤーを作成します。
3. レイヤーを関数に追加します。

このトピックには、外部ライブラリの依存関係を持つ Python Lambda レイヤーを適切にパッケージ化して作成する方法に関する手順とガイダンスが含まれています。

トピック

前提条件

このセクションの手順を完了するには、次の事項が必要です。

- 「Python 3.11」および「pip」のパッケージインストーラ
- AWS Command Line Interface (AWS CLI) バージョン 2

このトピック全体では、awsdocs GitHub リポジトリの「`layer-python`」サンプルアプリケーションを参照します。このアプリケーションには、依存関係をダウンロードしてレイヤーを生成するスクリプトが含まれています。アプリケーションは対応する関数も含んでおり、レイヤーの依存関係を使用します。レイヤーを作成したら、対応する関数をデプロイして呼び出し、すべてが正しく動作する

ことを確認できます。関数に Python 3.11 ランタイムを使用するため、レイヤーも Python 3.11 と互換性がある必要があります。

layer-python サンプルアプリケーションには、2 つの例があります。

- 最初の例では、「requests」ライブラリを Lambda レイヤーにパッケージ化します。layer/ ディレクトリには、レイヤーを生成するスクリプトが含まれています。function/ ディレクトリには、レイヤーが動作するテストを支援するサンプル関数が含まれています。このチュートリアルの大部分では、このレイヤーを作成してパッケージ化する方法について説明します。
- 2 番目の例では、「numpy」ライブラリを Lambda レイヤーにパッケージ化します。layer-numpy/ ディレクトリには、レイヤーを生成するスクリプトが含まれています。function-numpy/ ディレクトリには、レイヤーが動作するテストを支援するサンプル関数が含まれています。このレイヤーを作成およびパッケージ化する方法の例については、「the section called "manylinux ホイールディストリビューションを使用する"」を参照してください。

Python レイヤーと Amazon Linux の互換性

レイヤーを作成する最初のステップは、すべてのレイヤーコンテンツを .zip ファイルアーカイブにバンドルすることです。Lambda 関数は Amazon Linux 上で実行されるため、レイヤーコンテンツは Linux 環境でコンパイルおよびビルドできる必要があります。

Python では、ほとんどのパッケージはソースディストリビューションに加え、「ホイール」(.whl ファイル) として利用できます。各ホイールは、Python バージョン、オペレーティングシステム、機械語命令セットの特定の組み合わせをサポートするビルド済みディストリビューションの一種です。

ホイールは、レイヤーが Amazon Linux と互換性があることを確認するために便利です。依存関係をダウンロードするとき、可能であればユニバーサルホイールをダウンロードしてください。(デフォルトでは、ユニバーサルホイールが利用可能であれば、pip がインストールします。) ユニバーサルホイールはプラットフォームタグとして any が含まれており、Amazon Linux を含むすべてのプラットフォームと互換性があることを示しています。

次の例では、requests ライブラリを Lambda レイヤーにパッケージ化します。requests ライブラリは、ユニバーサルホイールとして利用できるパッケージの一例です。

すべての Python パッケージはユニバーサルホイールとして配布されているわけではありません。例えば、「numpy」には複数のホイールディストリビューションがあり、それぞれが異なるプラットフォームのセットをサポートしています。このようなパッケージには、manylinux ディストリビューションをダウンロードして Amazon Linux との互換性を確認してください。このような Layer

をパッケージ化する方法の詳細な手順については、「the section called “manylinux ホイールディストリビューションを使用する”」を参照してください。

まれに、Python パッケージがホイールとして利用できない場合があります。「ソースディストリビューション」(sdist) のみが存在する場合、「Amazon Linux 2023 ベースコンテナイメージ」に基づく「Docker」環境に依存関係をインストールしてパッケージ化することをお勧めします。C/C++ などの他の言語で記述された独自のカスタムライブラリを含める場合も、このアプローチをお勧めします。このアプローチは Docker の Lambda 実行環境を模倣しており、Python 以外のパッケージの依存関係が Amazon Linux と互換性があることを保証します。

Python ランタイムのレイヤーパス

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の /opt ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には /opt ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

- `python`
- `python/lib/python3.x/site-packages`

例えば、このチュートリアルで作成するレイヤーの .zip ファイルは、次のディレクトリ構造になっています。

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

「requests」ライブラリは python/lib/python3.11/site-packages ディレクトリに正しく配置されています。これにより、Lambda は関数の呼び出し中にライブラリを見つけられるようになります。

レイヤーコンテンツのパッケージ化

この例では、Python `requests` ライブラリをレイヤーの .zip ファイルにパッケージ化します。レイヤーコンテンツをインストールしてパッケージ化するには、次の手順を実行します。

レイヤーコンテンツをインストールしてパッケージ化する方法

1. `sample-apps/layer-python` ディレクトリで必要なサンプルコードを含む「aws-lambda-developer-guide GitHub リポジトリ」を複製します。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. `layer-python` サンプルアプリの `layer` ディレクトリに移動します。このディレクトリには、レイヤーを適切に作成してパッケージ化するために使用するスクリプトが含まれています。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-python/layer
   ```

3. `requirements.txt` ファイルを検証する このファイルは、レイヤー (つまり `requests` ライブラリ) に含める依存関係を定義します。このファイルを更新し、独自のレイヤーに含める依存関係を含めることができます。

   Example requirements.txt

   ```
   requests==2.31.0
   ```

4. 両方のスクリプトを実行する許可があることを確認してください。

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 次のコマンドを使用して「`1-install.sh`」スクリプトを実行します。

   ```
   ./1-install.sh
   ```

   このスクリプトは venv を使用し、`create_layer` という名前の Python 仮想環境を作成します。次に、必要な依存関係をすべて `create_layer/lib/python3.11/site-packages` ディレクトリにインストールします。

   Example 1-install.sh

   ```
   python3.11 -m venv create_layer
   ```

```
source create_layer/bin/activate
pip install -r requirements.txt
```

6. 次のコマンドを使用して「2-package.sh」スクリプトを実行します。

```
./2-package.sh
```

このスクリプトは、create_layer/lib ディレクトリから python という名前の新しいディレクトリに内容をコピーします。次に、python ディレクトリの内容を layer_content.zip という名前のファイルに圧縮します。これはレイヤーの .zip ファイルです。ファイルを解凍し、the section called “Python ランタイムのレイヤーパス” セクションで示されている正しいファイル構造が含まれていることを確認できます。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

レイヤーを作成する

このセクションでは、前のセクションで生成した「layer_content.zip」ファイルを取得し、Lambda レイヤーとしてアップロードします。AWS Command Line Interface (AWS CLI) を介して AWS Management Consoleまたは Lambda API を使用してレイヤーをアップロードできます。レイヤーの .zip ファイルをアップロードするとき、次の「PublishLayerVersion」 AWS CLI コマンドで python3.11 を互換性のあるランタイムとして指定し、arm64 を互換性のあるアーキテクチャとして指定します。

```
aws lambda publish-layer-version --layer-name python-requests-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "arm64"
```

レスポンスでは、arn:aws:lambda:us-east-1:*123456789012*:layer:python-requests-layer:1 に似ている LayerVersionArn に注目してください。この Amazon リソースネーム (ARN) は、このチュートリアルの次の手順で、レイヤーを関数に追加するときに必要になります。

レイヤーを関数に追加する

このセクションでは、関数コードで `requests` ライブラリを使用するサンプル Lambda 関数をデプロイし、Layer をアタッチします。関数をデプロイするには、the section called “実行ロール (関数に対する、他のリソースにアクセスするためのアクセス許可)” が必要です。既存の実行ロールをお持ちでない場合、折りたたみ可能なセクションの手順を実行してください。それ以外の場合、次の手順に進んで関数をデプロイします。

(オプション) 実行ロールを作成する

実行ロールを作成する

1. IAM コンソールの [ロールページ] を開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。

   - 信頼されたエンティティ – Lambda。
   - アクセス許可 – AWSLambdaBasicExecutionRole。
   - ロール名 – **`lambda-role`**。

   AWSLambdaBasicExecutionRole ポリシーには、ログを CloudWatch Logs に書き込むために関数が必要とするアクセス許可があります。

Lambda 関数をデプロイする方法

1. `function/` ディレクトリに移動します。現在、`layer/` ディレクトリにいる場合、次のコマンドを実行します。

   ```
   cd ../function
   ```

2. 「関数コード」を確認します。この関数は `requests` ライブラリをインポートし、簡単な HTTP GET リクエストを行ない、ステータスコードおよび本文を返します。

   ```
   import requests

   def lambda_handler(event, context):
       print(f"Version of requests library: {requests.__version__}")
       request = requests.get('https://api.github.com/')
   ```

```
    return {
        'statusCode': request.status_code,
        'body': request.text
    }
```

3. 次のコマンドを使用し、.zip ファイルのデプロイパッケージを作成します。

```
zip my_deployment_package.zip lambda_function.py
```

4. 関数をデプロイします。次の AWS CLI コマンドで、`--role` パラメータを実行ロール ARN に置き換えます。

```
aws lambda create-function --function-name python_function_with_layer \
    --runtime python3.11 \
    --architectures "arm64" \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

(オプション) レイヤーをアタッチせずに関数を呼び出す

この時点では、レイヤーをアタッチする前に関数の呼び出しを試みることもできます。これを試みると、関数が `requests` パッケージを参照できないため、インポートエラーが発生します。関数を呼び出すには、次の AWS CLI コマンドを使用します。

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

特定のエラーを表示するには、出力 `response.json` ファイルを開きます。次のエラーメッセージで `ImportModuleError` が表示されます。

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'requests'"
```

次に、レイヤーを関数にアタッチします。次の AWS CLI コマンドで、`--layers` パラメーターを先にメモしたレイヤーバージョン ARN に置き換えます。

```
aws lambda update-function-configuration --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

最後に、次の AWS CLI コマンドを使用して関数の呼び出しを試みます。

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

出力 `response.json` ファイルには、レスポンスに関する詳細が含まれています。

(オプション) リソースをクリーンアップする

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda レイヤーを削除する方法

1. Lambda コンソールの [Layers (レイヤー)] ページを開きます。
2. 作成したレイヤーを選択します。
3. [削除] を選択したら、[削除] を再度選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。

2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

manylinux ホイールディストリビューションを使用する

依存関係として含めるパッケージにユニバーサルホイールがない (具体的には、プラットフォームタグとして any がない) 場合があります。この場合は、代わりに manylinux をサポートするホイールをダウンロードしてください。これにより、レイヤーライブラリが Amazon Linux と互換性があることが保証されます。

「numpy」は、ユニバーサルホイールを備えていないパッケージの 1 つです。numpy パッケージをレイヤーに含める場合、次のサンプルステップを実行し、レイヤーを適切にインストールしてパッケージ化できます。

レイヤーコンテンツをインストールしてパッケージ化する方法

1. sample-apps/layer-python ディレクトリで必要なサンプルコードを含む「aws-lambda-developer-guide GitHub リポジトリ」を複製します。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. layer-python サンプルアプリの layer-numpy ディレクトリに移動します。このディレクトリには、レイヤーを適切に作成してパッケージ化するために使用するスクリプトが含まれています。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-python/layer-numpy
   ```

3. requirements.txt ファイルを検証する このファイルは、レイヤー (つまり、numpy ライブラリ) に含める依存関係を定義します。ここでは、Python 3.11、Amazon Linux、x86_64 指示セットと互換性のある manylinux ホイールディストリビューションの URL を指定します。

   Example requirements.txt

   ```
   https://files.pythonhosted.org/packages/3a/d0/
   edc009c27b406c4f9cbc79274d6e46d634d139075492ad055e3d68445925/numpy-1.26.4-cp311-
   cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
   ```

4. 両方のスクリプトを実行する許可があることを確認してください。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 次のコマンドを使用して「1-install.sh」スクリプトを実行します。

```
./1-install.sh
```

このスクリプトは venv を使用し、create_layer という名前の Python 仮想環境を作成します。次に、必要な依存関係をすべて create_layer/lib/python3.11/site-packages ディレクトリにインストールします。--platform タグを manylinux2014_x86_64 として指定する必要があるため、この場合の pip コマンドは異なります。ローカルマシンが MacOS または Windows を使用している場合でも、正しい manylinux ホイールをインストールするように pip に指示します。

Example 1-install.sh

```
python3.11 -m venv create_layer
source create_layer/bin/activate
pip install -r requirements.txt --platform=manylinux2014_x86_64 --only-binary=:all:
 --target ./create_layer/lib/python3.11/site-packages
```

6. 次のコマンドを使用して「2-package.sh」スクリプトを実行します。

```
./2-package.sh
```

このスクリプトは、create_layer/lib ディレクトリから python という名前の新しいディレクトリに内容をコピーします。次に、python ディレクトリの内容を layer_content.zip という名前のファイルに圧縮します。これはレイヤーの .zip ファイルです。ファイルを解凍し、the section called “Python ランタイムのレイヤーパス” セクションに示されている正しいファイル構造が含まれていることを確認できます。

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

このレイヤーを Lambda にアップロードするには、次の「PublishLayerVersion」 AWS CLI コマンドを使用します。

```
aws lambda publish-layer-version --layer-name python-numpy-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "x86_64"
```

レスポンスでは、arn:aws:lambda:us-east-1:*123456789012*:layer:python-numpy-layer:1 に似ている LayerVersionArn に注目してください。レイヤーが期待どおりに機能することを確認するには、Lambda 関数を function-numpy ディレクトリにデプロイします。

Lambda 関数をデプロイする方法

1. function-numpy/ ディレクトリに移動します。現在、layer-numpy/ ディレクトリにいる場合、次のコマンドを実行します。

   ```
   cd ../function-numpy
   ```

2. 「関数コード」を確認します。関数は numpy ライブラリをインポートし、単純な numpy 配列を作成してダミーのステータスコードおよび本文を返します。

   ```
   import json
   import numpy as np

   def lambda_handler(event, context):

       x = np.arange(15, dtype=np.int64).reshape(3, 5)
       print(x)

       return {
           'statusCode': 200,
           'body': json.dumps('Hello from Lambda!')
       }
   ```

3. 次のコマンドを使用し、.zip ファイルのデプロイパッケージを作成します。

   ```
   zip my_deployment_package.zip lambda_function.py
   ```

4. 関数をデプロイします。次の AWS CLI コマンドで、--role パラメータを実行ロール ARN に置き換えます。

```
aws lambda create-function --function-name python_function_with_numpy \
    --runtime python3.11 \
    --handler lambda_function.lambda_handler \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://my_deployment_package.zip
```

(オプション) レイヤーをアタッチせずに関数を呼び出す

オプションとして、レイヤーをアタッチする前に関数の呼び出しを試みることもできます。これを試みると、関数が numpy パッケージを参照できないため、インポートエラーが発生します。関数を呼び出すには、次の AWS CLI コマンドを使用します。

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

特定のエラーを表示するには、出力 `response.json` ファイルを開きます。次のエラーメッセージで `ImportModuleError` が表示されます。

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'numpy'"
```

次に、レイヤーを関数にアタッチします。次の AWS CLI コマンドでは、`--layers` パラメータを以下のレイヤーバージョン ARN に置き換えます。

```
aws lambda update-function-configuration --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1"
```

最後に、次の AWS CLI コマンドを使用して関数の呼び出しを試みます。

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

関数ログを調べ、コードが numpy 配列を標準出力に出力していることを確認できます。

Python の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティを提供します。コンテキストオブジェクトが関数ハンドラーに渡される方法の詳細については、「Python の Lambda 関数ハンドラーの定義」をご参照ください。

context メソッド

- `get_remaining_time_in_millis` — 実行がタイムアウトするまでの残り時間をミリ秒で返します。

context プロパティ

- `function_name` - Lambda 関数の名前。
- `function_version` - 関数のバージョン。
- `invoked_function_arn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `memory_limit_in_mb` - 関数に割り当てられたメモリの量。
- `aws_request_id` - 呼び出しリクエストの ID。
- `log_group_name` - 関数のロググループ。
- `log_stream_name` — 関数インスタンスのログストリーム。
- `identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
  - `cognito_identity_id` - 認証された Amazon Cognito ID
  - `cognito_identity_pool_id` — 呼び出しを承認した Amazon Cognito ID プール。
- `client_context` — (モバイルアプリ) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `custom` - モバイルクライアントアプリケーションが設定したカスタム値の `dict`。

- `env` - `dict` SDK が提供した環境情報の AWS。

以下の例は、コンテキスト情報を記録するハンドラ関数を示します。

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

上記に一覧表示するオプションに加えて、AWS 用 AWS Lambda での Python コードの作成 X-Ray SDK を使用して、重要なコードパスの識別、パフォーマンスのトレースおよび分析のためにデータをキャプチャすることもできます。

Python の AWS Lambda 関数ログ作成

AWS Lambda は、Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログエントリを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda のランタイム環境は、各呼び出しの詳細や、関数のコードからのその他の出力をログストリームに送信します。CloudWatch Logs の詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

関数コードからログを出力するには、組み込み `logging` モジュールを使用できます。より詳細なエントリを行うため、`stdout` または `stderr` に書き込みを行う任意のログ記録ライブラリを使用できます。

ログへの印刷

基本的な出力をログに送信するには、関数の `print` メソッドを使用できます。次の例では、Amazon CloudWatch Logs のロググループとストリーム、イベントオブジェクトの値をログに記録します。

関数が Python `print` ステートメントを使用してログを出力する場合、Lambda はログ出力をプレーンテキスト形式でのみ CloudWatch Logs に送信できることに注意してください。構造化 JSON でログをキャプチャするには、サポートされているログ記録ライブラリを使用する必要があります。詳細については、「the section called “Python での Lambda の高度なログ記録コントロールの使用”」を参照してください。

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example ログ出力

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2023/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
```

```
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Python ランタイムは、呼び出しごとに START、END、および REPORT の各行を記録します。REPORT 行には、次のデータが含まれています。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- サンプリング済み - トレースされたリクエストの場合、サンプリング結果。

ログ記録ライブラリを使用する

より詳細なログのためには、標準ライブラリ内のログ記録モジュール、または `stdout` や `stderr` への書き込みを行うサードパーティーのログ記録ライブラリを使用します。

サポートされている Python ランタイムでは、標準 `logging` モジュールを使用して作成されたログをプレーンテキストまたは JSON のいずれかでキャプチャすることを選択できます。詳細については、「the section called “Python での Lambda の高度なログ記録コントロールの使用”」を参照してください。

現在、すべての Python ランタイムのデフォルトのログ形式はプレーンテキストです。以下の例は、標準 `logging` モジュールを使用して作成されたログ出力が CloudWatch Logs にプレーンテキストでどのようにキャプチャされるかを示しています。

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

`logger` からの出力には、ログレベル、タイムスタンプおよびリクエスト ID が含まれています。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2023-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2023/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2023-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> 関数のログ形式がプレーンテキストに設定されている場合、Python ランタイムのデフォルトのログレベル設定は WARN です。つまり、Lambda は WARN 以下のレベルのログ出力のみを CloudWatch Logs に送信します。デフォルトのログレベルを変更するには、このコード例に示すように Python `logging` `setLevel()` メソッドを使用します。関数のログ形式を JSON に設定する場合、コードのログレベルを設定するのではなく、Lambda の高度なログ制御機能を使用して関数のログレベルを設定することをお勧めします。詳細については、「the section called “Python でのログレベルフィルタリングの使用”」を参照してください。

Python での Lambda の高度なログ記録コントロールの使用

関数のログのキャプチャ、処理、使用方法をより細かく制御するために、サポートされている Lambda Python ランタイムに以下のログ記録オプションを設定できます。

- ログの形式 - 関数のログをプレーンテキスト形式と構造化された JSON 形式から選択します
- ログレベル - JSON 形式のログの場合、Lambda が Amazon CloudWatch に送信するログの詳細レベル (ERROR、DEBUG、INFO など) を選択します。
- ロググループ - 関数がログを送信する CloudWatch ロググループを選択します

これらのログ記録オプションの詳細と、それらのオプションを使用するように関数を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。

Python Lambda 関数でのログ形式とログレベルオプションの使用の詳細については、以下のセクションのガイダンスをご覧ください。

Python での構造化された JSON ログの使用

関数のログ形式として JSON を選択した場合、Lambda は Python 標準ログ記録ライブラリによって出力されたログを構造化された JSON として CloudWatch に送信します。各 JSON ログオブジェクトには、以下のキーを含む少なくとも 4 つのキーと値のペアが含まれます。

- `"timestamp"` - ログメッセージが生成された時刻
- `"level"` - メッセージに割り当てられたログレベル
- `"message"` - ログメッセージの内容
- `"requestId"` - 関数呼び出しの一意のリクエスト ID

Python `logging` ライブラリは、この JSON オブジェクトに `"logger"` などのキーと値のペアを追加することもできます。

以下のセクションの例は、関数のログ形式を JSON として設定したときに、Python `logging` ライブラリを使用して生成されたログ出力が CloudWatch Logs にどのようにキャプチャされるかを示しています。

the section called “ログへの印刷” で説明されているように `print` メソッドを使用して基本的なログ出力を生成する場合、関数のログ記録形式を JSON として設定した場合でも、Lambda はこれらの出力をプレーンテキストとしてキャプチャすることに注意してください。

Python ログ記録ライブラリを使用した標準 JSON ログ出力

以下のコードスニペットとログ出力例は、関数のログ形式が JSON に設定されている場合、Python `logging` ライブラリを使用して生成された標準ログ出力が CloudWatch Logs にどのようにキャプチャされるかを示しています。

Example Python ログ記録コード

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON ログレコード

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

JSON の追加パラメータの記録

関数のログ形式が JSON に設定されている場合、標準 Python `logging` ライブラリで `extra` キーワードを使用して Python ディクショナリをログ出力に渡すことで、追加のパラメータをログに記録することもできます。

Example Python ログ記録コード

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON ログレコード

```
{
  "timestamp": "2023-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

JSON の例外の記録

以下のコードスニペットは、ログ形式を JSON として設定した場合、Python 例外が関数のログ出力にどのようにキャプチャされるかを示しています。`logging.exception` を使用して生成されたログ出力にはログレベル ERROR が割り当てられることに注意してください。

Example Python ログ記録コード

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON ログレコード

```
{
  "timestamp": "2023-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
```

```
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

他のログ記録ツールでの JSON 構造化ログ

コードで既に Powertools for AWS Lambda などの別のログ記録ライブラリを使用して JSON 構造化ログを生成している場合は、変更を加える必要はありません。AWS Lambda は、既に JSON でエンコードされているログを二重にエンコードしません。JSON ログ形式を使用するように関数を設定しても、ログ記録出力は定義した JSON 構造で CloudWatch に表示されます。

以下の例は、Powertools for AWS Lambda パッケージを使用して生成されたログ出力が CloudWatch Logs にどのようにキャプチャされるかを示しています。このログ出力の形式は、関数のログ記録設定が JSON であっても、TEXT であっても同じです。Powertools for AWS Lambda の使用の詳細については、「the section called “構造化されたログ記録用の Powertools for AWS Lambda (Python) および AWS SAM を使用する”」と「the section called “構造化されたログ記録用の Powertools for AWS Lambda (Python) および AWS CDK を使用する”」を参照してください。

Example Python ログ記録コードスニペット (Powertools for AWS Lambda を使用)

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON ログレコード (Powertools for AWS Lambda を使用)

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2023-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

Python でのログレベルフィルタリングの使用

ログレベルのフィルタリングを設定することで、特定のログレベル以下のログのみを CloudWatch Logs に送信するように選択できます。関数にログレベルのフィルタリングを設定する方法については、「the section called “ログレベルのフィルタリング”」を参照してください。

AWS Lambda でログレベルに従ってアプリケーションログをフィルタリングするには、関数で JSON 形式のログを使用する必要があります。このためには以下の 2 つの方法があります。

- 標準 Python `logging` ライブラリを使用してログ出力を作成し、JSON ログフォーマットを使用するように関数を設定します。その後、AWS Lambda は「the section called “Python での構造化された JSON ログの使用”」で説明されている JSON オブジェクトの「level」キー値のペアを使用してログ出力をフィルタリングします。関数のログ形式を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。
- 別のログ記録ライブラリまたはメソッドを使用して、ログ出力のレベルを定義する「level」キーと値のペアを含む JSON 構造化ログをコード内に作成する。例えば、Powertools for AWS Lambda を使用してコードから JSON 構造化されたログ出力を生成できます。

  print ステートメントを使用して、ログレベル識別子を含む JSON オブジェクトを出力することもできます。以下の print ステートメントは、ログレベルが INFO に設定された JSON 形式の出力を生成します。AWS Lambda は、関数のログ記録レベルが INFO、DEBUG、または TRACE に設定されている場合、JSON オブジェクトを CloudWatch Logs に送信します。

```
print('{"msg":"My log message", "level":"info"}')
```

Lambda で関数のログをフィルタリングするには、JSON ログ出力に `"timestamp"` のキーと値のペアも含める必要があります。時間は、有効な RFC 3339 タイムスタンプ形式で指定する必要があります。有効なタイムスタンプを指定しない場合、Lambda はログに INFO レベルを割り当ててタイムスタンプを追加します。

Lambda コンソールでログを表示する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールで ログを表示する

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/###)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。　X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS CLI でログを表示する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
```

```
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

ツールとライブラリ

Powertools for AWS Lambda (Python) は、サーバーレスのベストプラクティスを実装し、開発者の作業速度を向上させるための開発者ツールキットです。ロガーユーティリティは、Lambda に最適化

されたロガーを提供します。このロガーは、すべての関数の関数コンテキストに関する追加情報を含み、JSON 形式で構成されて出力されます。このユーティリティを使用して次のことができます。

- Lambda の コンテキスト、コールドスタート、構造から主要キーをキャプチャし、JSON 形式でログ出力する
- 指示された場合 Lambda 呼び出しイベントをログ記録する (デフォルトでは無効)
- ログサンプリングにより、特定の割合の呼び出しにのみすべてのログを出力する (デフォルトでは無効)
- 任意のタイミングで、構造化されたログにキーを追加する
- カスタムログフォーマッター (Bring Your Own Formatter) を使用して、組織の ログ記録 RFC と互換性のある構造でログを出力する

構造化されたログ記録用の Powertools for AWS Lambda (Python) および AWS SAM を使用する

以下の手順に従い、AWS SAM を使用する統合された Powertools for Python モジュールを備えた Hello World Python アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Python 3.9
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World Python テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name sam-app
```

ログ出力は次のようになります。

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
```

```
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

ログ保持の管理

関数を削除しても、ロググループは自動的には削除されません。ログを無期限に保存しないようにするには、ロググループを削除するか、CloudWatch がログを自動的に削除するまでの保持期間を設定します。ログ保持を設定するには、AWS SAM テンプレートに以下を追加します。

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

構造化されたログ記録用の Powertools for AWS Lambda (Python) および AWS CDK を使用する

以下の手順に従い、AWS CDK を使用する統合 Powertools for AWS Lambda (Python) モジュールを備えた Hello World Python アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は「hello world」メッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Python 3.9
- AWS CLI バージョン 2
- AWS CDK バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS CDK サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

```
mkdir hello-world
cd hello-world
```

2. アプリケーションを初期化します。

```
cdk init app --language python
```

3. Python の依存関係をインストールします。

```
pip install -r requirements.txt
```

4. ルートフォルダーの下に [lambda_function] ディレクトリを作成します。

```
mkdir lambda_function
cd lambda_function
```

5. ファイル [app.py] を作成して、ファイルに次のコードを追加します。これは Lambda 関数のコードです。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
```

```
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. [hello_world] ディレクトリを開きます。hello-world-stack.py という名前のファイルが表示されます。

```
cd ..
cd hello_world
```

7. hello_world_stack.py を開き、次のコードをファイルに追加します。これには、Lambda 関数を作成し、Powertools の環境変数を構成し、ログ保持を 1 週間に設定する Lambda コンストラクターと、REST API を作成する ApiGatewayv1 コンストラクター が含まれています。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

```
        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. アプリケーションをデプロイします。

```
cd ..
cdk deploy
```

9. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. API エンドポイントを呼び出します。

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

11. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name HelloWorldStack
```

ログ出力は次のようになります。

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2023-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2023-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
```

```
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
```

```
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2023/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2023-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 17 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
cdk destroy
```

Python での AWS Lambda 関数テスト

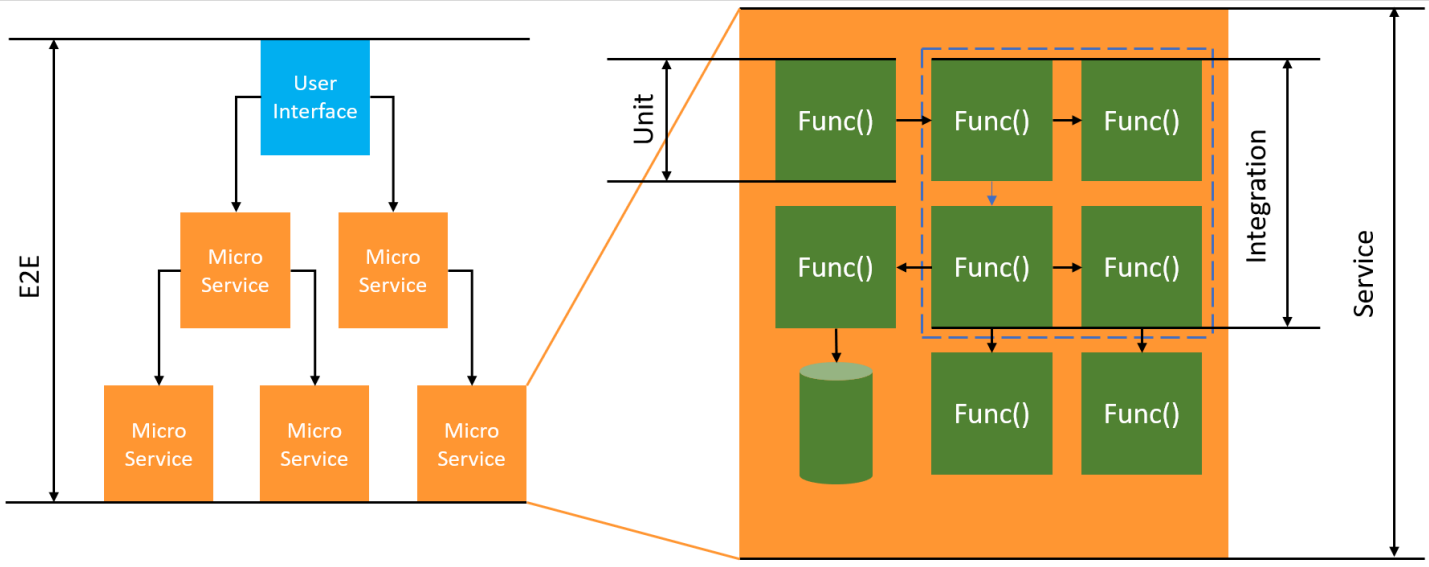
> **Note**
>
> サーバーレスソリューションをテストするためのテクニックとベストプラクティスの詳細については、「関数のテスト」の章を参照してください。

サーバーレス関数のテストでは、従来のテストタイプと手法を使用しますが、サーバーレスアプリケーション全体のテストも検討する必要があります。クラウドベースのテストでは、関数とサーバーレスアプリケーションの両方の品質を最も正確に測定できます。

サーバーレスアプリケーションアーキテクチャには、API 呼び出しを通じて重要なアプリケーション機能を提供するマネージドサービスが含まれます。このため、開発サイクルには、関数とサービスが相互に作用する際に機能を検証する自動テストを含める必要があります。

クラウドベースのテストを作成しない場合、ローカル環境とデプロイされた環境の違いにより問題が発生する可能性があります。継続的な統合プロセスでは、コードをQA、ステージング、本番稼働などの次のデプロイ環境に昇格する前に、クラウドにプロビジョニングされた一連のリソースに対してテストを実行する必要があります。

サーバーレスアプリケーションのテスト戦略に関する詳細については、このショートガイドを引き続きご覧ください。また、サーバーレステストサンプルリポジトリにアクセスして、選択した言語とランタイムに固有の実用的な例を調べることもできます。



サーバーレステストでも、ユニット 、 統合、end-to-endテストを記述します。

- ユニットテスト - 分離されたコードブロックに対して実行されるテスト。例えば、特定の商品と配送先を指定して配送料を計算するビジネスロジックを検証する場合です。
- 統合テスト - 通常はクラウド環境で相互作用する 2 つ以上のコンポーネントまたはサービスを対象としたテスト。例えば、キューからのイベントを処理する関数を検証する場合です。
- E nd-to-end テスト - アプリケーション全体の動作を検証するテスト。例えば、インフラストラクチャが正しくセットアップされ、顧客の注文を記録するためにイベントがサービス間で想定どおりに流れることを確認する場合です。

サーバーレスアプリケーションのテスト

通常、サーバーレスアプリケーションコードのテストには、クラウドでのテスト、モックを使ったテスト、場合によってはエミュレーターでのテストなど、さまざまな方法を組み合わせます。

クラウドでのテスト

クラウドでのテストは、ユニットテスト、統合テスト、テストなど、 end-to-end テストのすべてのフェーズで役立ちます。クラウドにデプロイされたコードやクラウドベースのサービスとやり取りするコードに対してテストを実行します。この方法では、コードの品質を最も正確に測定できます。

クラウドで Lambda 関数をデバッグする便利な方法は、コンソールからテストイベントを行うことです。テストイベントとは、関数への JSON 入力のことです。関数が入力を必要としない場合、イベントは空の JSON ドキュメント ({}) にすることができます。コンソールには、さまざまなサービス統合のサンプルイベントが用意されています。コンソールでイベントを作成したら、それをチームと共有して、テストを簡単かつ一貫性のあるものにすることができます。

> **Note**
>
> コンソールで関数をテストするのが簡単な方法ですが、テストサイクルを自動化することでアプリケーションの品質と開発スピードが保証されます。

テストツール

開発フィードバックのループを高速化するためのツールやテクニック。例えば、AWS SAM Accelerate と AWSCDK 監視モードは、いずれもクラウド環境の更新に要する時間を短縮します。

Moto は AWS サービスやリソースを模倣するための Python ライブラリです。応答をインターセプトしてシミュレートするデコレータを使用して、ほとんどまたはまったく変更を加えずに関数をテストできます。

Powertools for AWS Lambda (Python) の検証機能にはデコレータが用意されているため、Python 関数からの入力イベントと出力応答を検証できます。

詳細については、ブログ記事「Python と AWS モックサービスによる Lambda の単体テスト」を参照してください。

クラウドデプロイの反復に伴うレイテンシーを減らすには、「AWS サーバーレスアプリケーションモデル (AWS SAM) アクセラレート」、「AWS Cloud Development Kit (AWS CDK) ウォッチモード」を参照してください。これらのツールは、インフラストラクチャとコードの変更を監視します。これらの変更に対応して、増分更新を自動的に作成してクラウド環境にデプロイします。

これらのツールを使用する例は、Python テストサンプルのコードリポジトリにあります。

AWS Lambda での Python コードの作成

Lambda アプリケーションのトレース、デバッグ、および最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレーシングデータを X-Ray に送信するには、以下に表示された 3 つの SDK ライブラリのいずれかを使用できます。

- AWS Distro for OpenTelemetry (ADOT) – 安全で本番環境に対応し、AWS でサポートされている OpenTelemetry (OTel) SDK のディストリビューションです。
- AWS X-Ray SDK for Python - トレースデータを生成して X-Ray に送信するための SDK です。
- Powertools for AWS Lambda (Python) - サーバーレスのベストプラクティスを実装し、デベロッパーの作業速度を向上させるためのデベロッパーツールキットです。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。これらのソリューションのいずれかを使用して、X-Ray でエンドツーエンドのトレーシングを実装することができます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

- トレースに Powertools for AWS Lambda (Python) と AWS SAM を使用する
- トレースに Powertools for AWS Lambda (Python) と AWS CDK を使用する
- ADOT を使用して Python 関数をインストルメント化する
- X-Ray SDK を使用して Python 関数をインストルメント化する

- Lambda コンソールを使用してトレースを有効化する
- Lambda API でのトレースのアクティブ化
- AWS CloudFormation によるトレースのアクティブ化
- X-Ray トレースの解釈
- ランタイムの依存関係をレイヤー (X-Ray SDK) に保存する

トレースに Powertools for AWS Lambda (Python) と AWS SAM を使用する

以下の手順に従い、AWS SAM を使用する統合 Powertools for AWS Lambda (Python) モジュールを備えた Hello World Python アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は「hello world」メッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Python 3.9
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World Python テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
```

```
      - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

トレースに Powertools for AWS Lambda (Python) と AWS CDK を使用する

以下の手順に従い、AWS CDK を使用する統合 Powertools for AWS Lambda (Python) モジュールを備えた Hello World Python アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は「hello world」メッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Python 3.9
- AWS CLI バージョン 2
- AWS CDK バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS CDK サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

```
mkdir hello-world
cd hello-world
```

2. アプリケーションを初期化します。

```
cdk init app --language python
```

3. Python の依存関係をインストールします。

```
pip install -r requirements.txt
```

4. ルートフォルダーの下に [lambda_function] ディレクトリを作成します。

```
mkdir lambda_function
cd lambda_function
```

5. ファイル [app.py] を作成して、ファイルに次のコードを追加します。これは Lambda 関数のコードです。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. [hello_world] ディレクトリを開きます。hello-world-stack.py という名前のファイルが表示されます。

```
cd ..
cd hello_world
```

7. hello_world_stack.py を開き、次のコードをファイルに追加します。これには、Lambda 関数を作成し、Powertools の環境変数を構成し、ログ保持を 1 週間に設定する Lambda コンストラクターと、REST API を作成する ApiGatewayv1 コンストラクター が含まれています。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
```

```
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. アプリケーションをデプロイします。

```
cd ..
cdk deploy
```

9. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

11. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
cdk destroy
```

ADOT を使用して Python 関数をインストルメント化する

ADOT は、Otel SDK を使用してテレメトリデータを収集するために必要なすべてをパッケージ化した、フルマネージド型の Lambda レイヤーを提供します。このレイヤーを使用すると、関数コードを変更する必要はなしで、Lambda 関数を計測できます。また、このレイヤーは、OTel でのカスタムな初期化を実行するように構成することもできます。詳細については、ADOT のドキュメントの「Lambda 上での ADOT Collector のカスタム設定」を参照してください。

Python ランタイムの場合は、AWS マネージド Lambda layer for ADOT Python を追加して、関数を自動的に計測できます。このレイヤーは、arm64 アーキテクチャと x86_64 アーキテクチャの両方で機能します。このレイヤーの使用方法の詳細については、ADOT のドキュメントの「AWS Distro for OpenTelemetry Lambda Support for Python」を参照してください。

X-Ray SDK を使用して Python 関数をインストルメント化する

Lambda 関数がアプリケーション内の他のリソースに対して行う呼び出しの詳細を記録するために、AWS X-Ray SDK for Python を使用することもできます。SDK を取得するには、アプリケーションの依存関係に `aws-xray-sdk` パッケージを追加します。

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

機能コードでは、`boto3` ライブラリに `aws_xray_sdk.core` モジュールをパッチすることにより、AWS SDK クライアントをインストルメント化できます。

Example 関数 - AWS SDK クライアントのトレース

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()
```

```
def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

正しい依存関係を追加し、必要なコード変更を行った後、Lambda コンソールまたはAPIを介して関数の設定でトレースをアクティブにします。

Lambda コンソールを使用してトレースを有効化する

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。
4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [保存] をクリックします。

Lambda API でのトレースのアクティブ化

AWS CLI または AWS SDK で Lambda 関数のトレースを設定するには、次の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation によるトレースのアクティブ化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray トレースの解釈

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

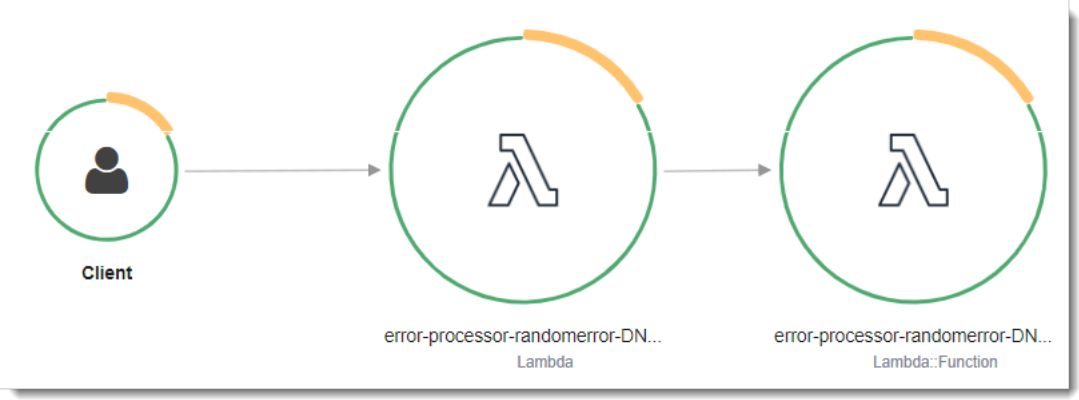
アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロググ

ループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

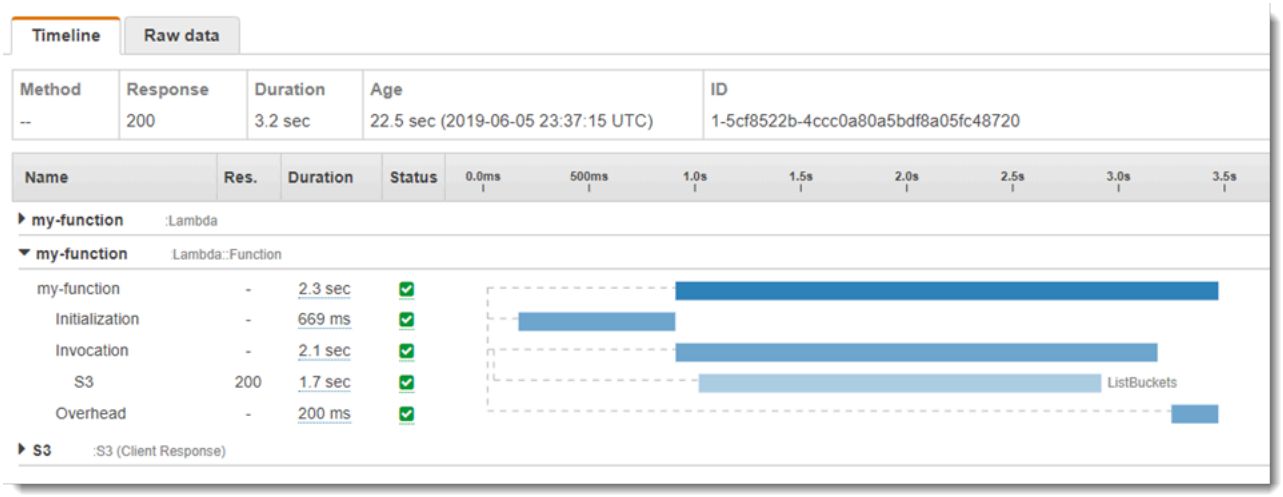
X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは AWS::Lambda の起点があり、もう 1 つは AWS::Lambda::Function の起点があります。AWS::Lambda セグメントにエラーが表示される場合は、Lambda サービスに問題があります。AWS::Lambda::Function セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、「AWS X-Ray デベロッパーガイド」の「AWS X-Ray SDK for Python」を参照してください。

> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

ランタイムの依存関係をレイヤー (X-Ray SDK) に保存する

X-Ray SDK を使用して AWS SDK クライアントを関数コードに埋め込むと、デプロイパッケージが巨大になる可能性があります。関数コードを更新するたびにランタイムの依存関係がアップロードされないようにするには、X-Ray SDK を「Lambda レイヤー」にパッケージ化します。

次に、AWS X-Ray SDK for Python を保存している `AWS::Serverless::LayerVersion` リソースの例を示します。

Example template.yml - 依存関係レイヤー

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.8
```

この設定では、ランタイム依存関係を変更した場合にのみ、ライブラリレイヤーの更新が必要です。関数のデプロイパッケージにはユーザーのコードのみが含まれるため、アップロード時間を短縮できます。

依存関係のレイヤーを作成するには、デプロイ前にレイヤーアーカイブを生成するようにビルドを変更する必要があります。実際の例については、blank-python サンプルアプリケーションを参照してください。

Ruby による Lambda 関数の構築

Ruby コードは AWS Lambda で実行できます。Lambda は、コードを実行してイベントを処理する Ruby 用のランタイムを提供します。コードは、管理している AWS Identity and Access Management (IAM) ロールの認証情報を使用して、AWS SDK for Ruby を含む環境で実行されます。Ruby ランタイムに含まれている SDK バージョンの詳細については、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

Lambda は以下の Ruby ランタイムをサポートします。

Ruby

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Ruby 3.3	`ruby3.3`	Amazon Linux 2023			
Ruby 3.2	`ruby3.2`	Amazon Linux 2			

Ruby 関数を作成するには

1. Lambda コンソールを開きます。
2. [Create function] (関数の作成) をクリックします。
3. 以下の設定を行います。
   - [Function name]: 関数名を入力します。
   - [ランタイム]:[Ruby 3.2] を選択します。
4. [Create function] (関数の作成) をクリックします。
5. テストイベントを設定するには、[テスト] を選択します。
6. [イベント名] で、「**test**」と入力します。
7. [変更を保存] をクリックします。
8. [テスト] を選択して関数を呼び出します。

コンソールで、lambda_function.rb という名前の単一のソースファイルを含む Lambda 関数が作成されます。このファイルを編集し、組み込みのコードエディタでファイルをさらに追加することができます。変更を保存するには [保存] を選択します。コードを実行するには、[Test] (テスト) を選択します。

> **Note**
>
> Lambda コンソールでは、AWS Cloud9 を使用して、ブラウザに統合開発環境を提供します。また、AWS Cloud9 を使用して、独自の環境で Lambda 関数を開発することもできます。詳細については、AWS Cloud9 ユーザーガイドの「AWS Toolkit を使用した AWS Lambda 関数の使用」を参照してください。

lambda_function.rb ファイルは、イベントオブジェクトおよびコンテキストオブジェクトを取得する lambda_handler という名前の関数をエクスポートします。これは、関数が呼び出されるときに Lambda が呼び出すハンドラー関数です。Ruby 関数のランタイムは、Lambda から呼び出しイベントを取得し、ハンドラーに渡します。関数設定で、ハンドラ値は lambda_function.lambda_handler です。

関数コードを保存すると、Lambda コンソールは .zip ファイルアーカイブのデプロイパッケージを作成します。コンソール外で (SDE を使用して) 関数コードを開発するときは、デプロイパッケージを作成して、Lambda 関数にコードをアップロードします。

> **Note**
>
> ローカル環境でアプリケーション開発を開始するには、このガイドの GitHub リポジトリで利用可能なサンプルアプリケーションの 1 つをデプロイします。
>
> Ruby のサンプル Lambda アプリケーション
>
> - blank-ruby - ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Ruby 関数。
> - AWS Lambda の Ruby コードサンプル – AWS Lambda との対話方法を示す、Ruby で記述されたコードサンプル。

関数のランタイムによって、呼び出しイベントに加えて、コンテキストオブジェクトがハンドラに渡されます。コンテキストオブジェクトには、呼び出し、関数、および実行環境に関する追加情報が含まれます。詳細情報は、環境変数から入手できます。

Lambda 関数には CloudWatch Logs ロググループが付属しています。関数のランタイムは、各呼び出しに関する詳細を CloudWatch Logs に送信します。これは呼び出し時に、任意の関数が出力するログを中継します。関数がエラーを返す場合、Lambda はエラー形式を整え、それを呼び出し元に返します。

トピック

ランタイムに含まれる SDK バージョン

Ruby ランタイムに含まれる AWS SDK のバージョンは、ランタイムバージョンと AWS リージョン によって異なります。AWSSDK for Ruby はモジュール式に設計されており、AWS のサービスごとに分かれています。使用しているランタイムに含まれている特定のサービス gem のバージョン番号を確認するには、次の形式のコードを使用して Lambda 関数を作成します。`aws-sdk-s3` と `Aws::S3` を、コードが使用するサービス gem の名前に置き換えます。

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
end
```

もうひとつの Ruby JIT (YJIT) を有効にする

Ruby 3.2 ランタイムは、軽量でミニマルな Ruby JIT コンパイラである YJIT をサポートしています。YJIT はパフォーマンスを大幅に向上しますが、Ruby インタープリタよりも多くのメモリを消費します。Ruby on Rails のワークロードには YJIT が推奨されます。

YJIT は、デフォルトでは有効になっていません。Ruby 3.2 関数で YJIT を有効にするには、RUBY_YJIT_ENABLE 環境変数を 1 に設定します。YJIT が有効であることを確認するには、RubyVM::YJIT.enabled? メソッドの結果を出力します。

Example — YJIT が有効になっていることの確認

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Ruby の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

次の例では、`function.rb` は、`handler` という名前のハンドラメソッドを定義します。ハンドラ関数は、2 つのオブジェクトを入力として識別し、JSON ドキュメントを返します。

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

関数設定の `handler` 設定は、ハンドラーの場所を Lambda に伝えます。前述の例では、この設定の適切な値は **`function.handler`** です。ドットで区切られた 2 つの名前 (ファイルの名前とハンドラメソッドの名前) が含まれています。

また、クラス内のハンドラメソッドを定義することもできます。以下の例では、モジュール `LambdaFunctions` のクラス `Handler` の ハンドラメソッド `process` を定義します。

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

この場合、ハンドラ設定は **`source.LambdaFunctions::Handler.process`** です。

ハンドラは、呼び出しイベントとコンテキストの 2 つのオブジェクトを受け入れます。このイベントは、呼び出し元が提供するペイロードを含む Ruby オブジェクトです。ペイロードが JSON ドキュメントの場合、イベントオブジェクトは Ruby ハッシュです。それ以外の場合は、文字列です。コンテキストオブジェクトには、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティがあります。

関数ハンドラーは、Lambda 関数が呼び出されるたびに実行されます。ハンドラの外側の静的コードは、関数のインスタンスごとに 1 回実行されます。ハンドラで SDK クライアントやデータベース接続などのリソースが使用される場合、ハンドラメソッドは、ハンドラメソッドの外で作成して複数の呼び出しに再利用することができます。

関数の各インスタンスで複数の呼び出しイベントを処理することはできますが、処理されるのは一度に 1 つのイベントのみです。任意の時点でイベントを処理するインスタンス数は、関数の同時実行数です。Lambda の実行環境の詳細については、Lambda 実行環境 を参照してください。

Ruby Lambda 関数で .zip ファイルアーカイブを使用する

AWS Lambda 関数のコードは、関数のハンドラーコードを含む .rb ファイルと、そのコードが依存する追加の依存関係 (gem) で構成されています。この関数コードを Lambda にデプロイするには、デプロイパッケージを使用します。このパッケージは、.zip ファイルアーカイブでもコンテナイメージでもかまいません。Ruby でコンテナイメージを使用する方法の詳細については、「コンテナイメージで Ruby Lambda 関数をデプロイする」を参照してください。

.zip ファイルのデプロイパッケージを .zip ファイルアーカイブとして作成するには、コマンドラインツール用の組み込み .zip ファイルアーカイブユーティリティ、または他の .zip ファイルユーティリティ (7zip など) を使用します。次のセクションに示す例では、Linux または macOS 環境でコマンドライン `zip` ツールを使用していることを前提としています。Windows で同じコマンドを使用するには、Windows Subsystem for Linux をインストールして、Windows 統合バージョンの Ubuntu と Bash を取得します

Lambda は POSIX ファイルアクセス許可を使用するため、.zip ファイルアーカイブを作成する前に、デプロイパッケージフォルダのアクセス許可を設定することが必要になる場合があります。

以下のセクションのコマンド例では、バンドラーユーティリティを使用してデプロイパッケージに依存関係を追加します。バンドラーをインストールするには、以下のコマンドを実行します。

```
gem install bundler
```

セクション

Ruby の依存関係

Ruby ランタイムを使用する Lambda 関数の場合、依存関係には任意の Ruby gem を使用できます。.zip アーカイブを使用して関数をデプロイするとき、関数コードでこれらの依存関係を .zip ファ

イルに追加するか、Lambda レイヤー を使用できます。レイヤーは、追加のコードまたはその他のコンテンツを含むことができる個別の .zip ファイルです。Lambda レイヤーの使用の詳細については、「Lambda レイヤー」を参照してください。

Ruby ランタイムには AWS SDK for Ruby が含まれます。関数で SDK を使用する場合は、それをコードにバンドルする必要はありません。ただし、依存関係を完全に制御したり、特定のバージョンの SDK を使用したりするには、関数のデプロイパッケージに追加することができます。SDK は .zip ファイルに含めるか、Lambda レイヤーを使用して追加することができます。.zip ファイルまたは Lambda レイヤー内の依存関係は、ランタイムに含まれるバージョンよりも優先されます。ご使用のランタイムバージョンに含まれている SDK for Ruby のバージョンを確認するには、「the section called “ランタイムに含まれる SDK バージョン”」を参照してください。

AWS 責任分担モデルでは、関数のデプロイパッケージに含まれる依存関係を管理する責任があります。これには、更新とセキュリティパッチの適用が含まれます。関数のデプロイパッケージ内の依存関係を更新するには、まず新しい .zip ファイルを作成し、そのファイルを Lambda にアップロードします。詳細については、「依存関係を含む .zip デプロイパッケージを作成する」と「.zip ファイルを使用した Ruby Lambda 関数の作成と更新」を参照してください。

依存関係のない .zip デプロイパッケージを作成する

関数コードに依存関係がない場合、.zip ファイルには関数のハンドラーコードを含む .rb ファイルのみが含まれます。任意の zip ユーティリティを使用して、.rb ファイルをルートとする .zip ファイルを作成します。.rb ファイルが .zip ファイルのルートにない場合、Lambda はコードを実行できません。

.zip ファイルをデプロイして新しい Lambda 関数を作成する方法の詳細、既存の Lambda 関数を更新する方法の詳細については、「.zip ファイルを使用した Ruby Lambda 関数の作成と更新」を参照してください。

依存関係を含む .zip デプロイパッケージを作成する

関数コードが追加の Ruby gem に依存している場合、これらの依存関係を関数コードとともに .zip ファイルに追加するか、Lambda レイヤーを使用できます。このセクションでは、依存関係を .zip デプロイパッケージに含める方法について説明します。依存関係をレイヤーに含める方法については、「the section called “依存関係の Ruby レイヤーを作成する”」を参照してください。

関数コードがプロジェクトディレクトリの `lambda_function.rb` という名前のファイルに保存されているとします。次の CLI コマンドの例では、関数コードとその依存関係を格納している `my_deployment_package.zip` という名前の .zip ファイルを作成します。

デプロイパッケージを作成するには

1. プロジェクトディレクトリに、依存関係を指定する Gemfile を作成します。

```
bundle init
```

2. 任意のテキストエディタを使用して、Gemfile を編集して関数の依存関係を指定します。例えば、TZInfo gem を使用するには、Gemfile を次のように編集します。

```
source "https://rubygems.org"
gem "tzinfo"
```

3. 次のコマンドを実行して、Gemfile で指定した gem をプロジェクトディレクトリにインストールします。このコマンドでは vendor/bundle を gem インストールのデフォルトパスとして設定します。

```
bundle config set --local path 'vendor/bundle' && bundle install
```

次のような出力が表示されます。

```
Fetching gem metadata from https://rubygems.org/...........
Resolving dependencies...
Using bundler 2.4.13
Fetching tzinfo 2.0.6
Installing tzinfo 2.0.6
...
```

> **Note**
>
> 後で gem をグローバルにインストールし直すには、次のコマンドを実行します。
>
> ```
> bundle config set --local system 'true'
> ```

4. 関数のハンドラーコードを含む lambda_function.rb ファイルと、前のステップでインストールした依存関係を格納した .zip ファイルアーカイブを作成します。

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

次のような出力が表示されます。

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

依存関係の Ruby レイヤーを作成する

このセクションでは、依存関係をレイヤーに含める方法について説明します。依存関係をデプロイパッケージに含める方法については、「the section called “依存関係を含む .zip デプロイパッケージを作成する”」を参照してください。

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の /opt ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には /opt ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

- `ruby/gems/2.7.0` (GEM_PATH)
- `ruby/lib` (RUBYLIB)

例えば、レイヤーの.zip ファイルの構造は次のようになります。

```
json.zip
# ruby/gems/2.7.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

さらに、Lambda は /opt/lib ディレクトリ内のライブラリ、および /opt/bin ディレクトリ内のバイナリを自動的に検出します。Lambda がレイヤーのコンテンツを正しく検出できるように、次の構造でレイヤーを作成することもできます。

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

レイヤーをパッケージ化したら、「the section called “レイヤーの作成と削除”」および「the section called “レイヤーの追加”」を参照してレイヤーの設定を完了してください。

ネイティブライブラリとともに .zip デプロイパッケージを作成する

nokogiri、nio4r、mysql などの多くの一般的な Ruby gem には、C で書かれたネイティブの拡張機能が含まれています。C コードを含むライブラリをデプロイパッケージに追加するときは、パッケージを正しく構築し、Lambda 実行環境と互換性があることを確認する必要があります。

本番環境のアプリケーションでは、AWS Serverless Application Model (AWS SAM) を使用してコードをビルドしてデプロイすることをお勧めします。AWS SAM では、Lambda のような Docker コンテナ内部に関数を構築するのに、sam build --use-container オプションを使用します。AWS SAM を使用して関数コードをデプロイする方法の詳細については、「AWS SAM デベロッパーガイド」の「アプリケーションの構築」を参照してください。

AWS SAM を使用せずに gem を含む .zip デプロイパッケージをネイティブ拡張で作成するには、代わりにコンテナを使用して Lambda Ruby ランタイム環境と同じ環境に依存関係をバンドルすることもできます。これらの手順を完了するには、ビルドマシンに Docker がインストールされている必要があります。Docker のインストールの詳細については、「Docker Engine のインストール」を参照してください。

Docker コンテナに.zip デプロイパッケージを作成するには

1. コンテナを保存するフォルダをローカルビルドマシンに作成します。そのフォルダ内に、dockerfile という名前のファイルを作成し、次のコードを貼り付けます。

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   ```

```
CMD "/bin/bash"
```

2. dockerfile を作成したフォルダ内で、次のコマンドを実行して Docker コンテナを作成します。

```
docker build -t awsruby32 .
```

3. 関数のハンドラーコードが記載された .rb ファイルと関数の依存関係を指定した Gemfile を含むプロジェクトディレクトリに移動します。そのディレクトリ内から、次のコマンドを実行して Lambda Ruby コンテナを起動します。

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

   > **Note**
   >
   > MacOS では、リクエストされたイメージのプラットフォームが検出されたホストプラットフォームと一致しないという警告が表示されることがあります。この警告は無視してください。

   Windows PowerShell

   ```
   docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
   ```

   コンテナが起動すると、bash プロンプトが表示されます。

   ```
   bash-4.2#
   ```

4. バンドルユーティリティを設定して、Gemfile で指定した gem をローカルの vendor/bundle ディレクトリにインストールし、依存関係をインストールします。

   ```
   bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
   ```

5. 関数コードとその依存関係を含む .zip デプロイパッケージを作成します。この例では、関数のハンドラーコードを含むファイルには lambda_function.rb という名前が付けられています。

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. コンテナを終了し、ローカルプロジェクトディレクトリに戻ります。

```
bash-4.2# exit
```

これで、.zip ファイルデプロイパッケージを使用して Lambda 関数を作成または更新できます。「.zip ファイルを使用した Ruby Lambda 関数の作成と更新」を参照してください。

.zip ファイルを使用した Ruby Lambda 関数の作成と更新

.zip デプロイパッケージを作成すると、このパッケージを使用して新しい Lambda 関数を作成するか、既存の関数を更新できます。.zip パッケージをデプロイするには、Lambda コンソール、AWS Command Line Interface、Lambda API を使用します。AWS Serverless Application Model (AWS SAM) および AWS CloudFormation を使用して、Lambda 関数を作成および更新することもできます。

Lambda の .zip デプロイパッケージの最大サイズは 250 MB (解凍) です。この制限は、Lambda レイヤーを含む、更新するすべてのファイルの合計サイズに適用されることに注意してください。

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

コンソールを使用して .zip ファイルの関数を作成、更新する

新しい関数を作成するには、まずコンソールで関数を作成し、次に .zip アーカイブをアップロードする必要があります。既存の関数を更新するには、その関数のページを開き、同じ手順に従って更新した .zip ファイルを追加します。

.zip ファイルが 50 MB 未満の場合は、ローカルマシンから直接ファイルをアップロードして関数を作成または更新できます。50 MB を超える .zip ファイルの場合は、まず Amazon S3 バケットにパッケージをアップロードする必要があります。AWS Management Console を使用して Amazon S3 バケットにファイルをアップロードする手順については、「Amazon S3 の開始方法」を参照してください。AWS CLI を使用してファイルをアップロードするには、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [一から作成] を選択します。
3. [基本的な情報] で、以下を行います。
   a. [関数名] に、関数名を入力します。
   b. [ランタイム] で、使用するランタイムを選択します。
   c. (オプション) [アーキテクチャ]で、関数の命令セットアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用の .zip デプロイパッケージと選択した命令セットのアーキテクチャに互換性があることを確認してください。
4. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。新しい [実行ロール] を作成することも、既存のロールを使用することもできます。
5. [関数の作成]を選択します。Lambda は、選択したランタイムを使用して基本的な「Hello world」関数を作成します。

ローカルマシンから zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、.zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [.zip ファイル] をクリックします。
5. .zip ファイルをアップロードするには、次の操作を行います。
   a. [アップロード] をクリックし、ファイルセレクターで .zip ファイルを選択します。
   b. [開く] をクリックします。
   c. [保存] をクリックします。

Amazon S3 バケットから .zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、新しい .zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [Amazon S3 ロケーション] を選択します。
5. .zip ファイルの Amazon S3 リンク URL を貼り付けて、[保存] を選択します。

コンソールコードエディタを使用して .zip ファイル関数を更新する

.zip デプロイパッケージを使用する一部の関数では、Lambda コンソールの組み込みコードエディタを使用して、関数コードを直接更新できます。この機能を使用するには、関数が次の基準を満たしている必要があります。

- 関数が、インタープリター言語ランタイムのいずれか (Python、Node.js、Ruby) を使用する必要があります。
- 関数のデプロイパッケージが 3 MB 未満である必要があります。

コンテナイメージデプロイパッケージを含む関数の関数コードは、コンソールで直接編集することはできません。

コンソールのコードエディタを使用して関数コードを更新するには

1. Lambda コンソールの「関数ページ」を開き、関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインでソースコードファイルを選択し、統合コードエディタで編集します。
4. コードの編集が終了したら、[デプロイ] を選択して変更を保存し、関数を更新します。

AWS CLI を使用して .zip ファイルで関数を作成、更新する

AWS CLI を使用して新しい関数を作成したり、.zip ファイルを使用して既存の関数を更新したりできます。create-function コマンドと update-function-code を使用して、.zip パッケージをデプロイします。.zip ファイルが 50 MB 未満の場合は、ローカルビルドマシン上のファイルの場所から .zip パッケージをアップロードできます。サイズの大きいファイルの場合は、Amazon S3 バケットから .zip パッケージをアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> AWS CLI を使用して Amazon S3 バケットから .zip ファイルをアップロードする場合、このバケットは関数と同じ AWS リージョン に配置する必要があります。

AWS CLI を含む .zip ファイルを使用して新しい関数を作成するには、以下を指定する必要があります。

- 関数の名前 (`--function-name`)
- 関数のランタイム (`--runtime`)
- 関数の実行ロール (`--role`) の Amazon リソースネーム (ARN)
- 関数コード内のハンドラーメソッド (`--handler`) の名前

.zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、`--zip-file` オプションを使用してファイルパスを指定します。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
```

```
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある `--code` オプションを使用します。`S3ObjectVersion` パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

CLI を使用して既存の関数を更新するには、`--function-name` パラメータを使用して関数の名前を指定します。関数コードの更新に使用する .zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、`--zip-file` オプションを使用してファイルパスを指定します。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある `--s3-bucket` および `--s3-key` オプションを使用します。`--s3-object-version` パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API を使用して .zip ファイルで関数を作成、更新する

.zip ファイルアーカイブを使用して関数を作成および更新するには、以下の API オペレーションを使用します。

- CreateFunction
- UpdateFunctionCode

AWS SAM を使用して .zip ファイルで関数を作成、更新する

AWS Serverless Application Model (AWS SAM) は、AWS のサーバーレスアプリケーションの構築と実行のプロセスを合理化するのに役立つツールキットです。YAML または JSON テンプレートでアプリケーションのリソースを定義し、AWS SAM コマンドラインインターフェイス (AWS SAM CLI) を使用して、アプリケーションを構築、パッケージ化、デプロイします。AWS SAM テンプレートから Lambda 関数を構築すると、AWS SAM は関数コードと指定した任意の依存関係を含む .zip デプロイパッケージまたはコンテナイメージを自動的に作成します。AWS SAM を使用して Lambda 関数を構築およびデプロイする方法の詳細については、「AWS Serverless Application Model 開発者ガイドの」の「AWS SAM の開始方法」を参照してください。

AWS SAM を使用して、既存の .zip ファイルアーカイブを使用する Lambda 関数を作成できます。AWS SAM を使用して Lambda 関数を作成するには、.zip ファイルを Amazon S3 バケットまたはビルドマシンのローカルフォルダに保存します。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS SAM テンプレートでは、Lambda 関数は `AWS::Serverless::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `CodeUri` - 関数コードの Amazon S3 URI、ローカルフォルダへのパス、または FunctionCode オブジェクトに設定
- `Runtime` - 選択したランタイムに設定

AWS SAM では、.zip ファイルが 50 MB を超える場合、この .zip ファイルを最初に Amazon S3 バケットにアップロードする必要はありません。AWS SAM では、ローカルビルドマシン上の場所から、最大許容サイズ 250 MB (解凍) の .zip パッケージをアップロードできます。

AWS SAM で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS SAM 開発者ガイド」の「AWS::Serverless::Function」を参照してください。

AWS CloudFormation を使用して .zip ファイルで関数を作成、更新する

AWS CloudFormation を使用して、.zip ファイルアーカイブを使用する Lambda 関数を作成できます。.zip ファイルから Lambda 関数を作成するには、最初にファイルを Amazon S3 バケットにアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロー

ドする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS CloudFormation テンプレートでは、Lambda 関数は `AWS::Lambda::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `Code` - `S3Bucket` および `S3Key` フィールドに Amazon S3 バケット名と .zip ファイル名を入力
- `Runtime` - 選択したランタイムに設定

AWS CloudFormation が生成する .zip ファイルは、4 MB を超えることはできません。AWS CloudFormation で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::Lambda::Function」を参照してください。

コンテナイメージで Ruby Lambda 関数をデプロイする

Ruby Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- Ruby の AWS ベースイメージを使用する

  AWS ベースイメージには、言語ランタイム、Lambda と関数コード間のやり取りを管理するランタイムインターフェースクライアント、ローカルテスト用のランタイムインターフェースエミュレーターがあらかじめロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、Ruby のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することができます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、Ruby のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

トピック

- Ruby の AWS ベースイメージ

- Ruby の AWS ベースイメージを使用する
- ランタイムインターフェイスクライアントで代替ベースイメージを使用する

Ruby の AWS ベースイメージ

AWS は、Ruby の次のベースイメージを提供します。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
3.3	Ruby 3.3	Amazon Linux 2023	「GitHub の Ruby 3.3 用 Dockerfile」	
3.2	Ruby 3.2	Amazon Linux 2	GitHub の Ruby 3.2 用 Dockerfile	

Amazon ECR リポジトリ: gallery.ecr.aws/lambda/ruby

Ruby の AWS ベースイメージを使用する

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker
- Ruby

ベースイメージからイメージを作成する

Ruby のコンテナイメージを作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir example
```

```
cd example
```

2. Gemfile という名前の新しいファイルを作成します。ここでは、アプリケーションに必要な RubyGems パッケージを一覧表示します。AWS SDK for Ruby は RubyGems から入手できます。インストールする特定の AWS サービス gem を選択する必要があります。例えば、Lambda 用 Ruby gem を使用するには、Gemfile は次のようになるはずです。

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

あるいは、aws-sdk gem には利用可能なすべての AWS サービス gem が含まれています。この gem はとても大きいことが特徴です。多くの AWS サービスを利用している場合にのみ使用することを推奨します。

3. バンドルインストールを使用して、Gemfile で指定された依存関係をインストールします。

```
bundle install
```

4. lambda_function.rb という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

Example Ruby 関数

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 新しい Dockerfile を作成します。次に、AWS ベースイメージ を使用した Dockerfile の例を示します。この Dockerfile では次の設定を使用します。

- ベースイメージの URI に FROM プロパティを設定します。
- COPY コマンドを使用し、関数コードおよびランタイムの依存関係を {LAMBDA_TASK_ROOT} (Lambda 定義の環境変数) にコピーします。
- CMD 引数を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.2

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

1. docker run コマンドを使用して、Docker イメージを起動します。この例では、docker-image はイメージ名、test はタグです。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、--platform linux/*amd64* の代わりに --platform linux/*arm64* オプションを使用してください。

2. 新しいターミナルウィンドウから、イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の curl コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の Invoke-WebRequest コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - --region 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   例：

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず `:latest` を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda

は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、`update-function-code` コマンドを使用する必要があります。

ランタイムインターフェイスクライアントで代替ベースイメージを使用する

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間の相互作用を管理する Lambda Runtime API を拡張します。

RubyGems.org パッケージマネージャーを使用して、Ruby 用の Lambda ランタイムインターフェイスクライアントをインストールします。

```
gem install aws_lambda_ric
```

Ruby ランタイムインターフェイスクライアントを GitHub からダウンロードすることもできます。ランタイムインターフェイスクライアントは、Ruby バージョン 2.5.x から 2.7.x に対応しています。

次の例は、非 AWS ベースイメージを使用して Ruby 用のコンテナイメージを構築する方法を示しています。サンプルの Dockerfile は公式の Ruby ベースイメージを使用しています。Dockerfile には、ランタイムインターフェイスクライアントが含まれています。

前提条件

このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- Docker
- Ruby

代替ベースイメージからイメージを作成する

代替のベースイメージを使用して Ruby 用のコンテナイメージを作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir example
```

```
cd example
```

2. Gemfile という名前の新しいファイルを作成します。ここでは、アプリケーションに必要な RubyGems パッケージを一覧表示します。AWS SDK for Ruby は RubyGems から入手できます。インストールする特定の AWS サービス gem を選択する必要があります。例えば、Lambda 用 Ruby gem を使用するには、Gemfile は次のようになるはずです。

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

あるいは、aws-sdk gem には利用可能なすべての AWS サービス gem が含まれています。この gem はとても大きいことが特徴です。多くの AWS サービスを利用している場合にのみ使用することを推奨します。

3. バンドルインストールを使用して、Gemfile で指定された依存関係をインストールします。

```
bundle install
```

4. lambda_function.rb という名前の新しいファイルを作成します。テスト用に次のサンプル関数コードをファイルに追加することも、独自のコードを使用することもできます。

Example Ruby 関数

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 新しい Dockerfile を作成します。次の Dockerfile は、AWS ベースイメージの代わりに Ruby ベースイメージを使用しています。Dockerfile には Ruby 用のランタイムインターフェースクライアントが含まれており、イメージに Lambda との互換性を持たせています。あるいは、ランタイムインターフェースクライアントをアプリケーションの Gemfile に追加することもできます。

   - FROM プロパティに Ruby ベースイメージを設定します。
   - 関数コードのディレクトリを作成し、そのディレクトリを指す環境変数を作成します。この例では、ディレクトリは /var/task であり、Lambda 実行環境をミラーリングします。ただ

し、Dockerfile は AWS ベースイメージを使用しないため、関数コードには任意のディレクトリを選択できます。

- ENTRYPOINT を、Docker コンテナの起動時に実行させるモジュールに設定します。この場合、モジュールはランタイムインターフェイスクライアントです。
- CMD 引数を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する `--platform linux/amd64` オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに `--platform linux/arm64` オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。エミュレーターはイメージに組み込むことも、次の手順を使用してローカルマシンにインストールすることもできます。

ローカルマシンにランタイムインターフェイスエミュレーターをインストールして実行するには

1. プロジェクトディレクトリから次のコマンドを実行して、GitHub からランタイムインターフェイスエミュレーター (x86-64 アーキテクチャ) をダウンロードし、ローカルマシンにインストールします。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   arm64 エミュレータをインストールするには、前のコマンドの GitHub リポジトリ URL を次のように置き換えます。

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

arm64 エミュレーターをインストールするには、$downloadLink を次のように置き換えます。

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。

   - docker-image はイメージ名、test はタグです。
   - aws_lambda_ric lambda_function.LambdaFunction::Handler.process は ENTRYPOINT で、その後に Dockerfile の CMD が続きます。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           aws_lambda_ric lambda_function.LambdaFunction::Handler.process
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
   ```

```
    aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、--platform linux/*amd64* の代わりに --platform linux/*arm64* オプションを使用してください。

3. イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の curl コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の Invoke-WebRequest コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. コンテナ ID を取得します。

```
docker ps
```

5. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。
   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず `:latest` を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda

は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、`update-function-code` コマンドを使用する必要があります。

Ruby の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティを提供します。

context メソッド

- `get_remaining_time_in_millis` — 実行がタイムアウトするまでの残り時間をミリ秒で返します。

context プロパティ

- `function_name` - Lambda 関数の名前。
- `function_version` - 関数のバージョン。
- `invoked_function_arn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `memory_limit_in_mb` - 関数に割り当てられたメモリの量。
- `aws_request_id` - 呼び出しリクエストの ID。
- `log_group_name` - 関数のロググループ。
- `log_stream_name` — 関数インスタンスのログストリーム。
- `deadline_ms` - 関数がタイムアウトした日付 (Unix 時間のミリ秒)。
- `identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
- `client_context` — (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。

Ruby の AWS Lambda 関数ログ作成

AWS Lambda は、ユーザーに代わって Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda ランタイム環境は、各呼び出しの詳細をログストリームに送信し、関数のコードからのログやその他の出力を中継します。詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

このページでは、AWS Command Line Interface、Lambda コンソール、または CloudWatch コンソールを使用して、Lambda 関数のコードからログ出力を生成する方法、またはアクセスログを生成する方法について説明します。

セクション

ログを返す関数の作成

関数コードからログを出力するには、`puts` ステートメントか、`stdout` または `stderr` に書き込むログ記録ライブラリを使用します。次の例では、環境変数の値とイベントオブジェクトをログに記録します。

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example ログの形式

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 16 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Ruby ランタイムは、呼び出しごとに START、END、および REPORT の各行を記録します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- サンプリング済み - トレースされたリクエストの場合、サンプリング結果。

詳細なログについては、the section called “ロガーライブラリ” を使用します。

Lambda コンソールを使用する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/**###**)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。　X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - `aws configure` によるクイック設定

AWS CLI および `--log-type` コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、`LogResult`フィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、`LogResult`という名前の関数の`my-function`フィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能にな

るまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

```
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

ロガーライブラリ

Ruby ロガーライブラリは、読みやすい効率的なログを返します。ロガーユーティリティを使用して、関数に関連する詳細情報、メッセージ、およびエラーコードを出力します。

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

logger からの出力には、ログレベル、タイムスタンプおよびリクエスト ID が含まれています。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 3 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

AWS Lambda での Ruby の作成

Lambda が AWS X-Ray と統合され、Lambda アプリケーションのトレース、デバッグ、および最適化が可能になりました。X-Ray を使用して、フロントエンド API からバックエンドのストレージとデータベースまで、アプリケーション内のリソースを通過する際にリクエストをトレースできます。X-Ray SDK ライブラリをビルド設定に追加するだけで、関数が AWS のサービスに対して行う呼び出しのエラーとレイテンシーを記録できます。

アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロググループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

コンソールを使用して Lambda 関数のアクティブトレースを切り替えるには、以下のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。
4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。

6. [Save] を選択します。

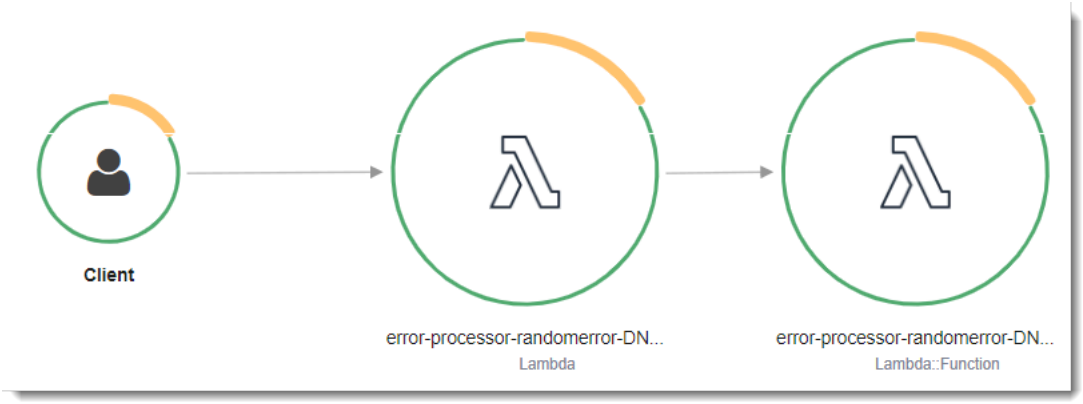
> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

X-Ray は、アプリケーションへのすべてのリクエストをトレースするとは限りません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

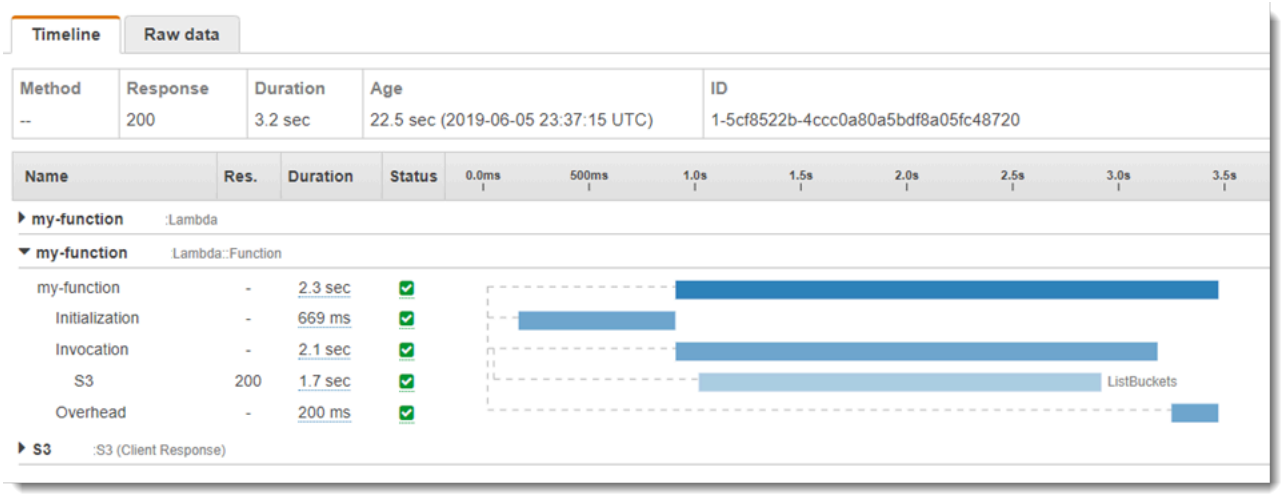
> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグ

メントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは AWS::Lambda の起点があり、もう 1 つは AWS::Lambda::Function の起点があります。AWS::Lambda セグメントにエラーが表示される場合は、Lambda サービスに問題があります。AWS::Lambda::Function セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [Overhead] (オーバーヘッド) - Lambda ランタイムが次のイベントを処理するための準備に費やす時間を表します。

ハンドラーコードを実装して、メタデータを記録し、ダウンストリームコールをトレースできます。ハンドラーが他のリソースやサービスに対して行うコールの詳細を記録するには、X-Ray SDK for Ruby を使用します。SDK を取得するには、アプリケーションの依存関係に aws-xray-sdk パッケージを追加します。

Example blank-ruby/function/Gemfile

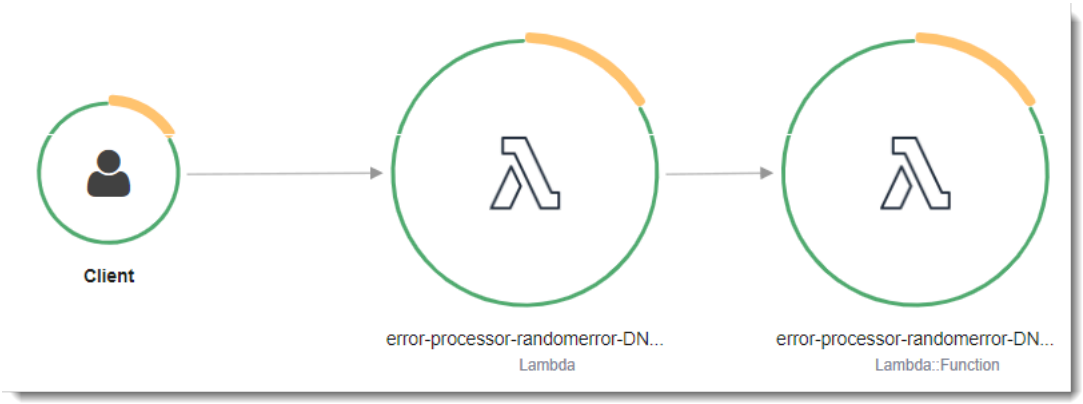
```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

AWS SDK クライアントを実装するには、初期化コードでクライアントを作成した後、`aws-xray-sdk/lambda` モジュールが必要です。

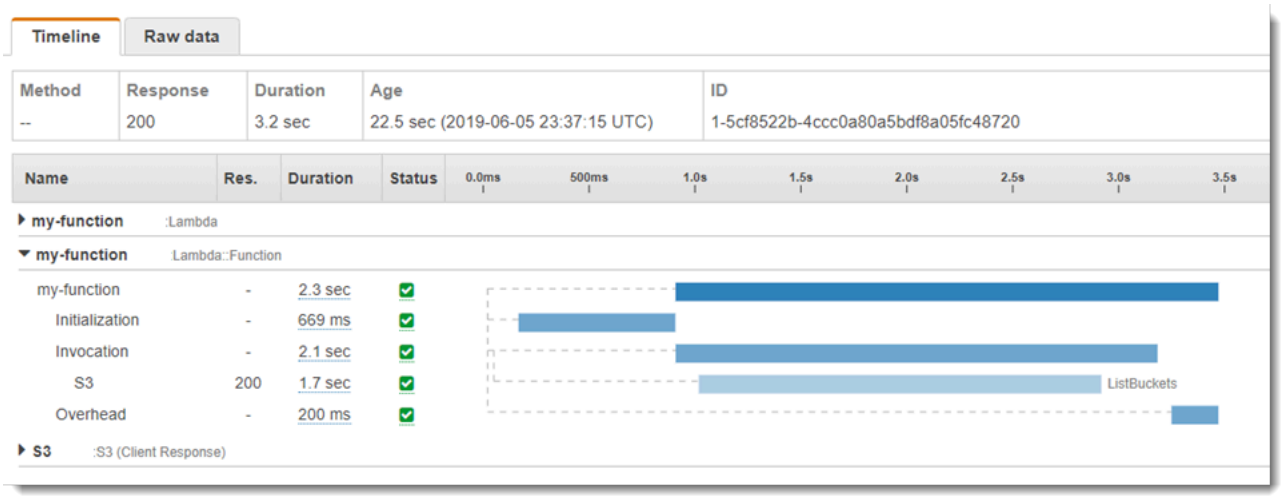
Example blank-ruby/function/lambda_function.rb - AWS SDK クライアントのトレース

```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは `AWS::Lambda` の起点があり、もう 1 つは `AWS::Lambda::Function` の起点があります。`AWS::Lambda` セグメントにエラーが表示される場合は、Lambda サービスに問題があります。`AWS::Lambda::Function` セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、AWS X-Ray デベロッパーガイドの The X-Ray SDK for Ruby を参照してください。

セクション

- Lambda API でのアクティブトレースの有効化
- AWS CloudFormation でのアクティブトレースの有効化
- ランタイム依存関係をレイヤーに保存する

Lambda API でのアクティブトレースの有効化

AWS CLI または AWS SDK を使用してトレース設定を管理するには、以下の API オペレーションを使用します。

- UpdateFunctionConfiguration

- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation でのアクティブトレースの有効化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

ランタイム依存関係をレイヤーに保存する

X-Ray SDK を使用して AWS SDK クライアントを関数コードに埋め込むと、デプロイパッケージが巨大になる可能性があります。機能コードを更新するたびに実行時の依存関係がアップロードされないようにするには、X-Ray SDK を [Lambda layer] (Lambda レイヤー) にパッケージ化します。

以下の例は、X-Ray SDK for Ruby を保存する `AWS::Serverless::LayerVersion` リソースを示しています。

Example template.yml - 依存関係レイヤー

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

この設定では、ランタイム依存関係を変更した場合にのみ、ライブラリレイヤーの更新が必要です。関数のデプロイパッケージにはユーザーのコードのみが含まれるため、アップロード時間を短縮できます。

依存関係のレイヤーを作成するには、デプロイ前にレイヤーアーカイブを生成するようにビルドを変更する必要があります。実際の例については、blank-ruby サンプルアプリケーションを参照してください。

Java による Lambda 関数の構築

Java コードを AWS Lambda で実行できます。Lambda は、コードを実行してイベントを処理する Java 用のランタイムを提供します。コードは、管理している AWS Identity and Access Management (IAM) ロールからの AWS 認証情報を含む Amazon Linux 環境で実行されます。

Lambda は、以下の Java ランタイムをサポートしています。

Java

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
Java 21	`java21`	Amazon Linux 2023			
Java 17	`java17`	Amazon Linux 2			
Java 11	`java11`	Amazon Linux 2			
Java 8	`java8.al2`	Amazon Linux 2			

Lambda には、Java 関数用に以下のライブラリが用意されています。

- com.amazonaws:aws-lambda-java-core (必須) - ランタイムがハンドラに渡すハンドラメソッドインターフェイスとコンテキストオブジェクトを定義します。独自の入力タイプを定義する場合、これが唯一必要なライブラリです。
- com.amazonaws:aws-lambda-java-events - Lambda 関数を呼び出すサービスからのイベントの入力タイプ。
- com.amazonaws:aws-lambda-java-log4j2 - 現在の呼び出しのリクエスト ID を関数ログに追加するために使用できる Apache Log4j 2 のアペンダーライブラリ。
- AWS SDK for Java 2.0 - Java プログラミング言語用の公式の AWS SDK。

> **Important**
>
> プライベートフィールド、メソッド、クラスなどの JDK API のプライベートコンポーネントは使用しないでください。非公開 API コンポーネントは更新時に変更または削除され、アプリケーションが動作しなくなる可能性があります。

Java 関数を作成するには

1. Lambda コンソールを開きます。
2. [Create function] (関数の作成) をクリックします。
3. 以下の設定を行います。
   - [Function name]: 関数名を入力します。
   - [Runtime]: [Java 21] を選択します。
4. [Create function] (関数の作成) をクリックします。
5. テストイベントを設定するには、[テスト] を選択します。
6. [イベント名] で、「**test**」と入力します。
7. [変更を保存] をクリックします。
8. [Test] (テスト) を選択して関数を呼び出します。

コンソールは、`Hello` という名前のハンドラクラスを持つ Lambda 関数を作成します。Java はコンパイルされた言語であるため、Lambda コンソールでソースコードを表示または編集することはできませんが、設定の変更、呼び出し、トリガーの設定を行うことができます。

> **Note**
>
> ローカル環境でアプリケーション開発を開始するには、このガイドの GitHub リポジトリで利用可能なサンプルアプリケーションの 1 つをデプロイします。

`Hello` クラスには、イベントオブジェクトおよびコンテキストオブジェクトを取得する `handleRequest` という名前の関数が含まれています。これは、関数が呼び出されるときに Lambda が呼び出すハンドラー関数です。Java 関数のランタイムは Lambda から呼び出しイベントを取得し、ハンドラに渡します。関数設定で、ハンドラ値は `example.Hello::handleRequest` です。

関数のコードを更新するには、デプロイパッケージを作成します。このパッケージは、関数コードを含む .zip ファイルアーカイブです。関数の開発が進むにつれて、ソース管理への関数コードの保存、ライブラリの追加、デプロイの自動化を行うことがあります。まず、デプロイパッケージを作成し、コマンドラインでコードを更新します。

関数のランタイムによって、呼び出しイベントに加えて、コンテキストオブジェクトがハンドラに渡されます。コンテキストオブジェクトには、呼び出し、関数、および実行環境に関する追加情報が含まれます。詳細情報は、環境変数から入手できます。

Lambda 関数には CloudWatch Logs ロググループが付属しています。関数のランタイムは、各呼び出しに関する詳細を CloudWatch Logs に送信します。これは呼び出し時に、任意の関数が出力するログを中継します。関数がエラーを返す場合、Lambda はエラー形式を整え、それを呼び出し元に返します。

トピック

- Java の Lambda 関数ハンドラーの定義
- .zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする
- コンテナイメージを使用した Java Lambda 関数のデプロイ
- Java Lambda 関数のレイヤーを操作する
- Lambda SnapStart による起動パフォーマンスの向上
- Java Lambda 関数のカスタマイズ設定
- Java の AWS Lambda context オブジェクト
- Java の AWS Lambda 関数ログ作成
- AWS Lambda での Java コードの作成
- AWS Lambda の Java サンプルアプリケーション

Java の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

このガイドの GitHub リポジトリには、さまざまなハンドラータイプを示す、簡単にデプロイできるサンプルアプリケーションが用意されています。詳細については、このトピックの最後を参照してください。

セクション

ハンドラーの例: Java 17 ランタイム

以下の Java 17 の例では、HandlerIntegerJava17 という名前のクラスで handleRequest という名前のハンドラーメソッドを定義しています。ハンドラーメソッドは以下の入力を受け取ります。

- IntegerRecord 。これは、イベントデータを表すカスタム Java レコードです。この例では、IntegerRecord を次のように定義します。

```
record IntegerRecord(int x, int y, String message) {
}
```

- コンテキストオブジェクト。このオブジェクトは、呼び出し、関数、および実行環境に関する情報を示すメソッドおよびプロパティを提供します。

入力 IntegerRecord から message を ログに記録し、x と y の合計を返す関数を記述したいとします。関数コードは次のとおりです。

Example HandlerIntegerJava17.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

// Handler value: example.HandlerInteger
public class HandlerIntegerJava17 implements RequestHandler<IntegerRecord, Integer>{

  @Override
  /*
   * Takes in an InputRecord, which contains two integers and a String.
   * Logs the String, then returns the sum of the two Integers.
   */
  public Integer handleRequest(IntegerRecord event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("String found: " + event.message());
    return event.x() + event.y();
  }
}

record IntegerRecord(int x, int y, String message) {
}
```

関数の設定でハンドラーパラメータを指定することにより、Lambda が呼び出すメソッドを指示します。ハンドラーは次の形式で表現できます。

- *package*.*Class*::*method* - 完全形式。例: example.Handler::handleRequest。
- *package*.*Class* - ハンドラーインターフェイスを実装するクラスの省略形式。例: example.Handler。

Lambda がハンドラーを呼び出すと、Lambda ランタイムはイベントを JSON 形式の文字列として受け取り、オブジェクトに変換します。前の例では、サンプルイベントは次のようになります。

Example event.json

```
{
  "x": 1,
```

```
  "y": 20,
  "message": "Hello World!"
}
```

このファイルを保存し、次の AWS Command Line Interface (CLI) コマンドを使用して関数をローカルでテストできます。

```
aws lambda invoke --function-name function_name --payload file://event.json out.json
```

ハンドラーの例: Java 11 以下のランタイム

Lambda は Java 17 以降のランタイムのレコードをサポートします。すべての Java ランタイムで、クラスを使用してイベントデータを表すことができます。次の例では、整数のリストとコンテキストオブジェクトを入力として受け取り、リスト内のすべての整数の合計を返します。

Example Handler.java

以下の例では、Handler という名前のクラスで handleRequest という名前のハンドラーメソッドを定義しています。ハンドラーメソッドは、イベントとコンテキストオブジェクトを入力として受け取り、文字列を返します。

Example HandlerList.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.List;

// Handler value: example.HandlerList
public class HandlerList implements RequestHandler<List<Integer>, Integer>{

  @Override
  /*
   * Takes a list of Integers and returns its sum.
   */
  public Integer handleRequest(List<Integer> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass().toString());
```

```
    return event.stream().mapToInt(Integer::intValue).sum();
  }
}
```

その他の例については、「サンプルハンドラーコード」を参照してください。

初期化コード

Lambda は、関数を初めて呼び出す前の初期化フェーズで静的コードとクラスコンストラクターを実行します。初期化中に作成されたリソースは、呼び出しと呼び出しの間でメモリ内に保持され、ハンドラーによって何千回も再利用できます。したがって、メインハンドラーメソッドの外側に初期化コードを追加することで、計算時間を節約し、複数の呼び出し間でリソースを再利用することができます。

次の例では、クライアント初期化コードはメインハンドラーメソッドの外にあります。ランタイムは、関数が最初のイベントを処理する前にクライアントを初期化します。Lambda がクライアントを再度初期化する必要がないため、後続のイベントはずっと速くなります。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.GetAccountSettingsResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String> {

    private static final LambdaClient lambdaClient = LambdaClient.builder().build();

    @Override
    public String handleRequest(Map<String,String> event, Context context) {

        LambdaLogger logger = context.getLogger();
        logger.log("Handler invoked");
```

```
        GetAccountSettingsResponse response = null;
        try {
            response = lambdaClient.getAccountSettings();
        } catch(LambdaException e) {
            logger.log(e.getMessage());
        }
        return response != null ? "Total code size for your account is " +
 response.accountLimit().totalCodeSize() + " bytes" : "Error";
    }
}
```

入力タイプと出力タイプの選択

イベントがマッピングするオブジェクトのタイプは、ハンドラーメソッドの署名で指定します。上記の例では、Java ランタイムはイベントを、Map<String,String> インターフェイスを実装するタイプに逆シリアル化します。文字列から文字列へのマッピングは、以下のような階層なしのイベントで機能します。

Example Event.json - 気象データ

```
{
  "temperatureK": 281,
  "windKmh": -3,
  "humidityPct": 0.55,
  "pressureHPa": 1020
}
```

ただし、各フィールドの値は文字列または数値であることが必要です。イベント内のフィールドにオブジェクトが値として含まれている場合、ランタイムはそれを逆シリアル化できず、エラーを返します。

関数が処理するイベントデータに使用する入力タイプを選択します。基本タイプ、汎用タイプ、または良定義タイプを使用できます。

入力タイプ

- Integer Long、Double、など - イベントは、追加の形式のない数値です (3.5 など)。ランタイムは、値を指定されたタイプのオブジェクトに変換します。
- String - イベントは、引用符を含む JSON 文字列です ("My string." など)。ランタイムは、引用符なしの値を String オブジェクトに変換します。

- *Type*、Map<String,*Type*> など - イベントは JSON オブジェクトです。ランタイムは、それを指定されたタイプまたはインターフェイスのオブジェクトに逆シリアル化します。
- List<Integer> List<String>、List<Object>、など - イベントは JSON 配列です。ランタイムは、それを指定されたタイプまたはインターフェイスのオブジェクトに逆シリアル化します。
- InputStream - イベントは任意の JSON タイプです。ランタイムは、ドキュメントのバイトストリームを変更せずにハンドラーに渡します。入力を逆シリアル化し、出力を出力ストリームに書き込みます。
- ライブラリタイプ - AWS サービスによって送信されるイベントの場合、aws-lambda-java-events ライブラリのタイプを使用します。

独自の入力タイプを定義する場合、そのタイプは、逆シリアライズかつ変更可能な Plain Old Java Object (POJO) であり、デフォルトのコンストラクタと、イベントの各フィールドのプロパティが必要です。プロパティにマッピングされないイベントのキー、およびイベントに含まれていないプロパティは、エラーなしでドロップされます。

出力タイプはオブジェクトまたは void です。ランタイムは、戻り値をテキストにシリアル化します。出力が、フィールドを含むオブジェクトの場合、ランタイムは、それを JSON ドキュメントにシリアル化します。出力が、プリミティブ値をラップするタイプの場合、ランタイムは、その値のテキスト表現を返します。

ハンドラーのインターフェイス

aws-lambda-java-core ライブラリは、ハンドラーメソッドの 2 つのインターフェイスを定義します。用意されているインターフェイスを使用して、ハンドラー設定をシンプルにし、コンパイル時にハンドラーメソッドの署名を検証します。

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

RequestHandler インターフェイスは、入力タイプと出力タイプの 2 つのパラメータを受け取る汎用タイプです。どちらのタイプもオブジェクトであることが必要です。このインターフェイスを使用すると、Java ランタイムはイベントを入力タイプのオブジェクトに逆シリアル化し、出力をテキストにシリアル化します。組み込みのシリアル化が入力タイプと出力タイプで機能する場合は、このインターフェイスを使用します。

Example Handler.java - ハンドラーインターフェイス

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
  @Override
  public String handleRequest(Map<String,String> event, Context context)
```

独自のシリアル化を使用するには、RequestStreamHandler インターフェイスを実装します。このインターフェイスでは、Lambda はハンドラーに入力ストリームと出力ストリームを渡します。ハンドラーは、入力ストリームからバイトを読み取り、出力ストリームに書き込み、void を返します。

以下の例では、バッファされたリーダーとライターのタイプを使用して、入力ストリームと出力ストリームを処理しています。

Example ハンドラーストリーム

```
import com.amazonaws.services.lambda.runtime.Context
import com.amazonaws.services.lambda.runtime.LambdaLogger
import com.amazonaws.services.lambda.runtime.RequestStreamHandler
...
// Handler value: example.HandlerStream
public class HandlerStream implements RequestStreamHandler {
  @Override
  /*
   * Takes an InputStream and an OutputStream. Reads from the InputStream,
   * and copies all characters to the OutputStream.
   */
  public void handleRequest(InputStream inputStream, OutputStream outputStream, Context
 context) throws IOException
  {
    LambdaLogger logger = context.getLogger();
    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream,
 Charset.forName("US-ASCII")));
    PrintWriter writer = new PrintWriter(new BufferedWriter(new
 OutputStreamWriter(outputStream, Charset.forName("US-ASCII"))));
    int nextChar;
    try {
      while ((nextChar = reader.read()) != -1) {
        outputStream.write(nextChar);
      }
    } catch (IOException e) {
```

```
      e.printStackTrace();
    } finally {
      reader.close();
      String finalString = writer.toString();
      logger.log("Final string result: " + finalString);
      writer.close();
    }
  }
}
```

サンプルハンドラーコード

このガイドの GitHub リポジトリには、さまざまなハンドラータイプとインターフェイスの使用方法を示すサンプルアプリケーションが含まれています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS SAM テンプレート、サポートリソースが含まれています。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API ゲートウェイを使用して関数を呼び出します。

`java-events` および `s3-java` アプリケーションは、AWS サービスのイベントを入力として受け取り、文字列を返します。`java-basic` アプリケーションには数タイプのハンドラーが含まれています。

- Handler.java - `Map<String,String>` を入力として受け取ります。

- HandlerInteger.java - `Integer` を入力として受け取ります。
- HandlerList.java - `List<Integer>` を入力として受け取ります。
- HandlerStream.java - `InputStream` と `OutputStream` を入力として受け取ります。
- HandlerString.java - `String` を入力として受け取ります。
- HandlerWeatherData.java - カスタムタイプを入力として受け取ります。

さまざまなハンドラータイプをテストするには、AWS SAM テンプレートのハンドラー値を変更するだけです。詳細な手順については、サンプルアプリケーションの readme ファイルを参照してください。

.zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする

AWS Lambda 関数のコードは、スクリプトまたはコンパイルされたプログラム、さらにそれらの依存関係で構成されます。デプロイパッケージを使用して、Lambda に関数コードをデプロイします。Lambda は、コンテナイメージと .zip ファイルアーカイブの 2 種類のデプロイパッケージをサポートします。

このページでは、デプロイパッケージを .zip ファイルまたは Jar ファイルとして作成し、そのデプロイパッケージを使用して、AWS Lambda (AWS Command Line Interface) で関数コードを AWS CLI にデプロイする方法について説明します。

セクション

前提条件

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - `aws configure` によるクイック設定

ツールとライブラリ

Lambda には、Java 関数用に以下のライブラリが用意されています。

- com.amazonaws:aws-lambda-java-core (必須) - ランタイムがハンドラに渡すハンドラメソッドインターフェイスとコンテキストオブジェクトを定義します。独自の入力タイプを定義する場合、これが唯一必要なライブラリです。
- com.amazonaws:aws-lambda-java-events - Lambda 関数を呼び出すサービスからのイベントの入力タイプ。
- com.amazonaws:aws-lambda-java-log4j2 - 現在の呼び出しのリクエスト ID を関数ログに追加するために使用できる Apache Log4j 2 のアペンダーライブラリ。
- AWS SDK for Java 2.0 - Java プログラミング言語用の公式の AWS SDK。

これらのライブラリは Maven Central Repository から入手できます。以下のようにそれらのライブラリをビルド定義に追加します。

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

デプロイパッケージを作成するには、関数コードと依存関係を 1 つの ZIP ファイルまたは Java Archive (JAR) ファイルにコンパイルします。Gradle の場合は、Zip ビルドタイプを使用します。Apache Maven の場合、Maven Shade プラグインを使用します。デプロイパッケージをアップロードするには、Lambda コンソール、Lambda API、AWS Serverless Application Model (AWS SAM) を使用します。

> **Note**
>
> デプロイパッケージのサイズを小さく保つため、関数の依存関係をレイヤーにパッケージ化します。レイヤーを使用すると、依存関係を独立して管理し、複数の関数で使用できます。また、他のアカウントと共有することもできます。詳細については、「Lambda レイヤー」を参照してください。

Gradle を使用したデプロイパッケージのビルド

関数のコードおよび Gradle の依存関係を含むデプロイパッケージを作成するには、`Zip` 構築タイプを使用します。「完全なサンプル build.gradle ファイル」の例を示します。

Example build.gradle - ビルドタスク

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

このビルド設定により、`build/distributions` ディレクトリにデプロイパッケージが作成されます。`into('lib')` ステートメント内で、`jar` タスクはメインクラスを含む jar アーカイブを `lib` という名前のフォルダにアセンブルします。さらに、`configurations.runtimeClassPath` タスクが、依存関係ライブラリをビルドのクラスパスから同じ `lib` フォルダにコピーします。

Example build.gradle - 依存関係

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
```

```
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda では、Unicode のアルファベット順で JAR ファイルをロードします。lib ディレクトリの複数の JAR ファイルに同じクラスが含まれている場合は、最初の JAR ファイルが使用されます。重複するクラスを識別するには、次のシェルスクリプトを使用します。

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

依存関係の Java レイヤーを作成する

> **Note**
>
> Java のようなコンパイル済み言語の関数でレイヤーを使用しても、Python のようなインタープリター言語と同じメリットが得られない場合があります。Java はコンパイル済み言語なので、関数は初期化フェーズで共有アセンブリを手動でメモリに読み込む必要があり、コールドスタート時間が長くなる可能性があります。代わりに、コンパイル時にすべての共有コードを含めて、組み込みコンパイラ最適化機能を活用することをお勧めします。

このセクションでは、依存関係をレイヤーに含める方法について説明します。依存関係をデプロイパッケージに含める方法については、「the section called “Gradle を使用したデプロイパッケージのビルド”」または「the section called “Maven を使用したデプロイパッケージのビルド”」を参照してください。

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の /opt ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には /opt ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

- java/lib (CLASSPATH)

例えば、レイヤーの.zip ファイルの構造は次のようになります。

```
jackson.zip
# java/lib/jackson-core-2.2.3.jar
```

さらに、Lambda は /opt/lib ディレクトリ内のライブラリ、および /opt/bin ディレクトリ内のバイナリを自動的に検出します。Lambda がレイヤーのコンテンツを正しく検出できるように、次の構造でレイヤーを作成することもできます。

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

レイヤーをパッケージ化したら、「the section called “レイヤーの作成と削除”」および「the section called “レイヤーの追加”」を参照してレイヤーの設定を完了してください。

Maven を使用したデプロイパッケージのビルド

Maven でデプロイパッケージをビルドするには、Maven Shade プラグインを使用します。このプラグインは、コンパイルされた関数コードとそのすべての依存関係を含む JAR ファイルを作成します。

Example pom.xml - プラグイン設定

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
```

```
            </goals>
          </execution>
        </executions>
      </plugin>
```

デプロイパッケージをビルドするには、`mvn package` コマンドを使用します。

```
[INFO] Scanning for projects...
[INFO] ----------------------< com.example:java-maven >----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

このコマンドは、`target` ディレクトリ内に JAR ファイルを生成します。

> **Note**
>
> 「マルチリリース JAR (MRJAR)」を使用している場合、を `lib` ディレクトリに MRJAR (つまり、Maven Shade プラグインによって生成されるシェーディング JAR) 含め、デプロイパッケージを Lambda にアップロードする前に圧縮する必要があります。そうしないと、Lambda が JAR ファイルを適切に解凍せず、`MANIFEST.MF` ファイルが無視される可能性があります。

アペンダーライブラリ (aws-lambda-java-log4j2) を使用する場合は、Maven Shade プラグインのトランスフォーマーも設定する必要があります。トランスフォーマーライブラリは、アペンダーライブラリと Log4j の両方のキャッシュファイルのバージョンを組み合わせます。

Example pom.xml - Log4j 2 アペンダーを使用したプラグイン設定

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

Lambda コンソールでデプロイパッケージのアップロード

新しい関数を作成するには、まずコンソールで関数を作成し、次に .zip または JAR ファイルをアップロードする必要があります。既存の関数を更新するには、その関数のページを開き、同じ手順に従って更新した .zip または JAR ファイルを追加します。

デプロイパッケージファイルが 50 MB 未満の場合は、ローカルマシンから直接ファイルをアップロードして関数を作成または更新できます。50 MB を超える .zip または JAR ファイルの場合は、まず Amazon S3 バケットにパッケージをアップロードする必要があります。AWS Management Console を使用して Amazon S3 バケットにファイルをアップロードする手順については、「Amazon S3 の開始方法」を参照してください。AWS CLI を使用してファイルをアップロードするには、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> 既存の関数のデプロイパッケージタイプ (.zip またはコンテナイメージ) を変更することはできません。例えば、既存のコンテナイメージ関数を、.zip ファイルアーカイブを使用するように変換することはできません。この場合は、新しい関数を作成する必要があります。

新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [一から作成] を選択します。
3. [基本的な情報] で、以下を行います。
   a. [関数名] に、関数名を入力します。
   b. [ランタイム] で、使用するランタイムを選択します。
   c. (オプション) [アーキテクチャ]で、関数の命令セットアーキテクチャを選択します。デフォルトのアーキテクチャは x86_64 です。関数用の .zip デプロイパッケージと選択した命令セットのアーキテクチャに互換性があることを確認してください。
4. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。新しい [実行ロール] を作成することも、既存のロールを使用することもできます。
5. [関数の作成]を選択します。Lambda は、選択したランタイムを使用して基本的な「Hello world」関数を作成します。

ローカルマシンから .zip または JAR アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、.zip または JAR ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード] をクリックします。

4. .zip または.jar ファイルを選択します。
5. .zip または JAR ファイルをアップロードするには、次の操作を行います。
   a. [アップロード] をクリックし、ファイルセレクターで .zip または JAR ファイルを選択します。
   b. [開く] をクリックします。
   c. [保存] をクリックします。

Amazon S3 バケットから .zip または JAR アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、新しい .zip または JAR ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [Amazon S3 ロケーション] を選択します。
5. .zip ファイルの Amazon S3 リンク URL を貼り付けて、[保存] をクリックします。

AWS CLI を使用したデプロイパッケージのアップロード

AWS CLI を使用して新しい関数を作成したり、.zip または JAR ファイルを使用して既存の関数を更新したりできます。create-function コマンドと update-function-code を使用して、.zip または JAR パッケージをデプロイします。ファイルが 50 MB 未満の場合は、ローカルビルドマシン上のファイルの場所からパッケージをアップロードできます。サイズの大きいファイルの場合は、Amazon S3 バケットから .zip または JAR パッケージをアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> AWS CLI を使用して Amazon S3 バケットから .zip または JAR ファイルをアップロードする場合、このバケットは関数と同じ AWS リージョン に配置する必要があります。

AWS CLI を含む .zip または JAR ファイルを使用して新しい関数を作成するには、以下を指定する必要があります。

- 関数の名前 (--function-name)
- 関数のランタイム (--runtime)
- 関数の実行ロール (--role) の Amazon リソースネーム (ARN)
- 関数コード内のハンドラーメソッド (--handler) の名前

.zip ファイルの場所も指定する必要があります。.zip または JAR ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --code オプションを使用します。S3ObjectVersion パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda create-function --function-name myFunction \
--runtime java21 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

CLI を使用して既存の関数を更新するには、--function-name パラメータを使用して関数の名前を指定します。関数コードの更新に使用する .zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --s3-bucket および --s3-key オプションを使用します。--s3-object-version パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

AWS SAM によるデプロイパッケージのアップロード

AWS SAM を使用して、関数コード、設定、依存関係のデプロイを自動化できます。AWS SAM は AWS CloudFormation の拡張であり、サーバーレスアプリケーションを定義するための構文が簡略化されています。以下のサンプルテンプレートは、Gradle 用の `build/distributions` ディレクトリにあるデプロイパッケージを使用する関数を定義します。

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java21
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

関数を作成するには、`package` コマンドと `deploy` コマンドを使用します。これらのコマンドは AWS CLI に合わせてカスタマイズされています。他のコマンドをラップして、デプロイパッケージを Amazon S3 にアップロードし、オブジェクト URI でテンプレートを書き換え、関数のコードを更新します。

以下のサンプルスクリプトは、Gradle ビルドを実行し、作成されたデプロイパッケージをアップロードします。また、初回実行時に AWS CloudFormation スタックを作成します。そのスタックがすでに存在する場合は更新します。

Example deploy.sh

```
#!/bin/bash
```

```
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

完全な使用例については、以下のサンプルアプリケーションを参照してください。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API ゲートウェイを使用して関数を呼び出します。

コンテナイメージを使用した Java Lambda 関数のデプロイ

Java Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- Java の AWS ベースイメージを使用する

  AWS ベースイメージには、言語ランタイム、Lambda と関数コード間のやり取りを管理するランタイムインターフェースクライアント、ローカルテスト用のランタイムインターフェイスエミュレーターがプリロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、Java のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することもできます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、Java のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

トピック

- Java の AWS ベースイメージ

- Java の AWS ベースイメージを使用する
- ランタイムインターフェイスクライアントで代替ベースイメージを使用する

Java の AWS ベースイメージ

AWS は、Java の次のベースイメージを提供します。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
21	Java 21	Amazon Linux 2023	GitHub の Java 21 用の Dockerfile	
17	Java 17	Amazon Linux 2	GitHub の Java 17 用の Dockerfile	
11	Java 11	Amazon Linux 2	GitHub の Java 11 用の Dockerfile	
8.al2	Java 8	Amazon Linux 2	GitHub の Java 8 用の Dockerfile	

Amazon ECR リポジトリ: gallery.ecr.aws/lambda/java

Java 21 以降のベースイメージは、Amazon Linux 2023 の最小コンテナイメージに基づいています。以前のベースイメージでは Amazon Linux 2 が使用されています。AL2023 ランタイムには、デプロイのフットプリントが小さいことや、`glibc` などのライブラリのバージョンが更新されていることなど、Amazon Linux 2 に比べていくつかの利点があります。

AL2023 ベースのイメージでは、Amazon Linux 2 のデフォルトのパッケージマネージャである `yum` の代わりに `microdnf` (`dnf` としてシンボリックリンク) がパッケージマネージャとして使用されています。`microdnf` は `dnf` のスタンドアロン実装です。AL2023 ベースのイメージに含まれるパッケージのリストについては、「Comparing packages installed on Amazon Linux 2023 Container Images」の「Minimal Container」列を参照してください。AL2023 と Amazon Linux 2 の違いの詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。

> **Note**
>
> AWS Serverless Application Model (AWS SAM) を含む AL2023 ベースのイメージをローカルで実行するには、Docker バージョン 20.10.10 以降を使用する必要があります。

Java の AWS ベースイメージを使用する

前提条件

このセクションの手順を完了するには、以下が必要です。

- Java (たとえば、「Amazon Corretto」)
- Docker (Java 21 以降のベースイメージの最小バージョンは 20.10.10)
- 「Apache Maven」または「Gradle」
- AWS Command Line Interface (AWS CLI) バージョン 2

ベースイメージからイメージを作成する

Maven

1. 次のコマンドを実行し、「Lambda のアーキタイプ」を使用して Maven プロジェクトを作成します。以下のパラメータは必須です。

   - サービス — Lambda 関数の使用対象となる AWS のサービス クライアント。利用可能なソースのリストについては、GitHub の「aws-sdk-java-v2/services」をご覧ください。
   - region – Lambda 関数を作成する AWS リージョン。
   - groupId — アプリケーションの完全パッケージの名前空間。
   - artifactId — ユーザーのプロジェクト名。これがプロジェクトのディレクトリの名前になります。

   Linux および macOS では、次のコマンドを実行します。

```
mvn -B archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
   -DgroupId=com.example.myapp \
```

```
    -DartifactId=myapp
```

PowerShell で、次のコマンドを実行します。

```
mvn -B archetype:generate `
    "-DarchetypeGroupId=software.amazon.awssdk" `
    "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
 `
    "-DgroupId=com.example.myapp" `
    "-DartifactId=myapp"
```

Lambda の Maven アーキタイプは、Java SE 8 でコンパイルするように事前設定されており、AWS SDK for Java への依存関係を含みます。別のアーキタイプまたは別の方法でプロジェクトを作成する場合、Maven の Java コンパイラを設定し、SDK を依存関係として宣言する必要があります。

2. *myapp*/src/main/java/com/example/*myapp* ディレクトリを開いて App.java ファイルを探します。これは Lambda 関数のコードです。提供されるサンプルコードをテストに使用することも、独自のサンプルコードで置き換えることもできます。
3. プロジェクトのルートディレクトリに戻り、次の設定で新しい Dockerfile を作成します。
   - FROM プロパティにベースイメージの URI を設定します。
   - CMD 引数を Lambda 関数ハンドラーに設定します。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Maven layout
COPY target/classes ${LAMBDA_TASK_ROOT}
COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.myapp.App::handleRequest" ]
```

4. プロジェクトをコンパイルしてランタイムの依存関係を収集します。

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

Gradle

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir example
cd example
```

2. 次のコマンドを実行して、Gradle が 環境の example ディレクトリに新しい Java アプリケーションプロジェクトを生成するようにします。[ビルドスクリプト DSL を選択] では、[2: Groovy] を選択します。

```
gradle init --type java-application
```

3. */example*/app/src/main/java/*example* ディレクトリを開いて App.java ファイルを探します。これは Lambda 関数のコードです。次のサンプルコードをテストに使用することも、独自のサンプルコードで置き換えることもできます。

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
```

```
    }
}
```

4. build.gradle ファイルを開きます。前のステップのサンプル関数コードを使用している場合、「build.gradle」の内容を次の内容で置き換えてください。独自の関数コードを使用している場合、必要に応じて「build.gradle」ファイルを変更してください。

   Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. また。ステップ 2 の gradle init コマンドが app/test ディレクトリにダミーのテストケースを生成しました。このチュートリアルの用途のため、/test ディレクトリを削除してテスト実行をスキップします。
6. プロジェクトをビルドします。

```
gradle build
```

7. プロジェクトのルートディレクトリ (/example) に、次の設定で Dockerfile を作成します。

   - FROM プロパティを「ベースイメージの URI」に設定します。
   - COPY コマンドを使用し、関数コードおよびランタイムの依存関係を {LAMBDA_TASK_ROOT} (Lambda 定義の環境変数) にコピーします。
   - CMD 引数を Lambda 関数ハンドラーに設定します。

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

1. docker run コマンドを使用して、Docker イメージを起動します。この例では、docker-image はイメージ名、test はタグです。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、`--platform linux/amd64` の代わりに `--platform linux/arm64` オプションを使用してください。

2. 新しいターミナルウィンドウから、イベントをローカルエンドポイントにポストします。

   Linux/macOS

   Linux および macOS では、次の `curl` コマンドを実行します。

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   PowerShell で次の `Invoke-WebRequest` コマンドを実行します。

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - --region 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
```

```
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   例：

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず `:latest` を含めてください。

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。`ImageUri` には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず `:latest` を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

ランタイムインターフェイスクライアントで代替ベースイメージを使用する

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間の相互作用を管理する Lambda Runtime API を拡張します。

Java 用のランタイムインターフェイスクライアントを Dockerfile にインストールするか、プロジェクトの依存関係としてインストールします。たとえば、Maven パッケージマネージャーを使用してランタイムインターフェイスクライアントをインストールするには、pom.xml ファイルに以下を追加します。

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

パッケージの詳細については、「Maven Central Repository」の「AWS Lambda Java ランタイムのインターフェイスクライアント」を参照してください。GitHub リポジトリの AWS Lambda Java Support Libraries で、ランタイムインターフェイスクライアントのソースコードを確認することもできます。

次の例は、Amazon Corretto イメージを使用して Java 用のコンテナイメージを構築する方法を示しています。Amazon Corretto は、Open Java Development Kit (OpenJDK) の、マルチプラットフォーム対応の本番稼働可能な、無償ディストリビューションです。Maven プロジェクトには、ランタイムインターフェイスクライアントが依存関係として含まれています。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Java (たとえば、「Amazon Corretto」)
- Docker
- Apache Maven
- AWS Command Line Interface (AWS CLI) バージョン 2

代替ベースイメージからイメージを作成する

1. Maven プロジェクトの作成 以下のパラメータは必須です。

   - groupId — アプリケーションの完全パッケージの名前空間。
   - artifactId — ユーザーのプロジェクト名。これがプロジェクトのディレクトリの名前になります。

   Linux/macOS

   ```
   mvn -B archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DgroupId=example \
      -DartifactId=myapp \
      -DinteractiveMode=false
   ```

   PowerShell

   ```
   mvn -B archetype:generate `
      -DarchetypeArtifactId=maven-archetype-quickstart `
      -DgroupId=example `
      -DartifactId=myapp `
      -DinteractiveMode=false
   ```

2. プロジェクトディレクトリを開きます。

   ```
   cd myapp
   ```

3. pom.xml ファイルを開き、内容を次に置き換えます。このファイルには、aws-lambda-java-runtime-interface-client が依存関係として含まれています。代わりに、ランタイムインターフェイスクライアントを Dockerfile にインストールすることもできます。ただし、最も簡単な方法は、ライブラリを依存関係として含めることです。

   ```
   <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
   www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
   maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>example</groupId>
     <artifactId>hello-lambda</artifactId>
   ```

```
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. *myapp*/src/main/java/com/example/*myapp* ディレクトリを開いて App.java ファイルを探します。これは Lambda 関数のコードです。このコードを、次のコードで置き換えます。

   Example 関数ハンドラ

```
package example;

public class App {
```

```
    public static String sayHello() {
        return "Hello world!";
    }
}
```

5. ステップ 1 の `mvn -B archetype:generate` コマンドにより、ダミーのテストケースも `src/test` ディレクトリに生成されます。このチュートリアルでは、この生成された `/test` ディレクトリ全体を削除して、実行中のテストをスキップします。
6. プロジェクトのルートディレクトリに戻り、新しい Dockerfile を作成します。次の Dockerfile の例では、Amazon Corretto イメージを使用しています。Amazon Corretto は、マルチプラットフォーム対応で本番稼働可能な OpenJDK の無償ディストリビューションです。

   - ベースイメージの URI に `FROM` プロパティを設定します。
   - `ENTRYPOINT` を、Docker コンテナの起動時に実行させるモジュールに設定します。この場合、モジュールはランタイムインターフェイスクライアントです。
   - `CMD` 引数を Lambda 関数ハンドラーに設定します。

   Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function

COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./
```

```
# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。エミュレーターはイメージに組み込むことも、次の手順を使用してローカルマシンにインストールすることもできます。

ローカルマシンにランタイムインターフェイスエミュレーターをインストールして実行するには

1. プロジェクトディレクトリから次のコマンドを実行して、GitHub からランタイムインターフェイスエミュレーター (x86-64 アーキテクチャ) をダウンロードし、ローカルマシンにインストールします。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

arm64 エミュレータをインストールするには、前のコマンドの GitHub リポジトリ URL を次のように置き換えます。

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

arm64 エミュレーターをインストールするには、$downloadLink を次のように置き換えます。

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。

   - docker-image はイメージ名、test はタグです。
   - /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello は ENTRYPOINT で、その後に Dockerfile の CMD が続きます。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
```

```
        /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/bin/java -cp './*'
 com.amazonaws.services.lambda.runtime.api.client.AWSLambda
 example.App::sayHello
```

このコマンドはイメージをコンテナとして実行し、`localhost:9000/2015-03-31/functions/function/invocations` でローカルエンドポイントを作成します。

> **Note**
>
> ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、`--platform linux/amd64` の代わりに `--platform linux/arm64` オプションを使用してください。

3. イベントをローカルエンドポイントにポストします。

Linux/macOS

Linux および macOS では、次の `curl` コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の `Invoke-WebRequest` コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. コンテナ ID を取得します。

```
docker ps
```

5. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。
   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

Java Lambda 関数のレイヤーを操作する

Lambda レイヤー は、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。レイヤーの作成には、次の 3 つの一般的な手順が含まれます。

1. レイヤーコンテンツのパッケージ化。これは、関数で使用する依存関係を含む .zip ファイルアーカイブを作成することを意味します。
2. Lambda でレイヤーを作成します。
3. レイヤーを関数に追加します。

このトピックには、外部ライブラリの依存関係を持つ Java Lambda レイヤーを適切にパッケージ化して作成する方法に関する手順およびガイダンスが含まれています。

トピック

前提条件

このセクションの手順を完了するには、次の事項が必要です。

- Java 21
- Apache Maven バージョン 3.8.6 以降
- AWS Command Line Interface (AWS CLI) バージョン 2

> **Note**
>
> Maven が参照する Java バージョンが、デプロイする関数の Java バージョンと同じであることを確認してください。例えば、Java 21 関数の場合、`mvn -v` コマンドは出力に Java バージョン 21 をリストする必要があります。

```
Apache Maven 3.8.6
...
Java version: 21.0.2, vendor: Oracle Corporation, runtime: /Library/Java/
JavaVirtualMachines/jdk-21.jdk/Contents/Home
...
```

このトピック全体では、awsdocs GitHub リポジトリの「layer-java」サンプルアプリケーションを参照します。このアプリケーションには、依存関係をダウンロードしてレイヤーを生成するスクリプトが含まれています。アプリケーションには、レイヤーから依存関係を使用する対応の関数も含まれています。レイヤーを作成したら、対応する関数をデプロイして呼び出し、すべてが正しく動作することを確認できます。関数に Java 21 ランタイムを使用するため、レイヤーは Java 21 と互換性がある必要もあります。

layer-java サンプルアプリケーションには、2 つのサブディレクトリ内に 1 つの例が含まれています。layer ディレクトリには、レイヤーの依存関係を定義する pom.xml ファイルに加え、レイヤーを生成するスクリプトが含まれています。function ディレクトリには、レイヤーが動作するテストを支援するサンプル関数が含まれています。このチュートリアルは、このレイヤーを作成してパッケージ化する方法について説明します。

Java レイヤーと Amazon Linux の互換性

レイヤーを作成する最初のステップは、すべてのレイヤーコンテンツを .zip ファイルアーカイブにバンドルすることです。Lambda 関数は Amazon Linux 上で実行されるため、レイヤーコンテンツは Linux 環境でコンパイルおよびビルドできる必要があります。

Java コードはプラットフォームに依存しないように設計されているため、ローカルマシンが Linux 環境を使用しなくてもレイヤーをパッケージ化できます。Lambda に Java レイヤーをアップロードした後も、Amazon Linux との互換性は維持されます。

Java ランタイムのレイヤーパス

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の /opt ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には /opt ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

- java/lib

例えば、このチュートリアルで作成するレイヤーの .zip ファイルは、次のディレクトリ構造になっています。

```
layer_content.zip
# java
    # lib
        # layer-java-layer-1.0-SNAPSHOT.jar
```

`layer-java-layer-1.0-SNAPSHOT.jar` JAR ファイル (必要な依存関係をすべて含む uber-jar) は、`java/lib` ディレクトリに正しく配置されています。Lambda は関数の呼び出し中にライブラリを見つけるようになります。

レイヤーコンテンツのパッケージ化

この例では、次の 2 つの Java ライブラリを 1 つの JAR ファイルにパッケージ化します。

- 「aws-lambda-java-core」 — AWS Lambda で Java を使用するためのインターフェイス定義の最小限セット
- 「Jackson」 — 特に JSON を使用するための一般的なデータ処理ツールのスイート。

レイヤーコンテンツをインストールしてパッケージ化するには、次の手順を実行します。

レイヤーコンテンツをインストールしてパッケージ化する方法

1. `sample-apps/layer-java` ディレクトリで必要なサンプルコードを含む「aws-lambda-developer-guide GitHub リポジトリ」を複製します。

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. `layer-java` サンプルアプリの `layer` ディレクトリに移動します。このディレクトリには、レイヤーを適切に作成してパッケージ化するために使用するスクリプトが含まれています。

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-java/layer
   ```

3. pom.xml ファイルを検証する <dependencies> セクションでは、レイヤーに含める依存関係を定義します(つまり、`aws-lambda-java-core` および `jackson-databind` ライブラリ)。このファイルを更新し、独自のレイヤーに含める依存関係を含めることができます。

Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-lambda-java-core</artifactId>
        <version>1.2.3</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>
```

> **Note**
>
> この pom.xml ファイルの <build> セクションには 2 つのプラグインが含まれています。「maven-compiler-plugin」はソースコードをコンパイルします。「maven-shade-plugin」はアーティファクトを 1 つの uber-jar にパッケージ化します。

4. 両方のスクリプトを実行する許可があることを確認してください。

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 次のコマンドを使用して「1-install.sh」スクリプトを実行します。

```
./1-install.sh
```

このスクリプトは現在のディレクトリで mvn clean install を実行します。必要な依存関係をすべて含む uber-jar が target/ ディレクトリに作成されます。

Example 1-install.sh

```
mvn clean install
```

6. 次のコマンドを使用して「2-package.sh」スクリプトを実行します。

```
./2-package.sh
```

このスクリプトは、レイヤーコンテンツを適切にパッケージングするために必要な java/lib ディレクトリ構造を作成します。次に、uber-jar を /target ディレクトリから新しく作成された java/lib ディレクトリにコピーします。最後に、スクリプトは java ディレクトリの内容を layer_content.zip という名前のファイルに圧縮します。これはレイヤーの .zip ファイルです。ファイルを解凍し、the section called “Java ランタイムのレイヤーパス” セクションで示されている正しいファイル構造が含まれていることを確認できます。

Example 2-package.sh

```
mkdir java
mkdir java/lib
cp -r target/layer-java-layer-1.0-SNAPSHOT.jar java/lib/
zip -r layer_content.zip java
```

レイヤーを作成する

このセクションでは、前のセクションで生成した「layer_content.zip」ファイルを取得し、Lambda レイヤーとしてアップロードします。AWS Command Line Interface (AWS CLI) を介して AWS Management Consoleまたは Lambda API を使用してレイヤーをアップロードできます。レイヤーの .zip ファイルをアップロードするとき、次の「PublishLayerVersion」 AWS CLI コマンドで java21 を互換性のあるランタイムとして指定し、arm64 を互換性のあるアーキテクチャとして指定します。

```
aws lambda publish-layer-version --layer-name java-jackson-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes java21 \
    --compatible-architectures "arm64"
```

レスポンスでは、arn:aws:lambda:us-east-1:*123456789012*:layer:java-jackson-layer:1 に似ている LayerVersionArn に注目してください。この Amazon リソースネーム (ARN) は、このチュートリアルの次の手順で、レイヤーを関数に追加するときに必要になります。

レイヤーを関数に追加する

このセクションでは、関数コードで Jackson ライブラリを使用する Lambda 関数の例をデプロイしたら、レイヤーをアタッチします。関数をデプロイするには、the section called “実行ロール (関数に対する、他のリソースにアクセスするためのアクセス許可)” が必要です。既存の実行ロールをお持ちでない場合、折りたたみ可能なセクションの手順を実行してください。それ以外の場合、次の手順に進んで関数をデプロイします。

(オプション) 実行ロールを作成する

実行ロールを作成する

1. IAM コンソールの [ロールページ] を開きます。
2. [ロールの作成] を選択します。
3. 次のプロパティでロールを作成します。

   - 信頼されたエンティティ – Lambda。
   - アクセス許可 – AWSLambdaBasicExecutionRole。
   - ロール名 – **`lambda-role`**。

   AWSLambdaBasicExecutionRole ポリシーには、ログを CloudWatch Logs に書き込むために関数が必要とするアクセス許可があります。

Lambda 関数をデプロイする方法

1. `function/` ディレクトリに移動します。現在、`layer/` ディレクトリにいる場合、次のコマンドを実行します。

   ```
   cd ../function
   ```

2. 「関数コード」を確認します。関数は `Map<String, String>` を入力値として取り込み、Jackson を使用して入力値を JSON 文字列として書き込んでから、事前定義された「F1Car」の Java オブジェクトに変換します。最後に、関数は F1Car オブジェクトのフィールドを使用し、関数が返す文字列を作成します。

   ```
   package example;

   import com.amazonaws.services.lambda.runtime.Context;
   ```

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;
import java.util.Map;

public class Handler {

    public String handleRequest(Map<String, String> input, Context context) throws
 IOException {
        // Parse the input JSON
        ObjectMapper objectMapper = new ObjectMapper();
        F1Car f1Car =
 objectMapper.readValue(objectMapper.writeValueAsString(input), F1Car.class);

        StringBuilder finalString = new StringBuilder();
        finalString.append(f1Car.getDriver());
        finalString.append(" is a driver for team ");
        finalString.append(f1Car.getTeam());
        return finalString.toString();
    }
}
```

3. 次の Maven コマンドを実行し、プロジェクトをビルドします。

```
mvn package
```

このコマンドは、layer-java-function-1.0-SNAPSHOT.jar という名前の target/ ディレクトリの JAR ファイルを生成します。

4. 関数をデプロイします。次の AWS CLI コマンドで、--role パラメータを実行ロール ARN に置き換えます。

```
aws lambda create-function --function-name java_function_with_layer \
    --runtime java21 \
    --architectures "arm64" \
    --handler example.Handler::handleRequest \
    --timeout 30 \
    --role arn:aws:iam::123456789012:role/lambda-role \
    --zip-file fileb://target/layer-java-function-1.0-SNAPSHOT.jar
```

(オプション) レイヤーをアタッチせずに関数を呼び出す

この時点では、レイヤーをアタッチする前に関数の呼び出しを試みることもできます。これを試してみると、関数が `requests` パッケージを参照できないため、`ClassNotFoundException` と表示されます。関数を呼び出すには、次の AWS CLI コマンドを使用します。

```
aws lambda invoke --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "driver": "Max Verstappen", "team": "Red Bull" }' response.json
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "FunctionError": "Unhandled",
    "ExecutedVersion": "$LATEST"
}
```

特定のエラーを表示するには、出力 `response.json` ファイルを開きます。次のエラーメッセージを含む `ClassNotFoundException` が表示されます。

```
"errorMessage":"com.fasterxml.jackson.databind.ObjectMapper","errorType":"java.lang.ClassNotFou
```

次に、レイヤーを関数にアタッチします。次の AWS CLI コマンドで、`--layers` パラメーターを先にメモしたレイヤーバージョン ARN に置き換えます。

```
aws lambda update-function-configuration --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --layers "arn:aws:lambda:us-east-1:123456789012:layer:java-jackson-layer:1"
```

最後に、次の AWS CLI コマンドを使用して関数の呼び出しを試みます。

```
aws lambda invoke --function-name java_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "driver": "Max Verstappen", "team": "Red Bull" }' response.json
```

次のような出力が表示されます。

```
{
```

```
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

関数が Jackson の依存関係を使用し、関数を適切に実行できたことを示しています。出力 `response.json` ファイルに返された正しい文字列が含まれていることを確認できます。

```
"Max Verstappen is a driver for team Red Bull"
```

(オプション) リソースをクリーンアップする

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda レイヤーを削除する方法

1. Lambda コンソールの [Layers (レイヤー)] ページを開きます。
2. 作成したレイヤーを選択します。
3. [削除] を選択したら、[削除] を再度選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[Delete] (削除) を選択します。

Lambda SnapStart による起動パフォーマンスの向上

Lambda SnapStart for Java は、レイテンシーの影響を受けやすいアプリケーションの起動パフォーマンスを追加のコストなしで最大 10 倍向上させることができ、通常は関数コードの料金もかかりません。起動時のレイテンシー (多くの場合、コールドスタート時間と呼ばれます) の最大の原因は、Lambda が関数の初期化に費やす時間で、これには関数のコードのロード、ランタイムの開始、および関数コードの初期化が含まれます。

SnapStart を使用すると、Lambda は関数バージョンを発行するときに関数を初期化します。Lambda は、初期化された実行環境のメモリとディスク状態の Firecracker MicroVM スナップショットを作成し、そのスナップショットを暗号化して、低レイテンシーアクセスのためにキャッシュします。関数バージョンを初めて呼び出し、呼び出しがスケールアップすると、Lambda はそれらをゼロから初期化するのではなく、キャッシュされたスナップショットから新しい実行環境を再開するので、起動時のレイテンシーが短縮されます。

> **Important**
>
> アプリケーションが状態の一意性に依存する場合は、関数コードを評価して、スナップショット操作に対する耐性があることを確認する必要があります。詳細については、「Lambda での一意性の処理 SnapStart」を参照してください。

トピック

- Lambda SnapStart パフォーマンスの最大化

サポートされている機能と制限事項

SnapStart は、Java 11 以降の Java マネージドランタイムをサポートします。他のマネージドランタイム (`nodejs20.x` や `python3.12` など)、OS 専用ランタイム、およびコンテナイメージはサポートされていません。

SnapStart は、プロビジョニングされた同時実行、arm64 アーキテクチャ、Amazon Elastic File System (Amazon EFS)、512 MB を超えるエフェメラルストレージをサポートしません。

SnapStart を使用する際、Lambda コンソール、AWS Command Line Interface (AWS CLI)、Lambda API、AWS SDK for Java、AWS CloudFormation、AWS Serverless Application Model (AWS SAM)、および AWS Cloud Development Kit (AWS CDK) を使用できます。詳細については、「Lambda のアクティブ化と管理 SnapStart」を参照してください。

> **Note**
>
> SnapStart を使用できるのは、発行済みの関数バージョンと、バージョンをポイントするエイリアスのみです。関数の未発行バージョン ($LATEST) で SnapStart を使用することはできません。

サポートされるリージョン

SnapStart は、以下の AWS リージョンで利用できます。

- 米国東部 (バージニア北部)
- 米国東部 (オハイオ)
- 米国西部 (北カリフォルニア)
- 米国西部 (オレゴン)
- アフリカ (ケープタウン)
- アジアパシフィック (香港)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (ハイデラバード)
- アジアパシフィック (東京)

- アジアパシフィック (ソウル)
- アジアパシフィック (大阪)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- アジアパシフィック (ジャカルタ)
- アジアパシフィック (メルボルン)
- カナダ (中部)
- 欧州 (ストックホルム)
- 欧州 (フランクフルト)
- 欧州 (チューリッヒ)
- 欧州 (アイルランド)
- 欧州 (ロンドン)
- 欧州 (パリ)
- 欧州 (ミラノ)
- 欧州 (スペイン)
- 中東 (アラブ首長国連邦)
- 中東 (バーレーン)
- 南米 (サンパウロ)

互換性に関する考慮事項

SnapStart では、Lambda が複数の実行環境の初期状態として単一のスナップショットを使用します。関数が初期化フェーズで以下のいずれかを使用する場合は、SnapStart を使用する前にいくつかの変更を行う必要がある場合があります。

一意性

スナップショットに包含される一意のコンテンツを初期化コードが生成する場合、そのコンテンツは、複数の実行環境で再利用されるときに一意にならない可能性があります。SnapStart の使用時に一意性を維持するには、初期化後に一意のコンテンツを生成する必要があります。これには、一意の ID、一意のシークレット、および疑似ランダム性を生成するために使用されるエントロピーが含まれます。一意性を回復する方法については、「Lambda での一意性の処理 SnapStart」を参照してください。

ネットワーク接続

Lambda がスナップショットから関数を再開するときは、関数が初期化フェーズ中に確立する接続の状態が保証されません。ネットワーク接続の状態を検証し、必要に応じて再確立してください。ほとんどの場合、AWS SDK が確立するネットワーク接続は、自動的に再開されます。その他の接続については、ベストプラクティスを確認してください。

一時的なデータ

関数には、初期化フェーズ中に一時的な認証情報やキャッシュされたタイムスタンプなどのエフェメラルデータをダウンロード、または初期化するものがあります。SnapStart を使用しない場合でも、エフェメラルデータは関数ハンドラーで更新してから使用してください。

SnapStart の料金

SnapStart の使用に追加のコストは発生しません。料金は、関数に対するリクエストの数、コードの実行に要する時間、および関数用に設定されたメモリの量に基づいて請求されます。所要時間は、コードの実行が開始されてから、それが戻る、または終了するまでの時間 (1 ミリ秒単位で切り上げ) で算出されます。

所要時間の料金は、関数ハンドラーで実行されるコード、ハンドラー外で宣言される初期化コード、ランタイム (JVM) のロードにかかる時間、およびランタイムフックで実行されるすべてのコードに適用されます。Lambda が期間を計算する方法の詳細については、「Lambda のモニタリング SnapStart」を参照してください。

SnapStart で設定された関数の場合、Lambda は実行環境を定期的にリサイクルし、初期化コードを再実行します。耐障害性のため、Lambda はスナップショットを複数のアベイラビリティーゾーンで作成します。Lambda が別のアベイラビリティーゾーンで初期化コードを再実行するたびに、料金が発生します。Lambda が料金を計算する方法の詳細については、「AWS Lambda 料金設定」を参照してください。

Lambda SnapStart とプロビジョニングされた同時実行の比較

Lambda SnapStart とプロビジョニングされた同時実行は、どちらも関数がスケールアップするときのコールドスタート時間と異常なレイテンシーを削減することができます。SnapStartは、起動パフォーマンスを最大 10 倍向上させるために役立ち、追加のコストはかかりません。プロビジョニングされた同時実行は、関数を、初期化され、2 桁ミリ秒台で応答できる状態に維持します。プロビジョニングされた同時実行を設定すると、AWS アカウントに料金が請求されます。プロビジョニングされた同時実行は、アプリケーションに厳格なコールドスタートレイテンシー要件がある場合に使

用してください。SnapStart とプロビジョニングされた同時実行の両方を同じ関数バージョンで使用することはできません。

> **Note**
>
> SnapStart は、大規模な関数呼び出しで使用すると最も効果的です。頻繁に呼び出されない関数では、パフォーマンスが同じように向上されない場合があります。

追加リソース

この章の他のトピックを読むだけでなく、「AWS Lambda SnapStart を使用した起動の高速化」ワークショップを試したり、AWS re: Invent 2022 の「Java 関数の高速コールドスタート」セッションを見ることもお勧めします。

Lambda のアクティブ化と管理 SnapStart

を使用するには SnapStart、新規または既存の Lambda 関数 SnapStart で を有効にします。次に、関数バージョンを発行し、呼び出します。

トピック

のアクティブ化 SnapStart （ コンソール ）

関数 SnapStart の をアクティブ化するには

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [Configuration] (設定) を選択してから、[General configuration] (一般設定) を選択します。
4. [General configuration] (一般設定) ペインで [Edit] (編集) を選択します。
5. 「基本設定の編集」ページの でSnapStart、「公開バージョン」を選択します。
6. [保存] を選択します。
7. 関数バージョンを発行します。Lambda がコードを初期化し、初期化された実行環境のスナップショットを作成してから、低レイテンシーアクセスのためにスナップショットをキャッシュします。
8. 関数バージョンを呼び出します。

SnapStart（ AWS CLI) のアクティブ化

既存の関数 SnapStart に対して をアクティブ化するには

1. --snap-start オプションを指定して update-function-configuration コマンドを実行して、関数の設定を更新します。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. publish-version コマンドを使用して、関数バージョンを発行します。

```
aws lambda publish-version \
  --function-name my-function
```

3. get-function-configuration コマンドを実行し、バージョン番号を指定して、 SnapStart が関数バージョンに対して有効になっていることを確認します。以下の例では、バージョン 1 が指定されています。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

レスポンスで OptimizationStatusが 、OnステートがActive、 であることが示されると SnapStart がアクティブになり、指定された関数バージョンでスナップショットが使用可能になります。

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. バージョンを指定した invoke コマンドを実行して、関数バージョンを呼び出します。以下の例は、バージョン 1 を呼び出します。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
```

```
  response.json
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

新しい関数の作成 SnapStart 時に を有効にするには

1. --snap-start オプションを指定した create-function コマンドを実行して、関数を作成します。--role には、実行ロールの Amazon リソースネーム (ARN) を指定します。

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java21" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. publish-version コマンドを使用して、バージョンを作成します。

```
aws lambda publish-version \
  --function-name my-function
```

3. get-function-configuration コマンドを実行し、バージョン番号を指定して、 SnapStart が関数バージョンに対して有効になっていることを確認します。以下の例では、バージョン 1 が指定されています。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

レスポンスで OptimizationStatusが On、ステートが Activeであることが示されると、SnapStart がアクティブになり、指定した関数バージョンでスナップショットが使用可能になります。

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
```

```
    "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. バージョンを指定した invoke コマンドを実行して、関数バージョンを呼び出します。以下の例は、バージョン 1 を呼び出します。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

SnapStart (API) のアクティブ化

をアクティブ化するには SnapStart

1. 次のいずれかを行います:
   - SnapStart パラメータを指定して CreateFunction API アクションを使用して、 SnapStart を有効にした新しい 関数を作成します。
   - SnapStart パラメータを指定して UpdateFunctionConfigurationアクションを使用して SnapStart 、既存の関数に対して を有効にします。
2. PublishVersion アクションを使用して関数バージョンを公開します。Lambda がコードを初期化し、初期化された実行環境のスナップショットを作成してから、低レイテンシーアクセスのためにスナップショットをキャッシュします。
3. GetFunctionConfiguration アクションを使用して、 SnapStart が関数バージョンに対して有効になっていることを確認します。バージョン番号を指定して、そのバージョンで SnapStart が有効になっていることを確認します。レスポンスで OptimizationStatusが On、ステートが であることが示されるとActive、 SnapStart がアクティブになり、指定された関数バージョンでスナップショットが使用可能になります。

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. Invoke アクションを使用して、関数バージョンを呼び出します。

Lambda SnapStart と関数の状態

を使用すると、次の関数状態が発生する可能性があります SnapStart。また、Lambda が実行環境を定期的にリサイクルし、 で設定された関数の初期化コードを再実行するときにも発生する可能性があります SnapStart。

- `Pending` – Lambda がコードを初期化し、初期化された実行環境のスナップショットを取得しています。関数バージョンに対して行われる呼び出しやその他の API アクションは、すべて失敗します。
- `Active` – スナップショットの作成が完了し、関数を呼び出すことができます。を使用するには SnapStart、未公開バージョン ($LATEST) ではなく、公開された関数バージョンを呼び出す必要があります。
- `Inactive` – 関数バージョンが 14 日間呼び出されていません。Lambda は、関数バージョンが `Inactive` になるとスナップショットを削除します。14 日後に関数バージョンを呼び出すと、Lambda は `SnapStartNotReadyException` レスポンスを返し、新しいスナップショットの初期化を開始します。関数バージョンが `Active` 状態になるまで待ってから、もう一度呼び出してください。
- `Failed` – 初期化コードの実行時、またはスナップショットの作成時に Lambda でエラーが発生しました。

スナップショットの更新

Lambda は、発行済みの関数バージョンそれぞれにスナップショットを作成します。スナップショットを更新するには、新しい関数バージョンを発行します。Lambda は、最新のランタイムとセキュリティパッチを使用して、スナップショットを自動的に更新します。

SnapStart で を使用する AWS SDK for Java

これらの関数から AWS SDK 呼び出しを行うために、Lambda は関数の実行ロールを引き受けることによって、一時的な一連の認証情報を生成します。これらの認証情報は、関数の呼び出し中に環境変数として利用できます。SDK の認証情報を、コード内で直接提供する必要はありません。デフォルトで、認証情報プロバイダーチェーンは認証情報を設定できる各場所を順番にチェックし、最初に利用できるものを選択します。これは通常、環境変数 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、および `AWS_SESSION_TOKEN`) です。

> **Note**
>
> SnapStart を有効にすると、Java ランタイムはアクセスキー環境変数の代わりにコンテナ認証情報 (`AWS_CONTAINER_CREDENTIALS_FULL_URI` および `AWS_CONTAINER_AUTHORIZATION_TOKEN`) を自動的に使用します。これは、関数が復元される前に認証情報の有効期限が切れることがないようにします。

SnapStart と AWS CloudFormation、AWS SAM、および の使用 AWS CDK

- AWS CloudFormation： テンプレートでSnapStartエンティティを宣言します。
- AWS Serverless Application Model (AWS SAM): テンプレートで SnapStartプロパティを宣言します。
- AWS Cloud Development Kit (AWS CDK)： SnapStartPropertyタイプを使用します。

スナップショットの削除

Lambda は、以下の場合にスナップショットを削除します。

- 関数または関数バージョンが削除された。
- 関数バージョンが 14 日間呼び出されなかった。呼び出されないまま 14 日間が過ぎると、関数バージョンの状態が Inactive に移行します。14 日後に関数バージョンを呼び出すと、Lambda は `SnapStartNotReadyException` レスポンスを返し、新しいスナップショットの初期化を開始します。関数バージョンが Active 状態になるまで待ってから、もう一度呼び出してください。

Lambda は、一般データ保護規則 (GDPR) に従って、削除されたスナップショットに関連付けられたすべてのリソースを削除します。

Lambda での一意性の処理 SnapStart

SnapStart 関数で呼び出しがスケールアップすると、Lambda は単一の初期化されたスナップショットを使用して複数の実行環境を再開します。スナップショットに包含される一意のコンテンツを初期化コードが生成する場合、そのコンテンツは、複数の実行環境で再利用されるときに一意にならない可能性があります。の使用時に一意性を維持するには SnapStart、初期化後に一意のコンテンツを生成する必要があります。これには、一意の ID、一意のシークレット、および疑似ランダム性を生成するために使用されるエントロピーが含まれます。

コードで一意性を維持できるように、以下のベストプラクティスをお勧めします。Lambda には、一意性を前提とするコードをチェックするのに役立つオープンソースのSnapStart スキャンツールも用意されています。初期化フェーズ中に一意のデータを生成する場合は、ランタイムフックを使用して一意性を復元することができます。ランタイムフックを使用すると、Lambda がスナップショットを取得する直前、または Lambda がスナップショットから関数を再開した直後に、特定のコードを実行できます。

一意性に依存する状態を初期化中に保存しない

関数の初期化フェーズ中は、ロギング用の一意の ID の生成など、一意であることが意図されたデータをキャッシュしないでください。その代わりに、関数ハンドラー内で一意のデータを生成する、またはランタイムフックを使用することをお勧めします。

Example – 関数ハンドラーでの一意の ID の生成

以下は、関数ハンドラーで UUID を生成する方法を示す例です。

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

暗号論的擬似乱数生成器 (CSPRNG) を使用する

アプリケーションがランダム性に依存している場合は、暗号論的擬似乱数生成器 (CSPRNG) を使用することをお勧めします。Java 用の Lambda マネージドランタイムには、 でランダム性を自動的に維持する 2 つの組み込み CSPRNGs (OpenSSL 1.0.2 および `java.security.SecureRandom`) が含まれています SnapStart。から乱数を常に取得`/dev/random`するか`/dev/urandom`、 で乱数を維持するソフトウェア SnapStart。

Example – java.security。SecureRandom

以下の例は `java.security.SecureRandom` を使用しています。これは、関数がスナップショットから復元された場合でも、一意の数列を生成します。

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
  }
```

SnapStart スキャンツール

Lambda は、一意性を前提としたコードをチェックするために役立つスキャンツールを提供しています。 SnapStart スキャンツールは、一連のルールに対して静的分析を実行するオープンソースSpotBugsプラグインです。スキャンツールは、一意性に関する前提を覆す可能性がある、潜在的なコード実装を特定するために役立ちます。インストール手順とスキャンツールが実行するチェックのリストについては、「」の「aws-lambda-snapstart-java-rules リポジトリ」を参照してください GitHub。

での一意性の処理の詳細については SnapStart、 AWS コンピューティングブログの「 による起動の高速化AWS Lambda SnapStart」を参照してください。

Lambda 関数スナップショットの前後のコード実装

ランタイムフックを使用して、Lambda がスナップショットを作成する前、または Lambda がスナップショットから関数を再開した後でコードを実装できます。ランタイムフックは、オープンソースの Coordinated Restore at Checkpoint (CRaC) プロジェクトの一部として提供されています。CRaC は、Open Java Development Kit (OpenJDK) 向けに開発中です。リファレンスアプリケーションで CRaC を使用する方法の例については、GitHub にある CRaC リポジトリを参照してください。CRaC は、3 つの主要要素を使用します。

- `Resource` – `beforeCheckpoint()` および `afterRestore()` の 2 つのメソッドを持つインターフェイス。これらのメソッドを使用して、スナップショット前、および復元後に実行するコードを実装します。
- `Context <R extends Resource>` – チェックポイントと復元に関する通知を受け取るには、`Resource` が `Context` に登録されている必要があります。
- `Core` – 静的メソッド `Core.getGlobalContext()` 経由でデフォルトのグローバル `Context` を提供するコーディネーションサービス。

`Context` および `Resource` の詳細については、CRaC ドキュメントの「Package org.crac」を参照してください。

org.crac package を使用してランタイムフックを実装するには、以下の手順を実行します。Lambda ランタイムには、チェックポイント作成前と復元後にランタイムフックを呼び出す、カスタマイズされた CRaC コンテキスト実装が含まれています。

ステップ 1: ビルド設定を更新する

ビルド設定に `org.crac` 依存関係を追加します。以下の例は、Gradle を使用しています。他のビルドシステムの例については、Apache Maven ドキュメントを参照してください。

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

ステップ 2: Lambda ハンドラーを更新する

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

詳細については、「Java の Lambda 関数ハンドラーの定義」を参照してください。

以下のハンドラー例は、チェックポイント作成前 (`beforeCheckpoint()`) と復元後 (`afterRestore()`) にコードを実行する方法を示しています。このハンドラーは、ランタイムが管理するグローバル `Context` への `Resource` の登録も行います。

> **Note**
>
> Lambda がスナップショットを作成するときは、初期化コードが最大 15 分間実行される場合があります。制限時間は 130 秒、または設定されている関数のタイムアウト (最大 900 秒) のいずれか長い方です。`beforeCheckpoint()` ランタイムフックは初期化コードの時間制限にカウントされます。Lambda がスナップショットを復元するときは、タイムアウト制限 (10 秒) 内にランタイム (JVM) がロードされ、`afterRestore()` ランタイムフックが完了される必要があります。その時間を超えると、SnapStartTimeoutException が発生します。

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
```

```
        throws Exception {
      System.out.println("After restore");
```

Context は、登録されたオブジェクトへの WeakReference のみを維持します。Resource に対してガベージコレクションが行われた場合、ランタイムフックは実行されません。ランタイムフックが実行されることを保証するには、コードが Resource への強参照を維持する必要があります。

以下は、避ける必要があるパターンの 2 つの例です。

Example – 強参照がないオブジェクト

```
Core.getGlobalContext().register( new MyResource() );
```

Example – 匿名クラスのオブジェクト

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }

} );
```

これらの代わりに、強参照を維持します。以下の例では、登録されたリソースに対してガベージコレクションが行われず、ランタイムフックが一貫的に実行されます。

Example – 強参照を持つオブジェクト

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda のモニタリング SnapStart

Amazon 、 CloudWatch、AWS X-Rayおよび を使用して Lambda SnapStart 関数をモニタリングできますLambda Telemetry API。

> **Note**
>
> AWS_LAMBDA_LOG_GROUP_NAME および AWS_LAMBDA_LOG_STREAM_NAME環境変数は Lambda SnapStart 関数では使用できません。

CloudWatch の SnapStart

SnapStart 関数のCloudWatch ログストリーム形式にはいくつかの違いがあります。

- 初期化ログ – 新しい実行環境が作成されると、REPORT は Init Duration フィールドを含みません。これは、Lambda が SnapStart 関数の呼び出し時ではなくバージョンを作成するときに関数を初期化するためです。 SnapStart 関数の場合、 Init Durationフィールドは INIT_REPORTレコードにあります。このレコードには、beforeCheckpoint ランタイムフックの所要時間を含めた、初期化フェーズ に関する所要時間の詳細情報が表示されています。
- 呼び出しログ – 新しい実行環境が作成されると、REPORT は Restore Duration および Billed Restore Duration フィールドを含みます。
  - Restore Duration: Lambda がスナップショットを復元し、ランタイム (JVM) をロードして、afterRestore ランタイムフックを実行するのにかかる時間。スナップショットを復元するプロセスには、MicroVM 外部でのアクティビティに費やす時間が含まれる場合があります。この時間は Restore Duration で報告されます。
  - Billed Restore Duration: Lambda がランタイム (JVM) をロードして afterRestore フックを実行するのにかかる時間。スナップショットの復元にかかった時間に対しては請求されません。

> **Note**
>
> 所要時間の料金は、関数ハンドラーで実行されるコード、ハンドラー外で宣言される初期化コード、ランタイム (JVM) のロードにかかる時間、およびランタイムフックで実行されるすべてのコードに適用されます。詳細については、「SnapStart の料金」を参照してください。

コールドスタートの所要時間は、Restore Duration と Duration の合計時間です。

次の例は、関数のレイテンシーパーセンタイルを返す Lambda Insights SnapStartクエリです。Lambda Insights クエリに関する詳細については、「クエリを使用して関数のトラブルシューティングを行うワークフローの例」を参照してください。

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

の X-Ray アクティブトレース SnapStart

X-Ray を使用して、Lambda SnapStart 関数へのリクエストをトレースできます。 SnapStart 関数の X-Ray サブセグメントにはいくつかの違いがあります。

- SnapStart 関数のInitializationサブセグメントはありません。
- Restore サブセグメントは、Lambda がスナップショットを復元し、ランタイム (JVM) をロードして、afterRestoreランタイムフックを実行するのにかかる時間を表しています。スナップショットを復元するプロセスには、MicroVM 外部でのアクティビティに費やす時間が含まれる場合があります。この時間は、Restore サブセグメントで報告されます。MicroVM 外部でスナップショットの復元に費やした時間については課金されません。

の Telemetry API イベント SnapStart

Lambda は、次の SnapStart イベントを に送信しますTelemetry API。

- platform.restoreStart – Restore フェーズの開始時刻を示します。
- platform.restoreRuntimeDone – Restore フェーズが正常に実行されたかどうかを示します。Lambda は、ランタイムが restore/next Runtime API リクエストを送信するときに、このメッセージを送信します。可能なステータスには、success (成功)、failure (失敗)、および timeout (タイムアウト) の 3 つがあります。

- `platform.restoreReport` – Restore フェーズの継続時間と、このフェーズ中に料金が請求されたミリ秒数を示します。

Amazon API Gateway と関数 URL メトリクス

API Gateway を使用してウェブ API を作成する場合、 IntegrationLatencyメトリクスを使用して end-to-end レイテンシーを測定できます (API Gateway がリクエストをバックエンドに中継してからバックエンドからレスポンスを受信するまでの時間) 。

Lambda 関数 URL を使用している場合は、 UrlRequestLatencyメトリクスを使用して end-to-end レイテンシーを測定できます (関数 URL がリクエストを受信してから関数 URL がレスポンスを返すまでの時間) 。

Lambda のセキュリティモデル SnapStart

Lambda は保管時の暗号化 SnapStart をサポートしています。Lambda は、AWS KMS key を使用してスナップショットを暗号化します。デフォルトで、Lambda は AWS マネージドキーを使用します。このデフォルト動作がワークフローに適している場合、追加の設定を行う必要はありません。それ以外の場合は、create-function または update-function-configuration コマンドの `--kms-key-arn`オプションを使用して、 AWS KMSカスタマーマネージドキーを指定できます。これは、KMS キーのローテーションを制御する、または KMS キーの管理に関する組織の要件を満たすために実行できます。カスタマーマネージドキーには、標準の AWS KMS 料金が発生します。詳細については、「AWS Key Management Service 料金表」をご覧ください。

SnapStart 関数または関数バージョンを削除すると、その関数または関数バージョンに対するすべての`Invoke`リクエストが失敗します。Lambda は、14 日間呼び出されなかったスナップショットを自動的に削除します。Lambda は、一般データ保護規則 (GDPR) に従って、削除されたスナップショットに関連付けられたすべてのリソースを削除します。

Lambda SnapStart パフォーマンスの最大化

トピック

パフォーマンスチューニング

> **Note**
>
> SnapStart は、大規模な関数呼び出しで使用すると最も効果的です。頻繁に呼び出されない関数では、パフォーマンスが同じように向上されない場合があります。

SnapStart のメリットを最大限に活用するためにも、起動時のレイテンシーの原因となるクラスは、関数ハンドラーではなく、初期化コードに事前ロードしておくことをお勧めします。そうすることで、高負荷のクラスローディングに関連するレイテンシーが呼び出しパスから排除され、SnapStart での起動パフォーマンスが最適化されます。

初期化中にクラスを事前ロードできない場合は、ダミー呼び出しを使用してクラスを事前ロードすることをお勧めします。これを実行するには、AWS Labs GitHub リポジトリの pet-store 関数からの以下の例にあるように、関数ハンドラーコードを更新します。

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
              FilterRegistration.Dynamic registration =
 servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
              registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
 false, "/*");
          });

          // Send a fake Amazon API Gateway request to the handler to load classes
 ahead of time
          ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
```

```
            identity.setApiKey("foo");
            identity.setAccountId("foo");
            identity.setAccessKey("foo");

            AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
            reqCtx.setPath("/pets");
            reqCtx.setStage("default");
            reqCtx.setAuthorizer(null);
            reqCtx.setIdentity(identity);

            AwsProxyRequest req = new AwsProxyRequest();
            req.setHttpMethod("GET");
            req.setPath("/pets");
            req.setBody("");
            req.setRequestContext(reqCtx);

            Context ctx = new TestContext();
            handler.proxy(req, ctx);


        } catch (ContainerInitializationException e) {
            // if we fail here. We re-throw the exception to force another cold start
            e.printStackTrace();
            throw new RuntimeException("Could not initialize Spring framework", e);
        }
    }
```

ネットワークのベストプラクティス

Lambda がスナップショットから関数を再開するときは、関数が初期化フェーズ中に確立する接続の状態が保証されません。ほとんどの場合、AWS SDK が確立するネットワーク接続は、自動的に再開されます。これ以外の接続については、以下のベストプラクティスが推奨されます。

ネットワーク接続を再確立する

関数がスナップショットから再開されるときは、常にネットワーク接続を再確立してください。ネットワーク接続を再確立は、関数ハンドラーで実行することをお勧めします。代替手段として、`afterRestore` ランタイムフックを使用することもできます。

ホスト名を一意の実行環境識別子として使用しない

実行環境をアプリケーション内の一意のノードやコンテナとして特定するために `hostname` を使用することはお勧めしません。SnapStart では、複数の実行環境の初期状態として単一のス

ナップショットが使用され、すべての実行環境が InetAddress.getLocalHost() に対して同じ hostname を返します。一意の実行環境アイデンティティ、または hostname 値を必要とするアプリケーションでは、関数ハンドラーで一意の ID を生成することをお勧めします。または、afterRestore ランタイムフックを使用して一意の ID を生成してから、その一意の ID を実行環境の識別子として使用してください。

接続を固定送信元ポートにバインドしない

ネットワーク接続を固定ソースポートにバインドしないことをお勧めします。接続は、関数がスナップショットから再開されるときに再確立され、固定送信元ポートにバインドされたネットワーク接続は失敗する可能性があります。

Java DNS キャッシュを使用しない

Lambda 関数は、既に DNS レスポンスをキャッシュしています。SnapStart で別の DNS キャッシュを使用すると、関数がスナップショットから再開されるときに接続タイムアウトが発生する可能性があります。

java.util.logging.Logger クラスは JVM DNS キャッシュを間接的に有効にできます。デフォルト設定を上書きするには、 を初期化する前に networkaddress.cache.ttl を 0 に設定しますlogger。例：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

UnknownHostException 障害を防ぐために、 networkaddress.cache.negative.ttlを 0 に設定することをお勧めします。このプロパティは、AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0環境変数を使用して Lambda 関数に設定できます。

JVM DNS キャッシュを無効にしても、Lambda のマネージド DNS キャッシュは無効になりません。

Java Lambda 関数のカスタマイズ設定

このページでは、AWS Lambda の Java 関数固有の設定について説明します。これらの設定を使用して、Java ランタイムの起動時の動作をカスタマイズできます。これにより、コードを変更することなく、関数全体のレイテンシを低下させ、関数全体のパフォーマンスを向上させることができます。

セクション

JAVA_TOOL_OPTIONS 環境変数

Java を使用する場合、Lambda は追加のコマンドライン変数を設定するために、JAVA_TOOL_OPTIONS 環境変数をサポートします。この環境変数は、階層化コンパイル設定のカスタマイズなど、さまざまな方法で使用できます。次の例では、このユースケースで JAVA_TOOL_OPTIONS 環境変数を使用する方法を示します。

例: 階層型コンパイル設定のカスタマイズ

階層型コンパイルとは、Java 仮想マシン (JVM) の機能です。JVM のジャストインタイム (JIT) コンパイラを十分に活用するため、特定の階層型コンパイル設定を使用することができます。通常、C1 コンパイラは起動時間を短縮するように最適化されています。C2 コンパイラは、全体的なパフォーマンスを最大限に高めるように最適化されていますが、メモリの使用量が多く、処理に時間がかかります。

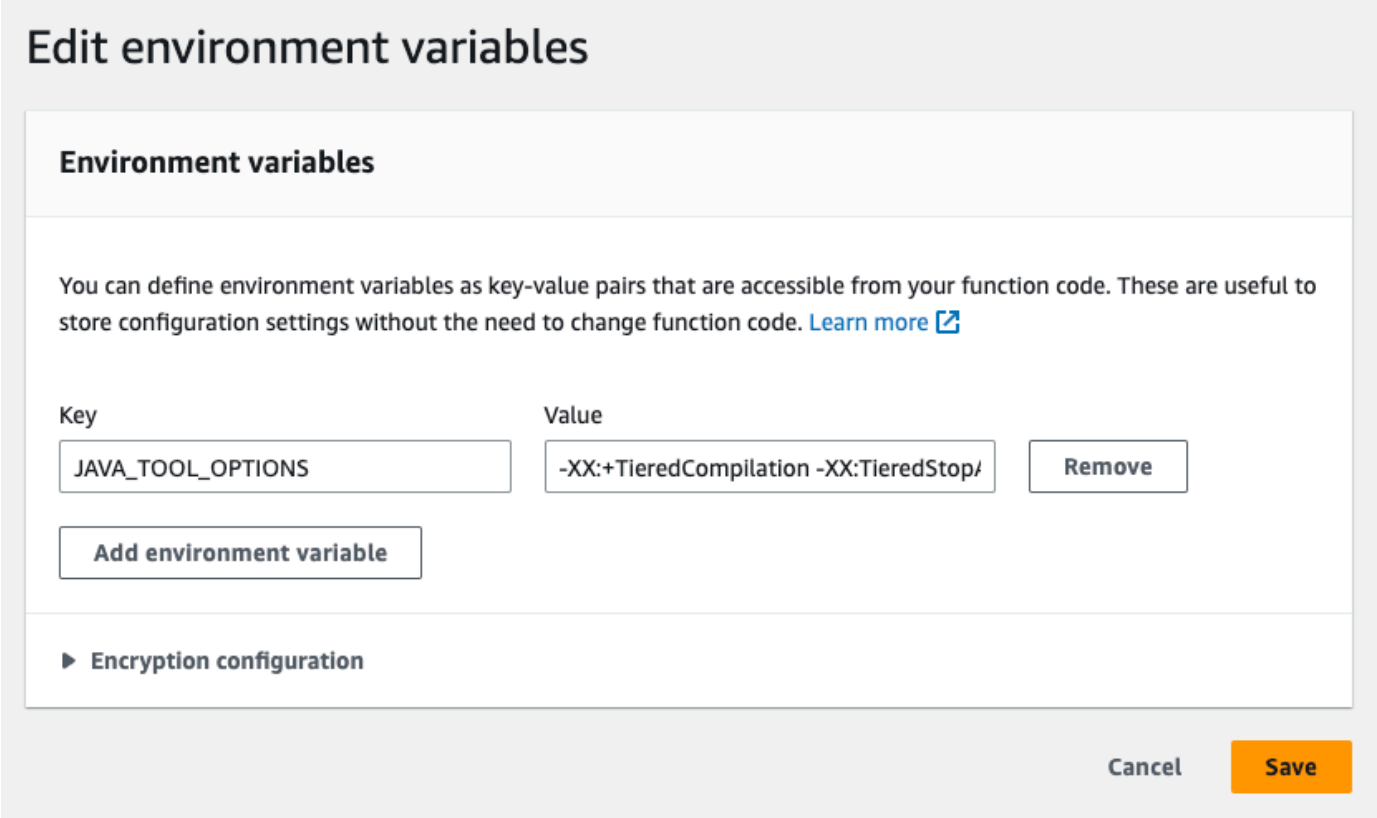
階層型コンパイルには 5 つの異なるレベルがあります。レベル 0 では、JVM は Java バイトコードを解釈します。レベル 4 では、JVM は C2 コンパイラを使用して、アプリケーションの起動時に収集されたプロファイリングデータを分析します。時間の経過とともに、コードの使用状況を監視して最善の最適化方法を特定します。

階層型コンパイルレベルをカスタマイズすると、Java 関数のコールドスタートのレイテンシ低下に役立ちます。例えば、JVM が C1 コンパイラを使用するようにするには、階層型コンパイルのレベルを 1 に設定します。このコンパイラは、最適化されたネイティブコードを迅速に生成しますが、プロファイリングデータは生成せず、C2 コンパイラも使用しません。

Java 17 ランタイムでは、階層化コンパイルの JVM フラグはデフォルトでレベル 1 で停止するように設定されています。Java 11 以前のランタイムでは、次の手順を実行して階層化コンパイルレベルを 1 に設定できます。

階層型コンパイル設定をカスタマイズするには (コンソール)

1. Lambda コンソールの [関数] ページを開きます。
2. 階層型コンパイルをカスタマイズする Java 関数を選択します。
3. [設定] タブを選択し、左側のメニューで [環境変数] を選択します。
4. [編集] を選択します。
5. [環境変数の追加] を選択します。
6. [キー] に、`JAVA_TOOL_OPTIONS` と入力します。[値] に、`-XX:+TieredCompilation -XX:TieredStopAtLevel=1` と入力します。



7. [保存] をクリックします。

> **Note**
>
> Lambda SnapStart を使用して、コールドスタートの問題を軽減することもできます。SnapStartは、実行環境のキャッシュされたスナップショットを使用して、起動時のパフォーマンスを大幅に向上させます。SnapStartの機能、制限事項、サポートされているリー

ジョンの詳細については、Lambda SnapStart による起動パフォーマンスの向上 を参照してください。

例: JAVA_TOOL_OPTIONS を使用した GC の動作のカスタマイズ

Java 11 ランタイムは、ガベージコレクションにシリアルガベージコレクター (GC) を使用します。デフォルトでは、Java 17 ランタイムもシリアル GC を使用します。ただし、Java 17 では、JAVA_TOOL_OPTIONS 環境変数を使用してデフォルトの GC を変更することもできます。パラレル GC と Shenandoah GC のどちらかを選択できます。

例えば、ワークロードでより多くのメモリと複数の CPU を使用する場合は、パフォーマンス向上のためにパラレル GC の使用を検討してください。そのためには、JAVA_TOOL_OPTIONS 環境変数の値に以下の値を追加します。

```
-XX:+UseParallelGC
```

Java の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を示すメソッドおよびプロパティを提供します。

context メソッド

- `getRemainingTimeInMillis()` - 実行がタイムアウトするまでの残りのミリ秒数を返します。
- `getFunctionName()` - Lambda 関数の名前を返します。
- `getFunctionVersion()` - 関数のバージョンを返します。
- `getInvokedFunctionArn()` - この関数を呼び出すために使用される Amazon リソースネーム (ARN) を返します。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `getMemoryLimitInMB()` - 関数に割り当てられたメモリの量を返します。
- `getAwsRequestId()` - 呼び出しリクエストの ID を返します。
- `getLogGroupName()` - 関数のロググループを返します。
- `getLogStreamName()` - 関数インスタンスのログストリームを返します。
- `getIdentity()` - (モバイルアプリ) リクエストを承認した Amazon Cognito ID に関する情報を返します。
- `getClientContext()` - (モバイルアプリ) クライアントアプリケーションから Lambda に提供されるクライアントコンテキストを返します。
- `getLogger()` - 関数のロガーオブジェクトを返します。

以下の例では、コンテキストオブジェクトを使用して Lambda ロガーにアクセスする関数を示しています。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```
import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

関数は、null を返す前に受信イベントのクラスタイプをログに記録します。

Example ログ出力

```
EVENT TYPE: class java.util.LinkedHashMap
```

コンテキストオブジェクトのインターフェイスは aws-lambda-java-core ライブラリに含まれています。このインターフェイスを実装して、テスト用のコンテキストクラスを作成できます。以下の例では、ほとんどのプロパティのダミー値を返すコンテキストクラス、およびテスト用のロガーを示しています。

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
```

```
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

ログ記録の詳細については、「」を参照してくださいJava の AWS Lambda 関数ログ作成

サンプルアプリケーションのコンテキスト

このガイドの GitHub リポジトリには、コンテキストオブジェクトの使用方法を示すサンプルアプリケーションが含まれています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS Serverless Application Model (AWS SAM) テンプレート、サポートリソースが含まれています。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。

- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API ゲートウェイを使用して関数を呼び出します。

Java の AWS Lambda 関数ログ作成

AWS Lambda は、Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログエントリを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda のランタイム環境は、各呼び出しの詳細や、関数のコードからのその他の出力をログストリームに送信します。CloudWatch Logs の詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

関数コードからログを出力するには、「java.lang.System」のメソッドを使用するか、stdout または stderr に書き込む任意のログ記録モジュールを使用できます。

セクション

ログを返す関数の作成

関数コードからログを出力するには、java.lang.System のメソッドを使用するか、`stdout` や `stderr` に書き込む任意のログ記録モジュールを使用できます。aws-lambda-java-core ライブラリには、コンテキストオブジェクトからアクセスできる `LambdaLogger` という名前のロガークラスが用意されています。ロガークラスは複数行のログをサポートしています。

以下の例では、コンテキストオブジェクトによって提供される `LambdaLogger` ロガーを使用しています。

Example Handler.java

```
// Handler value: example.Handler
```

```
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example ログの形式

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
       ...
    }
  ]
```

```
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 200 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

Java ランタイムは、呼び出しごとに START、END、および REPORT の各行を記録します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- サンプリング済み - トレースされたリクエストの場合、サンプリング結果。

Java での Lambda の高度なログ記録コントロールの使用

関数のログのキャプチャ、処理、利用方法をより細かく制御できるように、サポートされている Java ランタイムに以下のログ記録オプションを設定できます。

- ログの形式 - 関数のログをプレーンテキスト形式と構造化された JSON 形式から選択します
- ログレベル - JSON 形式のログの場合、Lambda が CloudWatch に送信するログの詳細レベル (ERROR、DEBUG、INFO など) を選択します。
- ロググループ - 関数がログを送信する CloudWatch ロググループを選択します

これらのログ記録オプションの詳細と、それらのオプションを使用するように関数を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。

Java Lambda 関数でログ形式とログレベルのオプションを使用するには、以下のセクションのガイダンスを参照してください。

Java での構造化された JSON ログ形式の使用

関数のログ形式に JSON を選択した場合、Lambda は `LambdaLogger` クラスを使用してログ出力を構造化された JSON として送信します。各 JSON ログオブジェクトには、以下のキーを含む少なくとも 4 つのキーと値のペアが含まれます。

- `"timestamp"` - ログメッセージが生成された時刻
- `"level"` - メッセージに割り当てられたログレベル
- `"message"` - ログメッセージの内容
- `"AWSrequestId"` - 関数呼び出しの一意のリクエスト ID

使用するログ記録方法によっては、JSON 形式でキャプチャされた関数からのログ出力に、追加のキーと値のペアが含まれる場合もあります。

`LambdaLogger` ロガーを使用して作成するログにレベルを割り当てるには、次の例に示すように、ログコマンドに `LogLevel` 引数を指定する必要があります。

Example Java ログ記録コード

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

このサンプルコードのログ出力は、次のように CloudWatch Logs にキャプチャされます。

Example JSON ログレコード

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

ログ出力にレベルを割り当てない場合、Lambda は自動的に INFO レベルを割り当てます。

コードで既に別のログ記録ライブラリを使用して JSON 構造化ログを作成している場合、変更を加える必要はありません。Lambda は、既に JSON でエンコードされているログを二重エンコードし

ません。JSON ログ形式を使用するように関数を設定しても、ログ記録出力は定義した JSON 構造で CloudWatch に表示されます。

Java でのログレベルフィルタリングの使用

AWS Lambda でログレベルに従ってアプリケーションログをフィルタリングするには、関数で JSON 形式のログを使用する必要があります。このためには以下の 2 つの方法があります。

- 標準的な `LambdaLogger` を使用してログ出力を作成し、JSON ログ形式を使用するように関数を設定する。次に、Lambda は、「the section called “Java での構造化された JSON ログ形式の使用”」で説明されている JSON オブジェクトの「level」キーと値のペアを使用してログ出力をフィルタリングします。関数のログ形式を設定する方法については、「the section called “Lambda 関数の高度なログ記録コントロールの設定”」を参照してください。
- 別のログ記録ライブラリまたはメソッドを使用して、ログ出力のレベルを定義する「level」キーと値のペアを含む JSON 構造化ログをコード内に作成する。`stdout` または `stderr` に JSON ログの書き込みを行う任意のログ記録ライブラリを使用できます。例えば、Powertools for AWS Lambda または Log4j2 パッケージを使用して、コードから JSON 構造化ログ出力を生成できます。詳細については、「the section called “AWS Lambda (Java) に Powertools の使用、構造化されたログ記録に AWS SAM の使用”」および「the section called “Log4j2 および SLF4J による高度なログ記録”」を参照してください。

ログレベルのフィルタリングを使用するように関数を設定する場合、Lambda が CloudWatch Logs に送信するログのレベルを以下のオプションから選択する必要があります。

ログレベル	標準的な使用状況
TRACE (最も詳細)	コードの実行パスを追跡するために使用される最も詳細な情報
DEBUG	システムデバッグの詳細情報
INFO	関数の通常の動作を記録するメッセージ
WARN	対処しないと予期しない動作を引き起こす可能性がある潜在的なエラーに関するメッセージ
ERROR	コードが期待どおりに動作しなくなる問題に関するメッセージ

ログレベル	標準的な使用状況
FATAL (詳細度が最も低い)	アプリケーションの機能停止を引き起こす重大なエラーに関するメッセージ

Lambda で関数のログをフィルタリングするには、JSON ログ出力に `"timestamp"` のキーと値のペアも含める必要があります。時間は、有効な RFC 3339 タイムスタンプ形式で指定する必要があります。有効なタイムスタンプを指定しない場合、Lambda はログに INFO レベルを割り当ててタイムスタンプを追加します。

Lambda は、選択したレベル以下のログを CloudWatch に送信します。例えば、ログレベルを WARN に設定すると、Lambda は WARN、ERROR、FATAL の各レベルに対応するログを送信します。

Log4j2 および SLF4J による高度なログ記録

> **Note**
>
> AWS Lambda では、そのマネージドランタイムとベースコンテナイメージに Log4j2 が含まれていません。このため、これらは CVE-2021-44228、CVE-2021-45046、および CVE-2021-45105 で説明されている問題の影響を受けません。
> お客様の関数に影響を受ける Log4j2 バージョンが含まれているという場合のため、CVE-2021-44228、CVE-2021-45046、および CVE-2021-45105 の問題の緩和に役立つ変更を Lambda Java マネージドランタイムとベースコンテナイメージに適用しました。この変更の結果、Log4J2 を使用しているお客様には、「`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`」のような追加のログエントリが表示される場合があります。Log4J2 出力の jndi マッパーを参照するログ文字列は、いずれも「`Patched JndiLookup::lookup()`」に置き換えられます。
> この変更にかかわらず、Log4j2 が含まれる関数を持つすべてのお客様に、最新バージョンへの更新を強くお勧めします。特に、関数で aws-lambda-java-log4j2 ライブラリを使用しているお客様は、バージョン 1.5.0 (またはそれ以降) に更新して、関数を再デプロイするようにしてください。このバージョンは、基盤となる Log4j2 ユーティリティの依存関係をバージョン 2.17.0 (またはそれ以降) に更新します。更新された aws-lambda-java-log4j2 バイナリは Maven リポジトリで、そのソースコードは GitHub で入手できます。
> 最後に、aws-lambda-java-log4j (v1.0.0 または 1.0.1) に関連するライブラリは、いかなる状況でも使用すべきではないことに注意してください。これらのライブラリは 2015 年にサ

ポートが終了した log4j のバージョン 1.x に関連しています。これらのライブラリはサポートも保守もパッチも適用されておらず、既知のセキュリティの脆弱性があります。

ログ出力をカスタマイズし、ユニットテスト中のログ記録をサポートし、AWS SDK 呼び出しをログに記録するには、SLF4J で Apache Log4j2 を使用します。Log4j は、ログレベルを設定し、アペンダーライブラリを使用できるようにする Java プログラムのログ記録ライブラリです。SLF4J は、関数コードを変更せずに使用するライブラリを切り替えることができるファサードライブラリです。

関数のログにリクエスト ID を追加するには、aws-lambda-java-log4j2 ライブラリのアペンダーを使用します。

Example src/main/resources/log4j2.xml - アペンダー設定

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
       <AppenderRef ref="Lambda"/>
     </Root>
     <Logger name="software.amazon.awssdk" level="WARN" />
     <Logger name="software.amazon.awssdk.request" level="DEBUG" />
   </Loggers>
 </Configuration>
```

Log4j2 ログをプレーンテキスト出力または JSON 出力用に設定する方法は、<LambdaTextFormat> および <LambdaJSONFormat> タグの下にレイアウトを指定することで決定できます。

この例では、テキストモードで、各行の先頭に日付、時刻、リクエスト ID、ログレベル、クラス名が追加されます。JSON モードでは、<JsonTemplateLayout> が aws-lambda-java-log4j2 ライブラリに付属する設定で使用されます。

SLF4J は、Java コードでのログ記録のためのファサードライブラリです。関数コードで、SLF4J ロガーファクトリを使用して、info() や warn() などのログレベルのメソッドにより、ロガーを取得します。ビルド設定で、ログ記録ライブラリと SLF4J アダプターをクラスパスに含めます。ビルド設定のライブラリを変更することで、関数コードを変更せずにロガータイプを切り替えることができます。SDK for Java からログをキャプチャするには SLF4J が必要です。

以下のサンプルコードでは、ハンドラークラスは SLF4J を使用してロガーを取得しています。

Example src/main/java/example/HandlerS3.java - SLF4J を使用したログ記録

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
```

```
}
```

このコードにより以下のようなログ出力が生成されます。

Example ログの形式

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

ビルド設定では、Lambda アペンダーおよび SLF4J アダプターのランタイム依存関係と、Log4j2 の実装依存関係を使用します。

Example build.gradle - 依存関係のログ記録

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

テスト用にローカルでコードを実行する場合、Lambda ロガーへのアクセスにコンテキストオブジェクトは使用できず、Lambda アペンダーが使用するリクエスト ID はありません。テスト設定の例については、次のセクションのサンプルアプリケーションを参照してください。

その他のツールとライブラリ

Powertools for AWS Lambda (Java) は、サーバーレスのベストプラクティスを実装し、デベロッパーの作業速度を向上させるためのデベロッパーツールキットです。ログ記録ユーティリティに

は、Lambda に最適化されたロガーを提供し、すべての関数を通して関数コンテキストに関する追加情報が含まれ、JSON 形式で構成した出力を行います。このユーティリティを使用して次のことができます。

- Lambda の コンテキスト、コールドスタート、構造から主要キーをキャプチャし、JSON 形式でログ出力する
- 指示された場合 Lambda 呼び出しイベントをログ記録する (デフォルトでは無効)
- ログサンプリングにより、特定の割合の呼び出しにのみすべてのログを出力する (デフォルトでは無効)
- 任意のタイミングで、構造化されたログにキーを追加する
- カスタムログフォーマッター (Bring Your Own Formatter) を使用して、組織の ログ記録 RFC と互換性のある構造でログを出力する

AWS Lambda (Java) に Powertools の使用、構造化されたログ記録に AWS SAM の使用

以下の手順に従い、AWS SAM を使用する統合された Powertools for AWS Lambda (Java) モジュールを備えた Hello World Java アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Java 11
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World Java テンプレートを使用してアプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name sam-app
```

ログ出力は次のようになります。

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
```

```
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
```

```
  "executionEnvironment": "AWS_Lambda_java11"
}
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
2023/02/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 4119 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

ログ保持の管理

関数を削除しても、ロググループは自動的には削除されません。ログを無期限に保存しないようにするには、ロググループを削除するか、CloudWatch がログを自動的に削除するまでの保持期間を設定します。ログ保持を設定するには、AWS SAM テンプレートに以下を追加します。

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Lambda コンソールを使用する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/###)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。　X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および `--log-type` コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、`LogResult`フィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、`LogResult`という名前の関数の`my-function`フィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
```

```
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

サンプルログ記録コード

このガイドの GitHub リポジトリには、さまざまなログ記録設定の使用方法を示すサンプルアプリケーションが含まれています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS SAM テンプレート、サポートリソースが含まれています。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API ゲートウェイを使用して関数を呼び出します。

`java-basic` サンプルアプリケーションは、ログ記録テストをサポートする最小限のログ記録設定を示しています。ハンドラーコードは、コンテキストオブジェクトによって提供される `LambdaLogger` ロガーを使用します。テストでは、アプリケーションは、Log4j2 ロガーとの `LambdaLogger` インターフェイスを実装するカスタム `TestLogger` クラスを使用します。また、AWS SDK との互換性のためのファサードとして SLF4J を使用します。ログ記録ライブラリは、デプロイパッケージを小さく保つために、ビルド出力から除外しています。

AWS Lambda での Java コードの作成

Lambda アプリケーションのトレース、デバッグ、および最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレースされたデータを X-Ray に送信するには、以下の 2 つの SDK ライブラリのいずれかを使用します。

- AWS Distro for OpenTelemetry (ADOT) – 安全で、本番環境に対応し、AWS でサポートされている OpenTelemetry (OTel) SDK のディストリビューションです。
- AWS X-Ray SDK for Java - トレースデータを生成して X-Ray に送信するための SDK です。
- Powertools for AWS Lambda (Java) - サーバーレスのベストプラクティスを実装し、デベロッパーの作業速度を向上させるためのデベロッパーツールキットです。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。これらのソリューションのいずれかを使用して、X-Ray でエンドツーエンドのトレーシングを実装することができます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

- AWS Lambda (Java) に Powertools の使用、トレースに AWS SAM の使用
- AWS Lambda (Java) に Powertools の使用、トレースに AWS CDK の使用
- Java 関数の計測への ADOT の使用
- Java 関数の計測のための X-Ray SDK の使用
- Lambda コンソールを使用してトレースを有効化する

- Lambda API でのトレースのアクティブ化
- AWS CloudFormation によるトレースのアクティブ化
- X-Ray トレースの解釈
- ランタイムの依存関係をレイヤー (X-Ray SDK) に保存する
- サンプルアプリケーションでの X-Ray トレース (X-Ray SDK)

AWS Lambda (Java) に Powertools の使用、トレースに AWS SAM の使用

以下の手順に従い、AWS SAM を使用する統合された Powertools for AWS Lambda (Java) モジュールを備えた Hello World Java アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Java 11
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World Java テンプレートを使用してアプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
New XRay Service Graph
  Start time: 2023-02-03 14:31:48+01:00
  End time: 2023-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
```

```
      - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 3] at (2023-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

AWS Lambda (Java) に Powertools の使用、トレースに AWS CDK の使用

以下の手順に従い、AWS CDK を使用する統合された Powertools for AWS Lambda (Java) モジュールを備えた Hello World Java アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は「hello world」メッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Java 11
- AWS CLI バージョン 2

- AWS CDK バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS CDK サンプルアプリケーションをデプロイする

1. 新しいアプリケーション用のプロジェクトディレクトリを作成します。

```
mkdir hello-world
cd hello-world
```

2. アプリケーションを初期化します。

```
cdk init app --language java
```

3. 次のコマンドを備えた Maven プロジェクトを作成します。

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. hello-world\app\Function ディレクトリで pom.xml を開き、既存のコードを、Powertools の依存関係および Maven プラグインを含む次のコードで置き換えます。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
```

```
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
```

```
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
```

```
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

5. hello-world\app\src\main\resource ディレクトリを作成し、ログ設定に log4j.xml を作成します。

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. log4j.xml を開いて次のコード追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. hello-world\app\Function\src\main\java\helloworld ディレクトリから App.java を開き、既存コードを次のコードで置き換えます。これは Lambda 関数のコードです。

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");
```

```
        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. hello-world\src\main\java\com\myorg ディレクトリから HelloWorldStack.java を開き、既存コードを次のコードで置き換えます。このコードは、「Lambda Constructor」および「Apigatewayv2 Constructor」を使用して REST API および Lambda 関数を作成します。

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
```

```
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);
```

```
        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());

        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. hello-world ディレクトリから pom.xml を開き、既存コードを次のコードで置き換えます。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
```

```
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
```

```
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. hello-world ディレクトリ内に移動していることを確認し、アプリケーションをデプロイしてください。

```
cdk deploy
```

11. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

13. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
cdk destroy
```

Java 関数の計測への ADOT の使用

ADOT は、Otel SDK を使用してテレメトリデータを収集するために必要なすべてをパッケージ化した、フルマネージド型の Lambda レイヤーを提供します。このレイヤーを使用すると、関数コードを変更する必要はなしで、Lambda 関数を計測できます。また、このレイヤーは、OTel でのカスタムな初期化を実行するように構成することもできます。詳細については、ADOT のドキュメントの「Custom configuration for the ADOT Collector on Lambda」(Lambda 上での ADOT Collector のカスタム設定) 参照してください。

Java ランタイムの場合、使用するレイヤーを以下の 2 つの中から選択できます。

- AWS ADOT Java 向けのマネージド Lambda レイヤー (自動計測エージェント) – このレイヤーは、起動時に関数コードを自動的に変換し、トレーシングデータを収集できるようにします。このレイヤーを ADOT Java エージェントとともに使用する方法の詳細については、「ADOT ドキュメント」の「AWS Distro for OpenTelemetry Lambda Support for Java (Auto-instrumentation Agent)」を参照してください。
- ADOT Java 用の AWS マネージド型 Lambda レイヤー – このレイヤーによっても、Lambda 関数の組み込み型の計測機能が提供されます。ただし、OTel SDK を初期化するために、手動によるコード変更がいくつか必要となります。このレイヤーを使用する方法の詳細については、「ADOT ドキュメント」の「AWS Distro for OpenTelemetry Lambda Support for Java」を参照してください。

Java 関数の計測のための X-Ray SDK の使用

関数が、アプリケーション内で他のリソースやサービスに対して行う呼び出しのデータを記録するには、X-Ray SDK for Java をビルド設定に追加します。以下に、AWS SDK for Java 2.x クライアントの自動計測をアクティブ化するライブラリを含む Gradle ビルド設定の例を示します。

Example build.gradle - 依存関係のトレース

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
```

```
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

正しい依存関係を追加し、必要なコード変更を行った後、Lambda コンソールまたはAPIを介して関数の構成でトレースをアクティブにします。

Lambda コンソールを使用してトレースを有効化する

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。
4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [保存] をクリックします。

Lambda API でのトレースのアクティブ化

AWS CLI または AWS SDK で Lambda 関数のトレースを設定するには、次の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation によるトレースのアクティブ化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray トレースの解釈

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

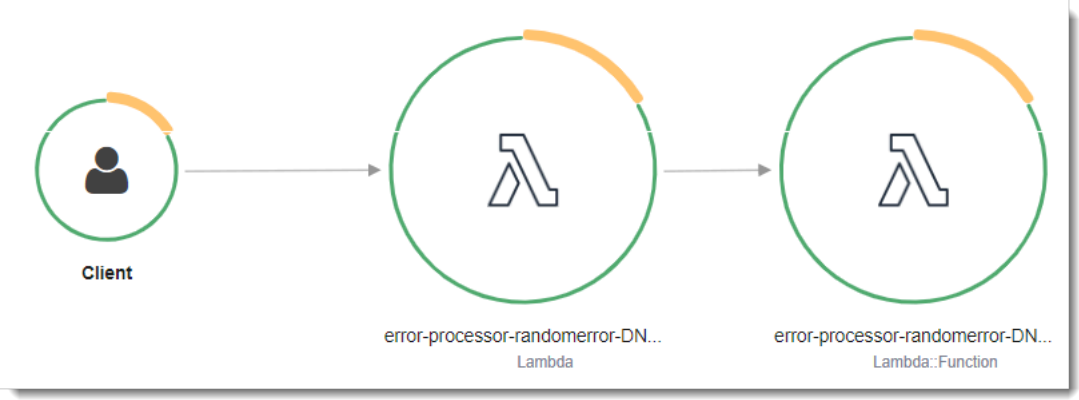
アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロッグ

ループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

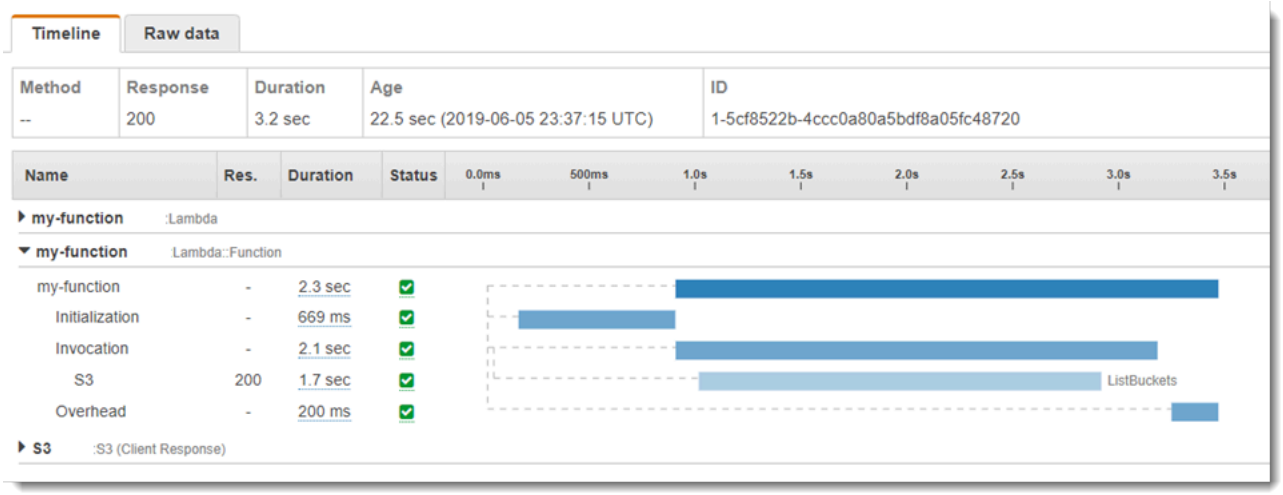
X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは AWS::Lambda の起点があり、もう 1 つは AWS::Lambda::Function の起点があります。AWS::Lambda セグメントにエラーが表示される場合は、Lambda サービスに問題があります。AWS::Lambda::Function セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

> **Note**
>
> Lambda SnapStart 関数には Restore サブセグメントも含まれます。Restore サブセグメントは、Lambda がスナップショットを復元し、ランタイム (JVM) をロードして、afterRestoreランタイムフックを実行するのにかかる時間を表しています。スナップショットを復元するプロセスには、MicroVM 外部でのアクティビティに費やす時間が含まれる場合があります。この時間は、Restore サブセグメントで報告されます。MicroVM 外部でスナップショットの復元に費やした時間については課金されません。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、「AWS X-Ray デベロッパーガイド」の「AWS X-Ray SDK for Java」を参照してください。

> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

ランタイムの依存関係をレイヤー (X-Ray SDK) に保存する

X-Ray SDK を使用して AWS SDK クライアントを関数コードに埋め込むと、デプロイパッケージが巨大になる可能性があります。機能コードを更新するたびに実行時の依存関係がアップロードされないようにするには、X-Ray SDK を [Lambda layer] (Lambda レイヤー) にパッケージ化します。

以下の例では、AWS SDK for Java および X-Ray SDK for Java を保存する `AWS::Serverless::LayerVersion` リソースを示しています。

Example template.yml - 依存関係レイヤー

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java21
```

この設定では、ランタイム依存関係を変更した場合にのみ、ライブラリレイヤーの更新が必要です。関数のデプロイパッケージにはユーザーのコードのみが含まれるため、アップロード時間を短縮できます。

依存関係のレイヤーを作成するには、デプロイ前にレイヤーアーカイブを生成するようにビルド設定を変更する必要があります。実際の例については、GitHub のサンプルアプリケーション、java-basic を参照してください。

サンプルアプリケーションでの X-Ray トレース (X-Ray SDK)

このガイドで参照する GitHub リポジトリには、X-Ray トレースの使用方法を示すサンプルアプリケーションが含まれています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS SAM テンプレート、サポートリソースが含まれています。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API ゲートウェイを使用して関数を呼び出します。

すべてのサンプルアプリケーションでは、Lambda 関数に対するアクティブトレースが有効になっています。例えば、`s3-java` アプリケーションは、AWS SDK for Java 2.x クライアントの自動インストルメンテーション、テストのセグメント管理、カスタムのサブセグメント、Lambda レイヤーによるラインタイムの依存関係の保存を示しています。

AWS Lambda の Java サンプルアプリケーション

このガイドの GitHub リポジトリには、AWS Lambda での Java の使用方法を示すサンプルアプリケーションが用意されています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS CloudFormation テンプレート、サポートリソースが含まれています。

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API Gateway ゲートウェイを使用して関数を呼び出します。

Lambda で一般的な Java フレームワークを実行する

- spring-cloud-function-samples — Spring クラウド関数フレームワークを使用して AWS Lambda 関数を作成する方法を示す Spring の例。
- サーバーレス Spring Boot アプリケーションのデモ — 一般的なSpring Bootアプリケーションを、SnapStart を使用または使用しないマネージド Java ランタイムでセットアップする方法、またはカスタムランタイムを使用した GraalVM ネイティブイメージとしてセットアップする方法を示す例。
- サーバーレス Micronaut アプリケーションのデモ — SnapStart を使用または使用しないマネージド Java ランタイムで Micronaut を使用する方法、またはカスタムランタイムを使用した GraalVM ネイティブイメージとして Micronaut を使用する方法を示す例。詳細については、「Micronaut/Lambda guides」を参照してください。
- サーバーレス Quarkus アプリケーションのデモ — SnapStart を使用または使用しないマネージド Java ランタイムで Quarkus を使用する方法、またはカスタムランタイムを使用した GraalVM ネ

イティブイメージとして Quarkus を使用する方法を示す例。詳細については、「Quarkus/Lambda guide」および「Quarkus/SnapStart guide」を参照してください。

Java での Lambda 関数を初めて使用する場合は、まず `java-basic` の例で始めます。Lambda イベントソースの使用を開始するには、`java-events` の例を参照してください。これらの両方の例では、Lambda の Java ライブラリ、環境変数、AWS SDK、AWS X-Ray SDK の使用を示しています。これらの例では、必要なセットアップは最小限となっており、コマンドラインから 1 分未満でデプロイできます。

Go による Lambda 関数の構築

Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。Go 関数を Lambda にデプロイするには、OS 専用ランタイム (provided ランタイム ファミリ) を使用します。

トピック

- Go ランタイムのサポート
- ツールとライブラリ
- Go の Lambda 関数ハンドラーの定義
- Go の AWS Lambda context オブジェクト
- .zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする
- コンテナイメージを使用して Go Lambda 関数をデプロイする
- Go の AWS Lambda 関数ログ作成
- AWS Lambda での Go コードの作成
- 環境変数の使用

Go ランタイムのサポート

Lambda の Go 1.x マネージドランタイムは非奨励になりました。Go 1.x ランタイムを使用する関数がある場合は、関数を `provided.al2023` または `provided.al2` に移行する必要があります。`provided.al2023` および `provided.al2` ランタイムには、arm64 アーキテクチャのサポート (AWS Graviton2 プロセッサー)、バイナリの小型化、若干の呼び出し時間短縮化など、`go1.x` と比べていくつか利点があります。

この移行ではコードの変更は必要ありません。必要な変更は、デプロイパッケージの構築方法と、関数の作成に使用するランタイムに関するもののみです。詳細については、「AWS コンピュートブログ」の「Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2」を参照してください。

OS 専用

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
OS 専用ランタイム	`provided.al2023`	Amazon Linux 2023			
OS 専用ランタイム	`provided.al2`	Amazon Linux 2			

ツールとライブラリ

Lambda は、Go ランタイム用の次のツールとライブラリを提供します。

- AWS SDK for Go: Go プログラミング言語用の公式の AWS SDK。
- github.com/aws/aws-lambda-go/lambda: Go 用の Lambda プログラミングモデルの実装。このパッケージは、ハンドラーを呼び出すために AWS Lambda で使用されます。
- github.com/aws/aws-lambda-go/lambdacontext: コンテキストオブジェクトからコンテキスト情報にアクセスするためのヘルパー。
- github.com/aws/aws-lambda-go/events: このライブラリは一般的なイベントソース統合のタイプの定義を提供します。
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip: このツールは、Windows で .zip ファイルアーカイブを作成するために使用することができます。

詳細については、GitHub の「aws-lambda-go」をご参照ください。

Lambda は、Go ランタイム用の次のサンプルアプリケーションを提供します。

Go のサンプル Lambda アプリケーション

- go-al2 — パブリック IP アドレスを返す Hello World 関数。このアプリは `provided.al2` カスタムランタイムを使用しています。
- blank-go – Lambda の Go ライブラリ、ログ記録、環境変数、AWS SDK の使用を示す Go 関数。このアプリは `go1.x` ランタイムを使用しています。

Go の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

Go で書き込まれた Lambda 関数は、Go 実行可能ファイルとして作成されます。Lambda 関数コードでは、Go の Lambda プログラミングモデルを実装する github.com/aws/aws-lambda-go/lambda パッケージを含める必要があります。加えて、ハンドラー関数および `main()` 関数の実装が必要となります。

Example Go Lambda 関数

```
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(ctx context.Context, event *MyEvent) (*string, error) {
 if event == nil {
  return nil, fmt.Errorf("received nil event")
 }
 message := fmt.Sprintf("Hello %s!", event.Name)
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

次に示すのはこの関数の入力例です。

```
{
  "name": "Jane"
```

```
}
```

次の点に注意してください。

- package main: Go では、`func main()` を含むパッケージは常に `main` と名付けられることが必要です。
- インポート: これを使用して、Lambda 関数に必要なライブラリを含めます。この場合、次のものが含まれます。
  - コンテキスト: Go の AWS Lambda context オブジェクト。
  - fmt: 関数の戻り値の書式に使用される Go フォーマット オブジェクト。
  - github.com/aws/aws-lambda-go/lambda: 前に説明した Go 用の Lambda プログラミングモデルを実装します。
- func HandleRequest(ctx context.Context, event *MyEvent) (*string, error): これは Lambda ハンドラーの署名です。Lambda 関数のエントリポイントであり、関数が呼び出されたときに実行されるロジックが含まれています。また、次に示すパラメータも含まれます。
  - ctx context.Context: Lambda 関数呼び出しへのランタイム情報を提供します。`ctx` は、Go の AWS Lambda context オブジェクト を介して利用できる情報を活用する宣言を行う変数です。
  - event *MyEvent: これは `MyEvent` をポイントする `event` という名前のパラメータです。Lambda 関数への入力を表します。
  - *string, error: ハンドラーは 2 つの値を返します。1 つ目は、Lambda 関数の結果を含む文字列へのポインタです。2 つ目はエラータイプで、エラーがない場合は `nil` となり、何か問題が発生した場合は標準的なエラー情報 を含みます。
  - return &message, nil: 2 つの値を返します。1 つ目は文字列メッセージへのポインタです。これは、入力イベントの `Name` フィールドを使用して作成された挨拶文です。2 つ目の値は `nil` であり、関数でエラーが発生しなかったことを示します。
- func main(): Lambda 関数コードが実行されるエントリポイント。これは必須です。

  `func main(){}` コードの括弧内に `lambda.Start(HandleRequest)` を追加すると、Lambda 関数が実行されます。Go 言語の規約に基づき、開き中括弧 `{` は `main` 関数シグネチャの直後に置く必要があります。

命名

provided.al2 および provided.al2023 ランタイム

.zip デプロイパッケージで provided.al2 または provided.al2023 ランタイムを使用する Go 関数の場合、関数コードを含む実行ファイルの名前は bootstrap である必要があります。.zip ファイルを使用して関数をデプロイする場合、bootstrap ファイルは .zip ファイルのルートにある必要があります。コンテナイメージで provided.al2 または provided.al2023 ランタイムを使用する Go 関数の場合、実行ファイルには任意の名前を使用できます。

ハンドラーには任意の名前を使用できます。コード内でハンドラー値を参照するには、_HANDLER 環境変数を使用できます。

go1.x ランタイム

go1.x ランタイムを使用する Go 関数の場合、実行ファイルとハンドラーは任意の名前を共有できます。例えば、ハンドラーの値を Handler に設定すると、Lambda はHandler 実行可能ファイル内の main() 関数を呼び出します。

Lambda コンソールの関数ハンドラー名を変更するには、ランタイム設定ペインで、[Edit] (編集) を選択します。

構造化されたタイプを使用した Lambda 関数ハンドラー

上記の例では、入力タイプは単純な文字列でした。ただし、構造化されたイベントを関数ハンドラーに渡すこともできます。

```
package main

import (
 "fmt"
 "github.com/aws/aws-lambda-go/lambda"
)

type MyEvent struct {
 Name string `json:"What is your name?"`
 Age  int    `json:"How old are you?"`
}

type MyResponse struct {
 Message string `json:"Answer"`
```

```
}

func HandleLambdaEvent(event *MyEvent) (*MyResponse, error) {
 if event == nil {
  return nil, fmt.Errorf("received nil event")
 }
 return &MyResponse{Message: fmt.Sprintf("%s is %d years old!", event.Name,
 event.Age)}, nil
}

func main() {
 lambda.Start(HandleLambdaEvent)
}
```

次に示すのはこの関数の入力例です。

```
{
    "What is your name?": "Jim",
    "How old are you?": 33
}
```

レスポンスは次のようになります。

```
{
    "Answer": "Jim is 33 years old!"
}
```

エクスポートするには、イベント構造体のフィールド名が大文字である必要があります。AWS イベントソースでイベントを処理するための詳細は、「aws-lambda-go/events」を参照してください。

有効なハンドラー署名

Go で Lambda 関数ハンドラーを構築するには複数の方法がありますが、次のルールに従う必要があります。

- ハンドラーは関数である必要があります。
- ハンドラーは 0 から 2 までの引数を取る場合があります。2 つの引数がある場合は、最初の引数が `context.Context` を実装する必要があります。
- ハンドラーは 0 から 2 までの引数を返す場合があります。単一の戻り値がある場合は、この値が `error` を実装する必要があります。2 つの戻り値がある場合には、2 番目の値が `error` を実装している必要があります。

次のリストは、有効なハンドラー署名の一覧です。 TIn と TOut は、 encoding/json 標準ライブラリと互換性のあるタイプを表しています。詳細については、「func アンマーシャリング」でこれらのタイプが逆シリアル化する方法を参照してください。

- `func ()`
- `func () error`
- `func (TIn) error`
- `func () (TOut, error)`
- `func (context.Context) error`
- `func (context.Context, TIn) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) (TOut, error)`

グローバルな状態を使用する

Lambda 関数のハンドラーコードとは別のグローバルな変数を宣言して変更することができます。さらに、ハンドラーがロードされている場合に実行される `init` 関数をハンドラーが宣言することがあります。これは、AWS Lambda でも標準の Go プログラムと同じように作動します。Lambda 関数の単一のインスタンスが、同時に複数のイベントを処理することはありません。

Example グローバル変数を使用する Go 関数

> **Note**
>
> このコードは AWS SDK for Go V2 を使用します。詳細については、「Getting Started with the AWS SDK for Go V2」を参照してください。

```
package main
```

```
import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
 "github.com/aws/aws-sdk-go-v2/service/s3/types"
 "log"
)

var invokeCount int
var myObjects []types.Object

func init() {
 // Load the SDK configuration
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("Unable to load SDK config: %v", err)
 }

 // Initialize an S3 client
 svc := s3.NewFromConfig(cfg)

 // Define the bucket name as a variable so we can take its address
 bucketName := "DOC-EXAMPLE-BUCKET"
 input := &s3.ListObjectsV2Input{
  Bucket: &bucketName,
 }

 // List objects in the bucket
 result, err := svc.ListObjectsV2(context.TODO(), input)
 if err != nil {
  log.Fatalf("Failed to list objects: %v", err)
 }
 myObjects = result.Contents
}

func LambdaHandler(ctx context.Context) (int, error) {
 invokeCount++
 for i, obj := range myObjects {
  log.Printf("object[%d] size: %d key: %s", i, obj.Size, *obj.Key)
 }
 return invokeCount, nil
}
```

```
func main() {
 lambda.Start(LambdaHandler)
}
```

Go の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を含むメソッドおよびプロパティを提供します。

Lambda コンテキストライブラリは、次のグローバル変数、メソッド、およびプロパティを提供します。

グローバル変数

- `FunctionName` - Lambda 関数の名前。
- `FunctionVersion` - 関数のバージョン。
- `MemoryLimitInMB` - 関数に割り当てられたメモリの量。
- `LogGroupName` - 関数のロググループ。
- `LogStreamName` — 関数インスタンスのログストリーム。

context メソッド

- `Deadline` — 実行がタイムアウトした日付 (Unix 時間のミリ秒単位) を返します。

context プロパティ

- `InvokedFunctionArn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `AwsRequestID` - 呼び出しリクエストの ID。
- `Identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
- `ClientContext` — (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。

context の呼び出し情報へのアクセス

Lambda 関数は、環境と呼び出しリクエストに関するメタデータにアクセスできます。これには、パッケージのコンテキストでアクセスできます。ハンドラーにはパラメータとして `context.Context` が含まれている必要があり、Lambda は関数に関する情報をコンテキス

トの Value プロパティに挿入します。lambdacontextのコンテキストにアクセスするために、context.Context ライブラリをインポートする必要があることに注意してください。

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
}
```

上記の例では、 lcはコンテキストオブジェクトがキャプチャした情報を使用し、その情報をlog.Print(lc.Identity.CognitoIdentityPoolID)出力するために使用される変数です。この場合は CognitoIdentityPoolID です。

以下の例では、コンテキストオブジェクトを使用して、Lambda 関数の完了にかかる時間をモニタリングする方法を紹介しています。これによって、予期されるパフォーマンスを分析し、必要な場合には関数コードを調整します。

```
package main

import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
```

```
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

.zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする

AWS Lambda 関数のコードは、スクリプトまたはコンパイルされたプログラム、さらにそれらの依存関係で構成されます。デプロイパッケージを使用して、Lambda に関数コードをデプロイします。Lambda は、コンテナイメージと .zip ファイルアーカイブの 2 種類のデプロイパッケージをサポートします。

このページでは、Go ランタイムのデプロイパッケージとして .zip ファイルを作成し、AWS Management Console、AWS Command Line Interface (AWS CLI)および AWS Serverless Application Model (AWS SAM) を使用して関数コードを AWS Lambda にデプロイするために .zip ファイルを使用する方法について説明します。

Lambda は POSIX ファイルアクセス許可を使用するため、.zip ファイルアーカイブを作成する前に、デプロイパッケージフォルダのアクセス許可を設定する必要がある場合があります。

セクション

macOS および Linux での .zip ファイルの作成

以下の手順は、`go build` コマンドを使用して実行ファイルをコンパイルし、Lambda 用の .zip ファイルデプロイパッケージを作成する方法を示しています。コードをコンパイルする前に、GitHub から Lambda パッケージをインストールしていることを確認してください。このモジュールは、ランタイムインターフェイスの実装を提供し、ランタイムインターフェイスは Lambda と関数コード間の相互作用を管理します。このライブラリをダウンロードするには、次のコマンドを実行します。

```
go get github.com/aws/aws-lambda-go/lambda
```

関数で AWS SDK for Go を使用している場合は、標準の SDK モジュールセットと、アプリケーションに必要な AWS サービス API クライアントをダウンロードしてください。SDK for Go のインストール方法については、「AWS SDK for Go V2 の使用開始」を参照してください。

指定されたランタイムファミリーの使用

Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。Go 関数を Lambda にデプロイするには、OS 専用ランタイム (provided ランタイム ファミリ) を使用します。

.zip デプロイパッケージを作成するには (macOS/Linux)

1. アプリケーションの main.go ファイルが含まれているプロジェクトディレクトリで、実行ファイルをコンパイルします。次の点に注意してください。

   - 実行ファイルには bootstrap という名前をつける必要があります。詳細については、「命名」を参照してください。
   - ターゲットの命令セットアーキテクチャを設定します。OS のみのランタイムは、arm64 と x86_64 の両方をサポートします。
   - オプションの lambda.norpc タグを使用して、Lambda ライブラリの Remote Procedure Call (RPC) コンポーネントを除外することができます。RPC コンポーネントは、Go 1.x ランタイムを使用している場合にのみ必要です。RPC を除外すると、デプロイパッケージのサイズが小さくなります。

   arm64 アーキテクチャの場合:

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   x86_64 アーキテクチャの場合:

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (オプション) Linux では、CGO_ENABLED=0 set を使用してパッケージをコンパイルする必要がある場合があります。

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   このコマンドは、標準の C ライブラリ (libc) バージョン用の安定したバイナリパッケージを作成します。このパッケージは、Lambda と他のデバイスでは異なる場合があります。

3. 実行可能ファイルを .zip ファイルにパッケージ化して、デプロイパッケージを作成します。

```
zip myFunction.zip bootstrap
```

> **Note**
>
> bootstrap ファイルは、.zip ファイルのルートに置く必要があります。

4. 関数を作成します。次の点に注意してください。

   - バイナリ名は bootstrap にする必要がありますが、ハンドラー名は何でもかまいません。詳細については、「命名」を参照してください。
   - `--architectures` オプションは、arm64 を使用している場合にのみ必須です。デフォルト値は x86_64 です。
   - `--role` には、実行ロールの Amazon リソースネーム (ARN) を指定します。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Windows での .zip ファイルの作成

次の手順は、GitHub から Windows 用の build-lambda-zip ツールをダウンロードし、実行可能ファイルをコンパイルして、.zip のデプロイパッケージを作成する方法を示しています。

> **Note**
>
> 上記を実行していない場合には、git をインストールした上で、お使いの Windows の `git` 環境変数に %PATH% の実行可能ファイルを追加します。

コードをコンパイルする前に、GitHub から Lambda ライブラリをインストールしていることを確認してください。このライブラリをダウンロードするには、次のコマンドを実行します。

```
go get github.com/aws/aws-lambda-go/lambda
```

関数で AWS SDK for Go を使用している場合は、標準の SDK モジュールセットと、アプリケーションに必要な AWS サービス API クライアントをダウンロードしてください。SDK for Go のインストール方法については、「AWS SDK for Go V2 の使用開始」を参照してください。

指定されたランタイムファミリーの使用

Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。Go 関数を Lambda にデプロイするには、OS 専用ランタイム (provided ランタイム ファミリ) を使用します。

.zip デプロイパッケージを作成するには (Windows)

1. GitHub から build-lambda-zip ツールをダウンロードします。

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. GOPATH のツールを使用して、.zip ファイルを作成します。Go のデフォルトのインストールがある場合、このツールは通常 %USERPROFILE%\Go\bin に置かれています。それ以外の場合は、Go ランタイムをインストールした場所に移動し、次のいずれかの操作を行います。

   cmd.exe

   cmd.exe で、ターゲットの命令セットアーキテクチャに応じて、次のいずれかを実行します。OS のみのランタイムは、arm64 と x86_64 の両方をサポートします。

   オプションの lambda.norpc タグを使用して、Lambda ライブラリの Remote Procedure Call (RPC) コンポーネントを除外することができます。RPC コンポーネントは、Go 1.x ランタイムを使用している場合にのみ必要です。RPC を除外すると、デプロイパッケージのサイズが小さくなります。

   Example — x86_64 アーキテクチャの場合

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

   Example — arm64 アーキテクチャの場合

   ```
   set GOOS=linux
   ```

```
set GOARCH=arm64
set CGO_ENABLED=0
go build -tags lambda.norpc -o bootstrap main.go
%USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

PowerShell

PowerShell で、ターゲットの命令セットアーキテクチャに応じて、次のいずれかを実行します。OS のみのランタイムは、arm64 と x86_64 の両方をサポートします。

オプションの `lambda.norpc` タグを使用して、Lambda ライブラリの Remote Procedure Call (RPC) コンポーネントを除外することができます。RPC コンポーネントは、Go 1.x ランタイムを使用している場合にのみ必要です。RPC を除外すると、デプロイパッケージのサイズが小さくなります。

x86_64 アーキテクチャの場合:

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

arm64 アーキテクチャの場合:

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. 関数を作成します。次の点に注意してください。

- バイナリ名は `bootstrap` にする必要がありますが、ハンドラー名は何でもかまいません。詳細については、「命名」を参照してください。
- `--architectures` オプションは、arm64 を使用している場合にのみ必須です。デフォルト値は x86_64 です。
- `--role` には、実行ロールの Amazon リソースネーム (ARN) を指定します。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

.zip ファイルを使用した Go Lambda 関数の作成と更新

.zip デプロイパッケージを作成すると、このパッケージを使用して新しい Lambda 関数を作成するか、既存の関数を更新できます。.zip パッケージをデプロイするには、Lambda コンソール、AWS Command Line Interface、Lambda API を使用します。AWS Serverless Application Model (AWS SAM) および AWS CloudFormation を使用して、Lambda 関数を作成および更新することもできます。

Lambda の .zip デプロイパッケージの最大サイズは 250 MB (解凍) です。この制限は、Lambda レイヤーを含む、更新するすべてのファイルの合計サイズに適用されることに注意してください。

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

コンソールを使用して .zip ファイルの関数を作成、更新する

新しい関数を作成するには、まずコンソールで関数を作成し、次に .zip アーカイブをアップロードする必要があります。既存の関数を更新するには、その関数のページを開き、同じ手順に従って更新した .zip ファイルを追加します。

.zip ファイルが 50 MB 未満の場合は、ローカルマシンから直接ファイルをアップロードして関数を作成または更新できます。50 MB を超える .zip ファイルの場合は、まず Amazon S3 バケットにパッケージをアップロードする必要があります。AWS Management Console を使用して Amazon S3 バケットにファイルをアップロードする手順については、「Amazon S3 の開始方法」を参照してください。AWS CLI を使用してファイルをアップロードするには、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> 既存のコンテナイメージ関数を変換して .zip アーカイブを使用することはできません。この場合は、新しい関数を作成する必要があります。

新しい関数を作成するには (コンソール)

1. Lambda コンソールの [関数] ページを開き、[関数の作成] を選択します。
2. [一から作成] を選択します。
3. [基本的な情報] で、以下を行います。
   a. [関数名] に、関数名を入力します。
   b. [Runtime (ランタイム)] で、`provided.al2023` を選択します。
4. (オプション) [アクセス権限] で、[デフォルトの実行ロールの変更] を展開します。新しい [実行ロール] を作成することも、既存のロールを使用することもできます。
5. [関数の作成]を選択します。Lambda は、選択したランタイムを使用して基本的な「Hello world」関数を作成します。

ローカルマシンから zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、.zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [.zip ファイル] をクリックします。
5. .zip ファイルをアップロードするには、次の操作を行います。
   a. [アップロード] をクリックし、ファイルセレクターで .zip ファイルを選択します。
   b. [開く] をクリックします。

c. [保存] をクリックします。

Amazon S3 バケットから .zip アーカイブをアップロードするには (コンソール)

1. Lambda コンソールの [関数ページ] で、新しい .zip ファイルをアップロードする関数を選択します。
2. [コード] タブを選択します。
3. [コードソース] ペインで、[アップロード元] をクリックします。
4. [Amazon S3 ロケーション] を選択します。
5. .zip ファイルの Amazon S3 リンク URL を貼り付けて、[保存] をクリックします。

AWS CLI を使用して .zip ファイルで関数を作成、更新する

AWS CLI を使用して新しい関数を作成したり、.zip ファイルを使用して既存の関数を更新したりできます。create-function コマンドと update-function-code を使用して、.zip パッケージをデプロイします。.zip ファイルが 50 MB 未満の場合は、ローカルビルドマシン上のファイルの場所から .zip パッケージをアップロードできます。サイズの大きいファイルの場合は、Amazon S3 バケットから .zip パッケージをアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

> **Note**
>
> AWS CLI を使用して Amazon S3 バケットから .zip ファイルをアップロードする場合、このバケットは関数と同じ AWS リージョン に配置する必要があります。

AWS CLI を含む .zip ファイルを使用して新しい関数を作成するには、以下を指定する必要があります。

- 関数の名前 (`--function-name`)
- 関数のランタイム (`--runtime`)
- 関数の実行ロール (`--role`) の Amazon リソースネーム (ARN)
- 関数コード内のハンドラーメソッド (`--handler`) の名前

.zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --code オプションを使用します。S3ObjectVersion パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=DOC-EXAMPLE-BUCKET,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

CLI を使用して既存の関数を更新するには、--function-name パラメータを使用して関数の名前を指定します。関数コードの更新に使用する .zip ファイルの場所も指定する必要があります。.zip ファイルがローカルビルドマシン上のフォルダにある場合は、次のコマンド例に示すように、--zip-file オプションを使用してファイルパスを指定します。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 バケット内の .zip ファイルの場所を指定するには、以下のコマンド例にある --s3-bucket および --s3-key オプションを使用します。--s3-object-version パラメータは、バージョン管理下のオブジェクトにのみ使用する必要があります。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API を使用して .zip ファイルで関数を作成、更新する

.zip ファイルアーカイブを使用して関数を作成および更新するには、以下の API オペレーションを使用します。

- CreateFunction

- UpdateFunctionCode

AWS SAM を使用して .zip ファイルで関数を作成、更新する

AWS Serverless Application Model (AWS SAM) は、AWS のサーバーレスアプリケーションの構築と実行のプロセスを合理化するのに役立つツールキットです。YAML または JSON テンプレートでアプリケーションのリソースを定義し、AWS SAM コマンドラインインターフェイス (AWS SAM CLI) を使用して、アプリケーションを構築、パッケージ化、デプロイします。AWS SAM テンプレートから Lambda 関数を構築すると、AWS SAM は関数コードと指定した任意の依存関係を含む .zip デプロイパッケージまたはコンテナイメージを自動的に作成します。AWS SAM を使用して Lambda 関数を構築およびデプロイする方法の詳細については、「AWS Serverless Application Model 開発者ガイドの」の「AWS SAM の開始方法」を参照してください。

AWS SAM を使用して、既存の .zip ファイルアーカイブを使用する Lambda 関数を作成できます。AWS SAM を使用して Lambda 関数を作成するには、.zip ファイルを Amazon S3 バケットまたはビルドマシンのローカルフォルダに保存します。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS SAM テンプレートでは、Lambda 関数は `AWS::Serverless::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `CodeUri` - 関数コードの Amazon S3 URI、ローカルフォルダへのパス、または FunctionCode オブジェクトに設定
- `Runtime` - 選択したランタイムに設定

AWS SAM では、.zip ファイルが 50 MB を超える場合、この .zip ファイルを最初に Amazon S3 バケットにアップロードする必要はありません。AWS SAM では、ローカルビルドマシン上の場所から、最大許容サイズ 250 MB (解凍) の .zip パッケージをアップロードできます。

AWS SAM で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS SAM 開発者ガイド」の「AWS::Serverless::Function」を参照してください。

例: provided.al2023 を使って Go 関数を構築するために AWS SAM を使用

1. 次のプロパティで AWS SAM テンプレートを作成します。

- [BuildMethod]: アプリケーションのコンパイラを指定します。go1.x を使用します。
- [ランタイム]: provided.al2023 を使用します。
- [CodeUri]: コードへのパスを入力します。
- [アーキテクチャ]: arm64 アーキテクチャ用の [arm64] を使用します。x86_64 命令セットアーキテクチャ用の場合、[amd64] を使用するか、または Architectures プロパティを削除します。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Metadata:
      BuildMethod: go1.x
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. sam build コマンドを使用して、実行ファイルをコンパイルします。

```
sam build
```

3. sam deploy コマンドを使用して、関数を Lambda にデプロイします。

```
sam deploy --guided
```

AWS CloudFormation を使用して .zip ファイルで関数を作成、更新する

AWS CloudFormation を使用して、.zip ファイルアーカイブを使用する Lambda 関数を作成できます。.zip ファイルから Lambda 関数を作成するには、最初にファイルを Amazon S3 バケットにアップロードする必要があります。AWS CLI を使用して Amazon S3 バケットにファイルをアップロードする方法については、「AWS CLI ユーザーガイド」の「オブジェクトの移動」を参照してください。

AWS CloudFormation テンプレートでは、Lambda 関数は `AWS::Lambda::Function` のリソースにより指定されます。このリソースで次のプロパティを設定し、.zip ファイルアーカイブを使用して関数を作成します。

- `PackageType` - `Zip` に設定
- `Code` - `S3Bucket` および `S3Key` フィールドに Amazon S3 バケット名と .zip ファイル名を入力
- `Runtime` - 選択したランタイムに設定

AWS CloudFormation が生成する .zip ファイルは、4 MB を超えることはできません。AWS CloudFormation で .zip ファイルを使用して関数をデプロイする方法の詳細については、「AWS CloudFormation ユーザーガイド」の「AWS::Lambda::Function」を参照してください。

依存関係の Go レイヤーを作成する

> **Note**
>
> Go のようなコンパイル済み言語の関数でレイヤーを使用しても、Python のようなインタープリター言語と同じメリットが得られない場合があります。Go はコンパイル済み言語なので、関数は初期化フェーズで共有アセンブリを手動でメモリに読み込む必要があり、コールドスタート時間が長くなる可能性があります。代わりに、コンパイル時にすべての共有コードを含めて、組み込みコンパイラ最適化機能を活用することをお勧めします。

このセクションでは、依存関係をレイヤーに含める方法について説明します。

Lambda は、`/opt/lib` ディレクトリ内のライブラリ、および `/opt/bin` ディレクトリ内のバイナリを自動的に検出します。Lambda がレイヤーのコンテンツを正しく検出できるように、次の構造でレイヤーを作成します。

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

レイヤーをパッケージ化したら、「the section called “レイヤーの作成と削除”」および「the section called “レイヤーの追加”」を参照してレイヤーの設定を完了してください。

コンテナイメージを使用して Go Lambda 関数をデプロイする

Go Lambda 関数のコンテナイメージを構築するには 2 つの方法があります。

- AWS の OS 専用ベースイメージを使用する

  Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。OS 専用のベースイメージを使用して、Lambda 用の Go イメージを構築します。イメージに Lambda との互換性を持たせるには、イメージに `aws-lambda-go/lambda` パッケージを含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することもできます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、イメージに `aws-lambda-go/lambda` パッケージを含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

Go 関数をデプロイするための AWS ベースイメージ

Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。OS 専用のベースイメージを使用して、Go 関数を Lambda にデプロイします。

OS 専用

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
OS 専用ランタイム	`provided.al2023`	Amazon Linux 2023			
OS 専用ランタイム	`provided.al2`	Amazon Linux 2			

Amazon Elastic コンテナレジストリ公開ギャラリー: gallery.ecr.aws/lambda/provided

Go 用ランタイムインターフェイスクライアント

`aws-lambda-go/lambda` パッケージには、ランタイムインターフェイスの実装が含まれています。イメージで `aws-lambda-go/lambda` を使用する方法の例については、AWS の OS 専用ベースイメージを使用する または 非 AWS ベースイメージを使用する を参照してください。

AWS の OS 専用ベースイメージを使用する

Go は、他のマネージドランタイムとは異なる方法で実装されています。Go は実行可能バイナリにネイティブにコンパイルするため、専用の言語ランタイムは必要ありません。OS 専用のベースイメージを使用して、Go 関数のコンテナイメージを構築します。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
al2023	OS 専用ランタイム	Amazon Linux 2023	GitHub の OS 専用ランタイムの Dockerfile	
al2	OS 専用ランタイム	Amazon Linux 2	GitHub の OS 専用ランタイムの Dockerfile	

これらのベースイメージの詳細については、Amazon ECR Public Gallery の「provided」を参照してください。

aws-lambda-go/lambda パッケージを Go ハンドラに含める必要があります。このパッケージにより、ランタイムインターフェイスを含む、Go のプログラミングモデルが実装されます。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Go
- Docker
- AWS Command Line Interface (AWS CLI) バージョン 2

provided.al2023 ベースイメージからイメージを作成する

`provided.al2023` ベースイメージを使用して Go 関数をビルドしデプロイするには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir hello
cd hello
```

2. 新しい Go モジュールを初期化します。

```
go mod init example.com/hello-world
```

3. Lambda ライブラリを新しいモジュールの依存関係として追加します。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 「main.go」という名前のファイルを作成し、テキストエディタで開きます。これは Lambda 関数のコードです。次のサンプルコードをテストに使用することも、独自のサンプルコードで置き換えることもできます。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)
```

```
func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. テキストエディタを使用して、プロジェクトディレクトリに Dockerfile を作成します。次の Dockerfile の例では、「マルチステージビルド」が使用されます。これにより、各ステップで異なるベースイメージを使用できます。「Go ベースイメージ」など、1 つのイメージを使用し、コードをコンパイルして実行可能なバイナリを構築できます。その後、最後の FROM ステートメントで `provided.al2023` など別のイメージを使用し、Lambda にデプロイするイメージを定義できます。ビルドプロセスは最終デプロイイメージとは分離されているため、最終イメージにはアプリケーションの実行に必要なファイルのみが含まれます。

   オプションの `lambda.norpc` タグを使用して、Lambda ライブラリの Remote Procedure Call (RPC) コンポーネントを除外することができます。RPC コンポーネントは、Go 1.x ランタイムを使用している場合にのみ必要です。RPC を除外すると、デプロイパッケージのサイズが小さくなります。

   Example — マルチステージビルド Dockerfile

   > **Note**
   >
   > Dockerfile で指定する Go のバージョン (たとえば、`golang:1.20`) が、アプリケーションの作成に使用した Go のバージョンと同じであることを確認してください。

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
```

```
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

6. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。provided.al2023 ベースイメージには、ランタイムインターフェイスエミュレーターが含まれています。

ローカルマシンでランタイムインターフェイスエミュレーターを実行するには

1. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。
   - docker-image はイメージ名、test はタグです。
   - ./main は Dockerfile からの ENTRYPOINT です。

```
docker run -d -p 9000:8080 \
--entrypoint /usr/local/bin/aws-lambda-rie \
docker-image:test ./main
```

このコマンドはイメージをコンテナとして実行し、localhost:9000/2015-03-31/functions/function/invocations でローカルエンドポイントを作成します。

2. 新しいターミナルウィンドウから、curl コマンドを使用して次のエンドポイントにイベントをポストします。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。関数によっては JSON ペイロードが必要な場合があります。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

3. コンテナ ID を取得します。

```
docker ps
```

4. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。

   - --region 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - 111122223333 を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から `repositoryUri` をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:
   - `docker-image:test` をお使いの Docker イメージの名前およびタグで置き換えます。
   - `<ECRrepositoryUri>` を、コピーした `repositoryUri` に置き換えます。URI の末尾には必ず `:latest` を含めてください。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

非 AWS ベースイメージを使用する

非 AWS ベースイメージからも、Go 用のコンテナイメージを構築できます。次のステップでは、Dockerfile の例で Alpine ベースイメージを使用しています。

aws-lambda-go/lambda パッケージを Go ハンドラに含める必要があります。このパッケージにより、ランタイムインターフェイスを含む、Go のプログラミングモデルが実装されます。

前提条件

このセクションの手順を完了するには、以下が必要です。

- Go
- Docker
- AWS Command Line Interface (AWS CLI) バージョン 2

代替ベースイメージからイメージを作成する

Alpine ベースイメージを使用して Go 関数をビルドおよびデプロイするには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir hello
cd hello
```

2. 新しい Go モジュールを初期化します。

```
go mod init example.com/hello-world
```

3. Lambda ライブラリを新しいモジュールの依存関係として追加します。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 「main.go」という名前のファイルを作成し、テキストエディタで開きます。これは Lambda 関数のコードです。次のサンプルコードをテストに使用することも、独自のサンプルコードで置き換えることもできます。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. テキストエディタを使用して、プロジェクトディレクトリに Dockerfile を作成します。次の Dockerfile の例では、Alpine ベースイメージを使用しています。

Example Dockerfile

> **Note**
>
> Dockerfile で指定する Go のバージョン (たとえば、golang:1.20) が、アプリケーションの作成に使用した Go のバージョンと同じであることを確認してください。

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build
COPY main.go .
RUN go build -o main main.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. Docker イメージを「Docker の構築」コマンドで構築します。次の例では、イメージを docker-image と名付けて test タグを付けます。

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> このコマンドは、ビルドマシンのアーキテクチャに関係なく、コンテナが Lambda の実行環境と互換性があることを確認する --platform linux/amd64 オプションを特定します。ARM64 命令セットアーキテクチャを使用して Lambda 関数を作成する場合は、代わりに --platform linux/arm64 オプションを使用するようにコマンドを変更してください。

(オプション) イメージをローカルでテストする

ランタイムインターフェイスエミュレーターを使用して、イメージをローカルでテストします。エミュレーターはイメージに組み込むことも、次の手順を使用してローカルマシンにインストールすることもできます。

ローカルマシンにランタイムインターフェイスエミュレーターをインストールして実行するには

1. プロジェクトディレクトリから次のコマンドを実行して、GitHub からランタイムインターフェイスエミュレーター (x86-64 アーキテクチャ) をダウンロードし、ローカルマシンにインストールします。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   arm64 エミュレータをインストールするには、前のコマンドの GitHub リポジトリ URL を次のように置き換えます。

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   arm64 エミュレーターをインストールするには、$downloadLink を次のように置き換えます。

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run コマンドを使用して、Docker イメージを起動します。次の点に注意してください。

   - `docker-image` はイメージ名、`test` はタグです。
   - `/main` は Dockerfile からの `ENTRYPOINT` です。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /main
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /main
   ```

   このコマンドはイメージをコンテナとして実行し、`localhost:9000/2015-03-31/functions/function/invocations` でローカルエンドポイントを作成します。

   > **Note**
   >
   > ARM64 命令セットアーキテクチャ用に Docker イメージをビルドした場合は、`--platform linux/amd64` の代わりに `--platform linux/arm64` オプションを使用してください。

3. イベントをローカルエンドポイントにポストします。

   Linux/macOS

   Linux および macOS では、次の `curl` コマンドを実行します。

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell で次の `Invoke-WebRequest` コマンドを実行します。

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

このコマンドは、空のイベントで関数を呼び出し、応答を返します。サンプル関数コードではなく独自の関数コードを使用している場合は、JSON ペイロードを使用して関数を呼び出すことをお勧めします。例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. コンテナ ID を取得します。

```
docker ps
```

5. 「docker kill」コマンドを使用してコンテナを停止します。このコマンドでは、3766c4ab331c を前のステップのコンテナ ID で置き換えます。

```
docker kill 3766c4ab331c
```

イメージのデプロイ

Amazon ECR にイメージをアップロードして Lambda 関数を作成するには

1. 「get-login-password」コマンドを実行して Amazon ECR レジストリに Docker CLI を認証します。
   - `--region` 値を Amazon ECR リポジトリを作成する AWS リージョン に設定します。
   - `111122223333` を AWS アカウント ID に置き換えます。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 「create-repository」コマンドを使用して Amazon ECR にリポジトリを作成します。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR リポジトリは Lambda 関数と同じ AWS リージョン に配置されている必要があります。

成功すると、次のようなレスポンスが表示されます。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
```

```
            "encryptionType": "AES256"
        }
    }
}
```

3. 前のステップの出力から repositoryUri をコピーします。
4. 「docker tag」コマンドを実行して、最新バージョンとしてローカルイメージを Amazon ECR リポジトリにタグ付けします。このコマンドで:

   - docker-image:test をお使いの Docker イメージの名前およびタグで置き換えます。
   - <ECRrepositoryUri> を、コピーした repositoryUri に置き換えます。URI の末尾には必ず :latest を含めてください。

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   例:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. 「docker push」コマンドを実行して Amazon ECR リポジトリにローカルイメージをデプロイします リポジトリ URI の末尾には必ず :latest を含めてください。

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. まだ作成済みでない場合、関数に「実行ロールの作成」を実行してください。次のステップではロールの Amazon リソースネーム (ARN) が必要です。
7. Lambda 関数を作成します。ImageUri には、先ほど使用したリポジトリ URI を指定します。URI の末尾には必ず :latest を含めてください。

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

> **Note**
>
> イメージが Lambda 関数と同じリージョンに配置されていれば、別の AWS アカウントのイメージを使用して関数を作成することができます。詳細については、「Amazon ECR クロスアカウント許可」を参照してください。

8. 関数を呼び出します。

```
aws lambda invoke --function-name hello-world response.json
```

次のような結果が表示されます。

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 関数の出力を確認するには、response.json ファイルをチェックします。

関数コードを更新するには、イメージを再構築し、新しいイメージを Amazon ECR リポジトリにアップロードしてから、update-function-code コマンドを使用してイメージを Lambda 関数にデプロイする必要があります。

Lambda は、イメージタグを特定のイメージダイジェストに解決します。これは、関数のデプロイに使用されたイメージタグを Amazon ECR 内の新しいイメージを指すように変更しても、Lambda は新しいイメージを使用するように自動的に関数を更新しないことを意味します。新しいイメージを同じ Lambda 関数にデプロイするには、Amazon ECR のイメージタグが同じままであっても、update-function-code コマンドを使用する必要があります。

Go の AWS Lambda 関数ログ作成

AWS Lambda は、ユーザーに代わって Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda ランタイム環境は、各呼び出しの詳細をログストリームに送信し、関数のコードからのログやその他の出力を中継します。詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

このページでは、AWS Command Line Interface、Lambda コンソール、または CloudWatch コンソールを使用して、Lambda 関数のコードからログ出力を生成する方法、またはアクセスログを生成する方法について説明します。

セクション

- ログを返す関数の作成
- Lambda コンソールの使用
- CloudWatch コンソールの使用
- AWS Command Line Interface (AWS CLI) を使用する
- ログの削除

ログを返す関数の作成

関数コードからログを出力するには、fmt パッケージのメソッドか、stdout または stderr に書き込む任意のログ記録のライブラリを使用します。以下の例では、ログパッケージを使用しています。

Example main.go - ログ記録

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example ログの形式

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 39 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

Go ランタイムは、呼び出しごとに START、END、および REPORT の各行を記録します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。

- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- Sampled - トレースされたリクエストの場合、サンプリング結果。

Lambda コンソールの使用

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/**###**)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。　X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2

- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
```

```
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。


```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
```

```
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

AWS Lambda での Go コードの作成

Lambda アプリケーションのトレース、デバッグ、最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレースされたデータを X-Ray に送信するには、以下の 2 つの SDK ライブラリのいずれかを使用します。

- AWS Distro for OpenTelemetry (ADOT) – 安全で、本番環境に対応し、AWS でサポートされている OpenTelemetry (OTel) SDK のディストリビューションです。
- AWS X-Ray SDK for Go – トレースデータを生成して X-Ray に送信するための SDK。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。これらのソリューションのいずれかを使用して、X-Ray でエンドツーエンドのトレーシングを実装することができます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

Go 関数の計測に対する ADOT の使用

ADOT は、Otel SDK を使用してテレメトリデータを収集するために必要なすべてをパッケージ化した、フルマネージド型の Lambda レイヤーを提供します。このレイヤーを使用すると、関数コードを変更する必要はなしで、Lambda 関数を計測できます。また、このレイヤーは、OTel でのカスタムな初期化を実行するように構成することもできます。詳細については、ADOT のドキュメントにある「Lambda 上での ADOT Collector のカスタム設定」を参照してください。

Go ランタイムの場合は、AWS 管理の Lambda layer for ADOT Go を追加して、関数を自動的に計測することが可能です。このレイヤーを追加する方法の詳しい手順については、「ADOT ドキュメント」で「Go に対する AWS Distro for OpenTelemetry Lambda のサポート」を参照してください。

Go 関数の計測のための X-Ray SDK の使用

AWS X-Ray SDK for Go を使用して、Lambda 関数がアプリケーション内の他のリソースに対して行う呼び出しの詳細を記録することもできます。SDK を取得するには、`go get` を使用して GitHub リポジトリからダウンロードします。

```
go get github.com/aws/aws-xray-sdk-go
```

AWS SDK クライアントを実装するには、クライアントを `xray.AWS()` メソッドに渡します。その後、このメソッドの `WithContext` バージョンを使用することで、呼び出しをトレースできます。

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

正しい依存関係を追加し、必要なコード変更を行った後、Lambda コンソールまたは API を介して関数の設定でトレースをアクティブにします。

Lambda コンソールを使用してトレースを有効化する

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。
4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [保存] をクリックします。

Lambda API でのトレースのアクティブ化

AWS CLI または AWS SDK で Lambda 関数のトレースを設定するには、次の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation によるトレースのアクティブ化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
```

```
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray トレースの解釈

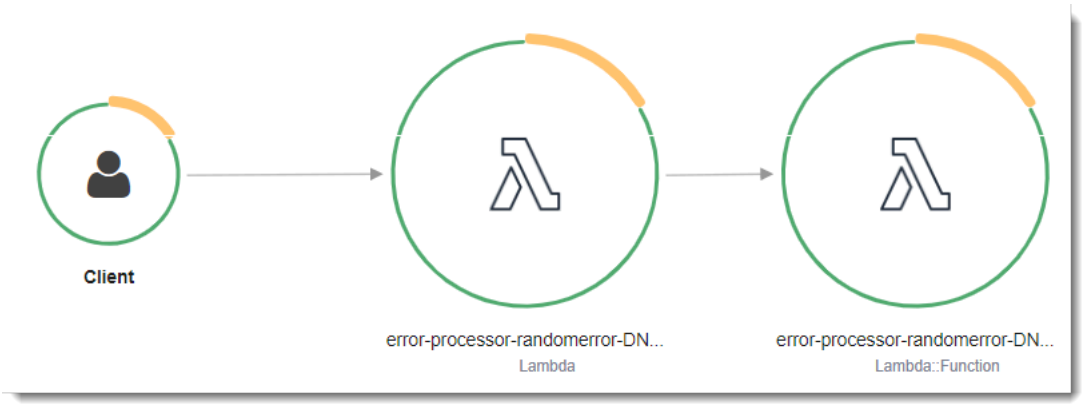
関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロググループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

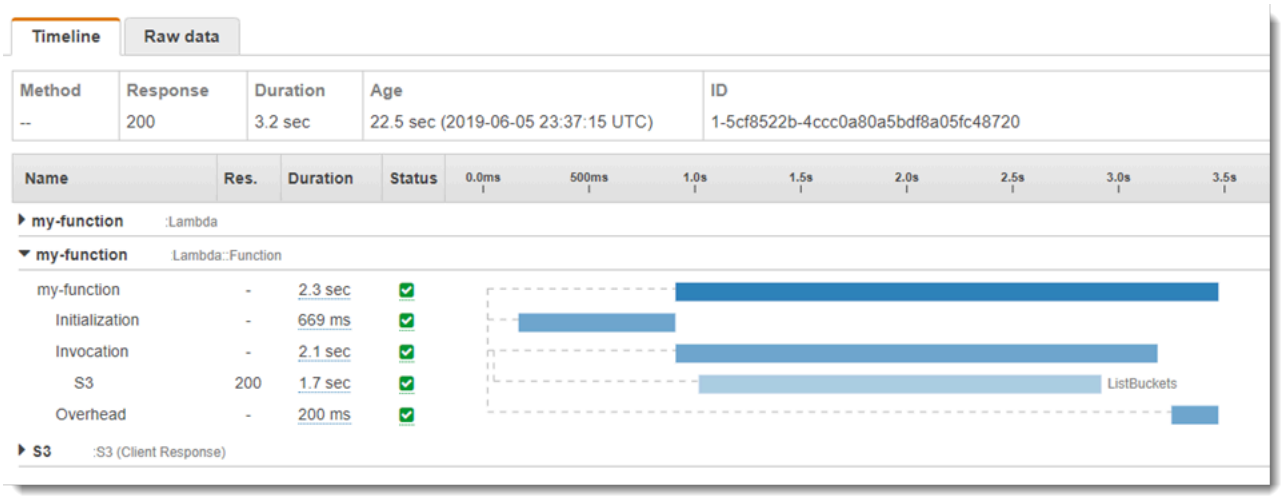
X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは `AWS::Lambda` の起点があり、もう 1 つは `AWS::Lambda::Function` の起点があります。`AWS::Lambda` セグメントにエラーが表示される場合は、Lambda サービスに問題があります。`AWS::Lambda::Function` セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、「AWS X-Rayデベロッパーガイド」の「AWS X-Ray SDK for Go」を参照してください。

> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

環境変数の使用

Go の環境変数にアクセスするには、Getenv 関数を使用します。

次にこれを実行する方法を説明します。この関数は、印刷した結果をフォーマットする fmt パッケージと環境変数にアクセスできる独立したプラットフォームシステムインターフェースである os パッケージをインポートすることに注意してください。

```
package main

import (
 "fmt"
 "os"
 "github.com/aws/aws-lambda-go/lambda"
)

func main() {
 fmt.Printf("%s is %s. years old\n", os.Getenv("NAME"), os.Getenv("AGE"))

}
```

Lambda ランタイムによって設定される環境変数のリストについては、「定義されたランタイム環境変数」を参照してください。

C# による Lambda 関数の構築

マネージド型 .NET 6 または .NET 8 ランタイム、カスタムランタイム、またはコンテナイメージを使用して Lambda で .NET アプリケーションを実行できます。アプリケーションコードをコンパイルしたら、.zip ファイルまたはコンテナイメージとして Lambda にデプロイできます。Lambda は、次の .NET 言語のランタイムをサポートしています。

.NET

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
.NET 8	`dotnet8`	Amazon Linux 2023			
.NET 6	`dotnet6`	Amazon Linux 2	2024 年 11 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日

.NET 開発環境のセットアップ

Lambda 関数の開発と構築には、Microsoft Visual Studio、Visual Studio Code、JetBrains Rider など、一般的に利用できる .NET 統合開発環境 (IDE) をどれでも使用できます。開発作業を簡素化するために、AWS には、.NET プロジェクトテンプレートのセットと `Amazon.Lambda.Tools` コマンドラインインターフェイス (CLI) が用意されています。

次の .NET CLI コマンドを実行して、これらのプロジェクトテンプレートとコマンドラインツールをインストールします。

.NETプロジェクトテンプレートのインストール

プロジェクトテンプレート (.NET 8) をインストールするには:

```
dotnet new install Amazon.Lambda.Templates
```

プロジェクトテンプレート (.NET 6) をインストールするには:

```
dotnet new --install Amazon.Lambda.Templates
```

> **Note**
>
> .NET 6 のマネージド Lambda ランタイムを使用している場合は、.NET 8 を使用するようにアップグレードすることをお勧めします。詳細については、 AWS コンピューティングブログの「AWS Lambda ランタイムアップグレードの管理」および「AWS Lambda の .NET 8 ランタイムの紹介」を参照してください。

CLI ツールのインストールと更新

`Amazon.Lambda.Tools` CLI をインストール、更新、アンインストールするには、次のコマンドを実行します。

コマンドラインツールをインストールするには:

```
dotnet tool install -g Amazon.Lambda.Tools
```

コマンドラインツールを更新するには:

```
dotnet tool update -g Amazon.Lambda.Tools
```

コマンドラインツールをインストールするには:

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

C# の Lambda 関数ハンドラーの定義

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

関数が呼び出され、Lambda が関数のハンドラーメソッドを実行すると、関数に 2 つの引数が渡されます。最初の引数は `event` オブジェクトです。別の AWS のサービスが関数を呼び出すと、その `event` オブジェクトには、関数が呼び出された原因となったイベントに関するデータが含まれます。例えば、API Gateway の `event` オブジェクトには、パス、HTTP メソッド、および HTTP ヘッダーに関する情報が含まれています。正確なイベント構造は、関数を呼び出す AWS のサービスによって異なります。個々のサービスのイベント形式の詳細については、「他のサービスの統合」を参照してください。

Lambda は関数に `context` オブジェクトも渡します。このオブジェクトには、呼び出し、関数、および実行環境に関する情報が含まれます。詳細については、「the section called “Context”」を参照してください。

すべての Lambda イベントのネイティブ形式は、JSON 形式のイベントを表すバイトストリームです。関数の入力パラメータと出力パラメータがタイプ `System.IO.Stream` でない限り、それらをシリアル化する必要があります。`LambdaSerializer` アセンブリ属性を設定して、使用するシリアライザーを指定します。詳細については、「the section called “Lambda 関数のシリアル化”」を参照してください。

トピック

- Lambda 用の .NET 実行モデル
- クラスライブラリハンドラー
- 実行可能アセンブリハンドラー
- Lambda 関数のシリアル化
- Lambda Annotations Framework による関数コードの簡略化
- Lambda 関数ハンドラーの制限

Lambda 用の .NET 実行モデル

.NET で Lambda 関数を実行するための実行モデルには、クラスライブラリアプローチと実行可能アセンブリアプローチの 2 種類があります。

クラスライブラリアプローチでは、呼び出す関数の AssemblyName、ClassName、Method を示す文字列を Lambda に渡します。この文字列の形式についての詳細は、「the section called “クラスライブラリハンドラー”」を参照してください。関数の初期化フェーズでは、関数のクラスが初期化され、コンストラクタ内のすべてのコードが実行されます。

実行可能アセンブリアプローチでは、C# 9 の最上位ステートメント機能を使用します。この方法では、関数の呼び出しコマンドを受信するたびに Lambda が実行する実行可能アセンブリが生成されます。Lambda には、実行する実行可能アセンブリの名前のみを指定します。

以下のセクションでは、これら 2 つの方法の関数コードの例を示しています。

クラスライブラリハンドラー

次の Lambda 関数コードは、クラスライブラリアプローチを使用する Lambda 関数のハンドラーメソッド (FunctionHandler) の例を示しています。このサンプル関数では、Lambda は、関数を呼び出すイベントを API Gateway から受信しています。この関数は、データベースからレコードを読み取り、API Gateway レスポンスの一部としてレコードを返します。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
```

```
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Lambda 関数を作成するときは、関数のハンドラーに関する情報をハンドラー文字列の形式で Lambda に提供する必要があります。これにより、関数の呼び出し時に実行するコードのメソッドを Lambda に指示します。C# では、クラスライブラリアプローチを使用する場合のハンドラー文字列の形式は次のようになります。

`ASSEMBLY::TYPE::METHOD`。ここで、

- `ASSEMBLY` は、アプリケーションの .NET アセンブリファイルの名前です。`Amazon.Lambda.Tools` CLI を使用してアプリケーションを構築し、.csproj ファイルの `AssemblyName` プロパティを使用してアセンブリ名を設定しない場合、`ASSEMBLY` は単に .csproj ファイルの名前になります。
- `TYPE` は、`Namespace` と `ClassName` からなるハンドラー型の正式名称となります。
- `METHOD` は、コード内の関数ハンドラーメソッドの名前です。

ここに示すコード例では、アセンブリに `GetProductHandler` という名前が付けられている場合、ハンドラー文字列は `GetProductHandler::GetProductHandler.Function::FunctionHandler` になります。

実行可能アセンブリハンドラー

次の例では、Lambda 関数を実行可能アセンブリとして定義します。このコードのハンドラーメソッドには `Handler` という名前が付いています。実行可能アセンブリを使用する場合は、Lambda ランタイムをブートストラップする必要があります。これを行うには、`LambdaBootstrapBuilder.Create` メソッドを使用します。このメソッドは、関数がハンドラーとして使用するメソッドと、使用する Lambda シリアライザーを入力として受け取ります。

上位のステートメントの使用の詳細については、AWS コンピューティングブログの「AWS Lambda 用の .NET 6 ランタイムのご紹介」を参照してください。

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();
```

```
await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = input.PathParameters["id"];

    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

実行可能アセンブリを使用する場合、コードの実行方法を Lambda に指示するハンドラー文字列はアセンブリの名前です。この例では、GetProductHandler になります。

Lambda 関数のシリアル化

Lambda 関数が、Stream オブジェクト以外の入出力タイプを使用する場合は、アプリケーションにシリアル化ライブラリを追加する必要があります。シリアル化は、System.Text.Json および Newtonsoft.Json が提供する標準のリフレクションベースのシリアル化を使用するか、ソース生成のシリアル化を使用して実装できます。

ソース生成のシリアル化を使用する

ソース生成のシリアル化は NETバージョン 6 以降の機能で、コンパイル時にシリアル化コードを生成できます。これにより、リフレクションが不要になり、関数のパフォーマンスが向上します。ソース生成のシリアル化を関数で使用するには、以下を実行します。

- JsonSerializerContext を継承する新しい部分クラスを作成し、シリアル化または逆シリアル化を必要とするすべての型の JsonSerializable 属性を追加します。
- LambdaSerializer を設定して SourceGeneratorLambdaJsonSerializer<T> を使用します。
- アプリケーションコード内の手動シリアル化または逆シリアル化をすべて更新して、新しく作成したクラスを使用するようにします。

ソース生成のシリアル化を使用する関数の例を次のコードに示します。

```
[assembly:
 LambdaSerializer(typeof(SourceGeneratorLambdaJsonSerializer<CustomSerializer>))]

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord,
 CustomSerializer.Default.Product)
        };
    }
}

[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{

}
```

> **Note**
>
> Lambda でネイティブの事前コンパイル (AOT) を使用する場合は、ソース生成のシリアル化を使用する必要があります。

リフレクションベースのシリアル化を使用する

AWS は、アプリケーションにシリアル化を迅速に追加できる事前構築済みライブラリを提供しています。これは、Amazon.Lambda.Serialization.SystemTextJson または Amazon.Lambda.Serialization.Json NuGet パッケージを使用して設定します。背後では、Amazon.Lambda.Serialization.SystemTextJson は System.Text.Json を使用してシリアル化タスクを実行し、Amazon.Lambda.Serialization.Json は Newtonsoft.Json パッケージを使用します。

ILambdaSerializer インターフェイスの実装によって独自のシリアル化ライブラリを作成することもできます。これは、Amazon.Lambda.Core ライブラリの一部として使用できます。このインターフェイスは 2 つのメソッドを定義します。

- `T Deserialize<T>(Stream requestStream);`

  このメソッドを実装して、Invoke API から Lambda 関数ハンドラーに渡されるオブジェクトにリクエストのペイロードを逆シリアル化することができます。
- `T Serialize<T>(T response, Stream responseStream);`

  このメソッドを実装して、Lambda 関数のハンドラーから返される結果を Invoke API オペレーションが返すレスポンスペイロードにシリアル化することができます。

Lambda Annotations Framework による関数コードの簡略化

Lambda Annotations は、.NET 6 と.NET 8 用のフレームワークで、C# を使用して Lambda 関数を簡単に記述できます。Annotations Framework では、通常のプログラミングモデルを使用して記述された Lambda 関数のコードの多くを置き換えることができます。このフレームワークを使用して記述されたコードでは、ビジネスロジックに集中できるより単純な式が使用されます。

次のコード例は、Annotations Framework を使用して Lambda 関数の記述を簡単にする方法を示しています。最初の例は、通常の Lambda プログラムモデルを使用して記述されたコードを示し、2 番目の例は、Annotations Framework を使用した同等のコードを示しています。

```
public APIGatewayHttpApiV2ProxyResponse LambdaMathAdd(APIGatewayHttpApiV2ProxyRequest
 request, ILambdaContext context)
{
    if (!request.PathParameters.TryGetValue("x", out var xs))
    {
        return new APIGatewayHttpApiV2ProxyResponse
```

```
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    if (!request.PathParameters.TryGetValue("y", out var ys))
    {
        return new APIGatewayHttpApiV2ProxyResponse
        {
            StatusCode = (int)HttpStatusCode.BadRequest
        };
    }
    var x = int.Parse(xs);
    var y = int.Parse(ys);
    return new APIGatewayHttpApiV2ProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = (x + y).ToString(),
        Headers = new Dictionary#string, string> { { "Content-Type", "text/plain" } }
    };
}
```

```
[LambdaFunction]
[HttpApi(LambdaHttpMethod.Get, "/add/{x}/{y}")]
public int Add(int x, int y)
{
    return x + y;
}
```

Lambda Annotations を使用してコードを簡略化する方法を示す別の例については、awsdocs/aws-doc-sdk-examples GitHub リポジトリにあるこの「cross-service example application」を参照してください。フォルダ PamApiAnnotations は、メイン function.cs ファイルで Lambda Annotations を使用しています。比較のために、PamApi フォルダには、通常の Lambda プログラミングモデルを使用して記述された同等のファイルがあります。

Annotations Framework はソースジェネレーターを使用して、Lambda プログラミングモデルから、2 番目の例にあるコードに変換するコードを生成します。

.NET で Lambda Annotations を使用する方法の詳細については、次のリソースを参照してください。

- aws/aws-lambda-dotnet GitHub リポジトリ。

- AWS 開発者ツールブログの「Introducing .NET Annotations Lambda Framework (Preview)」。
- `Amazon.Lambda.Annotations` NuGet パッケージ。

Lambda Annotations Framework による依存関係インジェクション

Lambda Annotations Framework を使用して、使い慣れた構文を用いて Lambda 関数に依存関係インジェクションを追加することもできます。`[LambdaStartup]` 属性を `Startup.cs` ファイルに追加すると、Lambda Annotations Framework がコンパイル時に必要なコードを生成します。

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Lambda 関数は、コンストラクタインジェクションを使用するか、`[FromServices]` 属性を使用して個々のメソッドに挿入することで、サービスを挿入できます。

```

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
```

```
        return await this._repo.GetById(id);
    }
}
```

Lambda 関数ハンドラーの制限

ハンドラー署名にはいくつかの制限があることに注意してください。

- ハンドラーメソッドとその依存関係内で unsafe コンテキストを使用できますが、unsafe ではなく、ハンドラー署名でポインター型を使用する場合があります。詳細については、Microsoft Docs ウェブサイトの「unsafe (C# Reference)」を参照してください。
- params キーワードを使用して可変数のパラメータを渡さなかったり、可変数のパラメータをサポートするために使用する入力パラメータまたは戻りパラメータとして ArgIterator を使用できない場合があります。
- ハンドラーは、IList<T> Sort<T>(IList<T> input) などの汎用メソッドではない場合があります。
- async void 署名を持つ Async ハンドラーはサポートされていません。

.zip ファイルアーカイブを使用して C# Lambda 関数を構築し、デプロイする

.NET デプロイパッケージ (.zip ファイルアーカイブ) は、関数のコンパイル済みアセンブリと、そのアセンブリのすべての依存関係で構成されています。このパッケージには、*proj*`.deps.json` ファイルも含まれています。これは、関数のすべての依存関係と、.NET ランタイムを設定するために使用される *proj*`.runtimeconfig.json` ファイルを、.NET ランタイムに伝達します。

個々の Lambda 関数をデプロイするには、`Amazon.Lambda.Tools` .NET Lambda Global CLI を使用できます。`dotnet lambda deploy-function` コマンドを使用すると、.zip デプロイパッケージが自動的に作成され、Lambda にデプロイされます。ただし、.NET アプリケーションを AWS にデプロイするには、AWS Serverless Application Model (AWS SAM) や AWS Cloud Development Kit (AWS CDK) などのフレームワークを使用することをお勧めします。

サーバーレスアプリケーションは通常、Lambda 関数とその他のマネージド AWS のサービスを組み合わせて構成され、連携して特定のビジネスタスクを実行します。AWS SAM と AWS CDK は、Lambda 関数を大規模に他の AWS のサービスと組み合わせて構築し、デプロイするのを簡略化します。AWS SAM テンプレート仕様は、サーバーレスアプリケーションを構成する Lambda 関数、API、アクセス許可、設定、およびその他の AWS リソースを記述するためのシンプルで簡潔な構文を提供します。AWS CDK を使用すると、クラウドインフラストラクチャをコードとして定義し、最新のプログラミング言語と .NET などのフレームワークを使用して、信頼性が高く、スケーラブルで、コスト効率の高いアプリケーションをクラウドで構築することができます。AWS CDK と AWS SAM はどちらも、.NET Lambda Global CLI を使用して関数をパッケージ化します。

.NET Core CLI を使用することで、C# の関数で Lambda レイヤーを使用することは可能ですが、使用しないことをお勧めします。レイヤーを使用する C# の関数では、初期化フェーズ 中に共有アセンブリを手動でメモリに読み込みます。これにより、コールドスタート時間が長くなる可能性があります。代わりに、コンパイル時にすべての共有コードを含めて、.NET コンパイラの組み込み最適化機能を活用してください。

AWS SAM、AWS CDK、および .NET Lambda Global CLI を使用して .NET Lambda 関数を構築およびデプロイする手順については、以下のセクションを参照してください。

トピック

- .NET Lambda Global CLI を使用する
- AWS SAM を使用した C# Lambda 関数のデプロイ
- AWS CDK を使用した C# Lambda 関数のデプロイ

- ASP.NET アプリケーションのデプロイ

.NET Lambda Global CLI を使用する

.NET CLI および .NET Lambda Global Tools 拡張機能 (`Amazon.Lambda.Tools`) は、.NET ベースの Lambda アプリケーションを作成し、パッケージ化して、Lambda にデプロイするためのクロスプラットフォームな方法を提供します。このセクションでは、.NET CLI と Amazon Lambda テンプレートを使用して新しい Lambda .NET プロジェクトを作成する方法と、`Amazon.Lambda.Tools` を使用してそれらをパッケージ化してデプロイする方法を学習します。

トピック

前提条件

.NET 8 SDK

まだインストールしていない場合は、.NET 8 SDK とランタイムをインストールします。

AWS Amazon.Lambda.Templates .NET プロジェクトテンプレート

Lambda 関数コードを生成するには、Amazon.Lambda.Templates NuGet パッケージを使用します。このテンプレートパッケージをインストールするには、以下のコマンドを実行します。

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI Tools

Lambda 関数を作成するには、Amazon.Lambda.Tools .NET Global Tools 拡張機能を使用します。Amazon.Lambda.Tools をインストールするには、以下のコマンドを実行します。

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 拡張機能の詳細については、GitHub の「AWS Extensions for .NET CLI」リポジトリを参照してください。

.NET CLI を使用して .NET プロジェクトを作成する

.NET CLI では、コマンドラインから .NET プロジェクトを作成するために `dotnet new` コマンドを使用します。Lambda には、`Amazon.Lambda.Templates` NuGet パッケージを使用する追加のテンプレートが用意されています。

このパッケージをインストールしたら、次のコマンドを実行して、使用可能なテンプレートのリストを表示します。

```
dotnet new list
```

テンプレートに関する詳細を確認するには、`help` オプションを使用します。例えば、`lambda.EmptyFunction` テンプレートの詳細を表示するには、以下のコマンドを実行します。

```
dotnet new lambda.EmptyFunction --help
```

.NET Lambda 関数の基本テンプレートを作成するには、`lambda.EmptyFunction` テンプレートを使用します。これにより、入力として文字列を受け取り、`ToUpper` メソッドを使用して大文字に変換する単純な関数が作成されます。このテンプレートでは、以下のオプションがサポートされています。

- `--name` - 関数の名前。
- `--region` - 関数を作成する AWS リージョン。
- `--profile` - AWS SDK for .NET 認証情報ファイルにあるプロファイルの名前。.NET の認証情報プロファイルの詳細については、「.NET 用 AWS SDK 開発者ガイド」の「AWS 認証情報を設定」を参照してください。

この例では、デフォルトのプロファイルと AWS リージョン設定を使用して、`myDotnetFunction` という名前の新しい空の関数を作成します。

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

このコマンドは、プロジェクトディレクトリに以下のファイルとディレクトリを作成します。

```
### myDotnetFunction
    ### src
```

```
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

src/myDotnetFunction ディレクトリで次のファイルを調べます。

- aws-lambda-tools-defaults.json: Lambda 関数をデプロイするときに指定するコマンドラインオプションの場所です。以下に例を示します。

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- Function.cs: Lambda ハンドラの関数コード。これは、デフォルトの Amazon.Lambda.Coreライブラリ とデフォルトの LambdaSerializer 属性が含まれる C# テンプレートです。要件とオプションのシリアル化に関する詳細は、「Lambda 関数のシリアル化」を参照してください。これには、Lambda 関数コードに適用するために編集できるサンプル関数も含まれています。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{

    /// <summary>
```

```
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- myDotnetFunction.csproj: アプリケーションを構成するファイルとアセンブリをリスト表示する MSBuild ファイル。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme: このファイルを使用して Lambda 関数をドキュメント化します。

`myfunction/test` ディレクトリで次のファイルを調べます。

- myDotnetFunction.Tests.csproj: 既に説明したように、これは、テストプロジェクトを構成するファイルとアセンブリをリスト表示するための、MSBuild ファイルです。`Amazon.Lambda.Core`

ライブラリが含まれているため、関数のテストに必要な Lambda テンプレートをシームレスに統合できることにも留意してください。

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs: src ディレクトリに含まれているのと同様の C# コードテンプレートファイルです。このファイルを編集して関数の本番稼働コードを写し、本番稼働環境にアップロードする前に Lambda 関数をテストします。

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

.NET CLI を使用して .NET プロジェクトをデプロイする

デプロイパッケージを構築して Lambda にデプロイするには、Amazon.Lambda.Tools CLI ツールを使用します。前のステップで作成したファイルから関数をデプロイするには、まず、関数の .csproj ファイルが含まれるフォルダに移動します。

```
cd myDotnetFunction/src/myDotnetFunction
```

コードを .zip デプロイパッケージとして Lambda にデプロイするには、以下のコマンドを実行します。独自の関数名を選択してください。

```
dotnet lambda deploy-function myDotnetFunction
```

デプロイ中、ウィザードでは [the section called “実行ロール (関数に対する、他のリソースにアクセスするためのアクセス許可)”] を選択するよう求められます。この例では、lambda_basic_role を選択します。

関数をデプロイしたら、dotnet lambda invoke-function コマンドを使用してクラウドでテストできます。lambda.EmptyFunction テンプレートのサンプルコードでは、--payload オプションを使用して文字列を渡すことで関数をテストできます。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

関数が正常にデプロイされると、以下のような出力が表示されます。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

.NET CLI で Lambda レイヤーを使用する

> **Note**
>
> C# のようなコンパイル済み言語の関数でレイヤーを使用しても、Python のようなインタープリター言語と同じメリットが得られない場合があります。C# はコンパイル済み言語なので、関数は初期化フェーズで共有アセンブリを手動でメモリに読み込む必要があり、コールドスタート時間が長くなる可能性があります。代わりに、コンパイル時にすべての共有コードを含めて、組み込みコンパイラ最適化機能を活用することをお勧めします。

.NET CLI では、レイヤーの公開や、レイヤーを使用する C# 関数のデプロイに役立つコマンドがサポートされています。指定した Amazon S3 バケットにレイヤーを公開するには、自身の `.csproj` と同じディレクトリで次のコマンドを実行します。

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

その後、.NET CLI を使用して関数をデプロイする際に、次のコマンドで使用するレイヤーの ARN を指定します。

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

Hello World 関数の完全な例については、blank-csharp-with-layer のサンプルを参照してください。

AWS SAM を使用した C# Lambda 関数のデプロイ

AWS Serverless Application Model (AWS SAM) は、AWS のサーバーレスアプリケーションの構築と実行のプロセスを合理化するのに役立つツールキットです。YAML または JSON テンプレートでアプリケーションのリソースを定義し、AWS SAM コマンドラインインターフェイス (AWS SAM CLI) を使用して、アプリケーションを構築、パッケージ化、デプロイします。AWS SAM テンプレートから Lambda 関数を構築すると、AWS SAM は関数コードと指定した任意の依存関係を含む .zip デプロイパッケージまたはコンテナイメージを自動的に作成します。そして、AWS SAM は AWS CloudFormation スタックを使用して関数をデプロイします。AWS SAM を使用して Lambda 関数を構築およびデプロイする方法の詳細については、「AWS Serverless Application Model 開発者ガイドの」の「AWS SAM の開始方法」を参照してください。

以下の手順は、AWS SAM を使用してサンプルの .NET Hello World アプリケーションをダウンロード、構築、およびデプロイする方法を示しています。このサンプルアプリケーションは、Lambda 関数と Amazon API Gateway エンドポイントを使用して基本的な API バックエンドを実装します。API Gateway エンドポイントに HTTP GET リクエストを送信すると、API Gateway が Lambda 関数を呼び出します。この関数は、リクエストを処理する Lambda 関数インスタンスの IP アドレスと共に「hello world」メッセージを返します。

AWS SAM を使用してアプリケーションを構築し、デプロイする場合、AWS SAM CLI は背後で `dotnet lambda package` コマンドを使用して、個々の Lambda 関数コードバンドルをパッケージ化します。

前提条件

.NET 8 SDK

.NET 8 SDK とランタイムをインストールします。

AWS SAM CLI バージョン 1.39 以降

AWS SAM CLI の最新バージョンをインストールする方法については、「AWS SAM CLI のインストール」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. 次のコマンドを使用し、Hello World .NET テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

このコマンドは、プロジェクトディレクトリに以下のファイルとディレクトリを作成します。

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
```

```
    #       ### HelloWorld.csproj
    #       ### aws-lambda-tools-defaults.json
    ### template.yaml
    ### test
        ### HelloWorld.Test
            ### FunctionTest.cs
            ### HelloWorld.Tests.csproj
```

2. `template.yaml file` が含まれるディレクトリに移動します。このファイルは、Lambda 関数や API Gateway API など、アプリケーションの AWS リソースを定義するテンプレートです。

```
cd sam-app
```

3. アプリケーションのソースを構築するには、次のコマンドを実行します。

```
sam build
```

4. AWS にアプリケーションをデプロイするには、次のコマンドを実行します。

```
sam deploy --guided
```

このコマンドは、次の一連のプロンプトを使用してアプリケーションをパッケージ化し、デプロイします。デフォルトのオプションを受け入れるには、[Enter] キーを押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか?] には、必ず y を入力してください。

- [スタック名]: AWS CloudFormation にデプロイするスタックの名前。この名前は AWS アカウントと AWS リージョンに対して一意である必要があります。
- [AWS リージョン]: アプリをデプロイしたい AWS リージョン。
- [デプロイ前に変更を確認]: [はい] を選択すると、AWS SAM がアプリケーションの変更をデプロイする前に、変更セットを手動でレビューできます。[いいえ] を選択すると、AWS SAM CLI はアプリケーションの変更を自動的にデプロイします。
- [SAM CLI の IAM ロールの作成を許可]: この例の Hello World のものを含む、多くの AWS SAM テンプレートで AWS Identity and Access Management (IAM) ロールが作成され、Lambda 関数に他の AWS のサービスへのアクセス権限が付与されます。IAM ロールを作

成または変更する AWS CloudFormation スタックをデプロイする権限を付与するには、[はい] を選択します。

- [ロールバックを無効化]: デフォルトでは、スタックの作成またはデプロイ中に AWS SAM でエラーが発生すると、スタックは以前のバージョンにロールバックされます。このデフォルトを受け入れるには [いいえ] を選択します。
- [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか?]: [y] を入力します。
- [引数を samconfig.toml に保存]: [はい] を選択して設定内容を保存します。今後は、パラメータなしで `sam deploy` を再実行して、アプリケーションに変更をデプロイできます。

5. アプリケーションのデプロイが完了すると、CLI は Hello World Lambda 関数の Amazon リソースネーム (ARN) と、その関数用に作成された IAM ロールを返します。API Gateway API のエンドポイントも表示されます。アプリケーションをテストするには、ブラウザでエンドポイントを開きます。次のようなレスポンスが表示されます。

```
{"message":"hello world","location":"34.244.135.203"}
```

6. リソースを削除するには、次のコマンドを実行します。作成した API エンドポイントは、インターネット経由でアクセス可能なパブリックエンドポイントであることに注意してください。テスト後に、このエンドポイントを削除することを推奨します。

```
sam delete
```

次のステップ

AWS SAM を使用し、.NET を使用して Lambda 関数を構築し、デプロイする方法については、以下のリソースを参照してください。

- AWS Serverless Application Model (AWS SAM) 開発者ガイド
- Building Serverless .NET Applications with AWS Lambda and the SAM CLI

AWS CDK を使用した C# Lambda 関数のデプロイ

AWS Cloud Development Kit (AWS CDK) は、最新のプログラミング言語と .NET などのフレームワークを使用して、クラウドインフラストラクチャをコードとして定義するためのオープンソースソフトウェア開発フレームワークです。AWS CDK プロジェクトを実行して AWS CloudFormation テンプレートを生成し、そのテンプレートを使用してコードをデプロイします。

AWS CDK を使用してサンプルの Hello World .NET アプリケーションを構築し、デプロイするには、以下のセクションの指示に従ってください。サンプルアプリケーションは、API Gateway エンドポイントと Lambda 関数で構成される基本的な API バックエンドを実装しています。エンドポイントに HTTP GET リクエストを送信すると、API Gateway によって Lambda 関数が呼び出されます。この関数は、リクエストを処理する Lambda インスタンスの IP アドレスと共に Hello World メッセージを返します。

前提条件

.NET 8 SDK

.NET 8 SDK とランタイムをインストールします。

AWS CDK バージョン 2

AWS CDK の最新バージョンをインストールする方法については、「AWS Cloud Development Kit (AWS CDK) v2 開発者ガイド」の「AWS CDK の開始方法」を参照してください。

AWS CDK サンプルアプリケーションをデプロイする

1. サンプルアプリケーションのプロジェクトディレクトリを作成して、そこに移動します。

```
mkdir hello-world
cd hello-world
```

2. 以下のコマンドを実行して、新しい AWS CDK アプリケーションを初期化します。

```
cdk init app --language csharp
```

このコマンドは、プロジェクトディレクトリに以下のファイルとディレクトリを作成します。

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. src ディレクトリを開き、.NET CLI を使用して新しい Lambda 関数を作成します。これは、AWS CDK を使用してデプロイする関数です。この例では、lambda.EmptyFunction テンプレートを使用して、HelloWorldLambda という名前の Hello World 関数を作成します。

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

このステップ完了後のプロジェクトディレクトリ内のディレクトリ構造は、次のようになります。

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. src/HelloWorld ディレクトリの HelloWorldStack.cs ファイルを開きます。ファイルの内容を次のコードに置き換えます。

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
```

```
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

これは、アプリケーションコードをコンパイルしてバンドルするコードであり、Lambda 関数自体の定義でもあります。この BundlingOptions オブジェクトにより、zip ファイルの内容を生成するために使用される一連のコマンドと共に zip ファイルを作成できます。この場合、`dotnet lambda package` コマンドは zip ファイルのコンパイルと生成に使用されます。

5. アプリケーションをデプロイするには、次のコマンドを実行します。

```
cdk deploy
```

6. .NET Lambda CLI を使用して、デプロイした Lambda 関数を呼び出します。

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. テストの完了後、作成したリソースは、保持することを希望しない限り削除できます。リソースを削除するには、次のコマンドを実行します。

```
cdk destroy
```

次のステップ

AWS CDK を使用し、.NET を使用して Lambda 関数を構築し、デプロイする方法については、以下のリソースを参照してください。

- C# で AWS CDK を操作
- Build, package, and publish .NET C# Lambda functions with the AWS CDK

ASP.NET アプリケーションのデプロイ

イベント駆動型関数をホストするだけでなく、.NET と Lambda を組み合わせて軽量な ASP.NET アプリケーションをホストすることもできます。`Amazon.Lambda.AspNetCoreServer` NuGet パッケージを使用して ASP.NET アプリケーションを構築し、デプロイすることができます。このセクションでは、.NET Lambda CLI ツールを使用して ASP.NET ウェブ API を Lambda にデプロイする方法について説明します。

トピック

- 前提条件
- ASP.NET ウェブ API を Lambda にデプロイする
- ASP.NET の最小限の API を Lambda にデプロイする

前提条件

.NET 8 SDK

.NET 8 SDK と ASP.NET Core ランタイムをインストールします。

Amazon.Lambda.Tools

Lambda 関数を作成するには、Amazon.Lambda.Tools .NET Global Tools 拡張機能を使用します。Amazon.Lambda.Tools をインストールするには、以下のコマンドを実行します。

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 拡張機能の詳細については、GitHub の「AWS Extensions for .NET CLI」リポジトリを参照してください。

Amazon.Lambda.Templates

Lambda 関数コードを生成するには、Amazon.Lambda.Templates NuGet パッケージを使用します。このテンプレートパッケージをインストールするには、以下のコマンドを実行します。

```
dotnet new --install Amazon.Lambda.Templates
```

ASP.NET ウェブ API を Lambda にデプロイする

ASP.NET を使用してウェブ API をデプロイするには、.NET Lambda テンプレートを使用して新しいウェブ API プロジェクトを作成します。次のコマンドを使用して、新しい ASP.NET ウェブ API プロジェクトを初期化します。このコマンド例では、プロジェクトに AspNetOnLambda という名前を付けています。

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

このコマンドは、プロジェクトディレクトリに以下のファイルとディレクトリを作成します。

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
```

```
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

Lambda が関数を呼び出すとき、使用するエントリポイントは LambdaEntryPoint.cs ファイルです。.NET Lambda テンプレートによって作成されたファイルには、次のコードが含まれています。

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Lambda が使用するエントリポイントは、Amazon.Lambda.AspNetCoreServer パッケージ内の 3 つの基本クラスのいずれかから継承する必要があります。この 3 つの基本クラスは以下のとおりです。

- APIGatewayProxyFunction
- APIGatewayHttpApiV2ProxyFunction
- ApplicationLoadBalancerFunction

提供されている .NET Lambda テンプレートを使用して LambdaEntryPoint.cs ファイルを作成するときに使用されるデフォルトクラスは APIGatewayProxyFunction です。関数で使用する基本クラスは、Lambda 関数の前にある API レイヤーによって異なります。

3 つの基本クラスにはそれぞれ、FunctionHandlerAsync という名前のパブリックメソッドが含まれています。このメソッドの名前は、Lambda が関数を呼び出すために使用するハンドラー文字列の一部になります。この FunctionHandlerAsync メソッドは、受信イベントペイロードを正しい ASP.NET 形式に変換し、ASP.NET レスポンスを Lambda レスポンスペイロードに戻します。示されているサンプル AspNetOnLambda プロジェクトでは、ハンドラー文字列は次のようになります。

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

API を Lambda にデプロイするには、次のコマンドを実行して、ソースコードファイルが含まれるディレクトリに移動し、AWS CloudFormation を使用して関数をデプロイします。

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> **dotnet lambda deploy-serverless** コマンドを使用して API をデプロイすると、AWS CloudFormation によりデプロイ時に指定したスタック名に基づき Lambda 関数に名前が付けられます。Lambda 関数にカスタム名を付けるには、serverless.template ファイルを編集して AWS::Serverless::Function リソースに FunctionName プロパティを追加します。詳細については、「AWS CloudFormation ユーザーガイド」の「Name タイプ」を参照してください。

ASP.NET の最小限の API を Lambda にデプロイする

ASP.NET の最小限の API を Lambda にデプロイするには、.NET Lambda テンプレートを使用して、新しい最小限の API プロジェクトを作成します。次のコマンドを使用して、新しい最小限の API プロジェクトを初期化します。この例では、プロジェクトに MinimalApiOnLambda という名前を付けています。

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

このコマンドは、プロジェクトディレクトリに以下のファイルとディレクトリを作成します。

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

Program.cs ファイルには次のコードが含まれています。

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

最小限の API を Lambda で実行するように設定するには、Lambda と ASP.NET Core 間のリクエストとレスポンスが正しく変換されるように、このコードを編集する必要がある場合があります。デフォルトでは、この関数は REST API イベントソース用に設定されま

す。HTTP API または Application Load Balancer の場合は、以下のオプションのいずれかに (`LambdaEventSource.RestApi`) を置き換えてください。

- (`LambdaEventSource.HttpAPi`)
- (`LambdaEventSource.ApplicationLoadBalancer`)

最小限の API を Lambda にデプロイするには、次のコマンドを実行して、ソースコードファイルが含まれるディレクトリに移動し、AWS CloudFormation を使用して関数をデプロイします。

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

コンテナイメージを使用して.NET の Lambda 関数をデプロイする

.NET Lambda 関数のコンテナイメージを構築するには 3 つの方法があります。

- .NET の AWS ベースイメージを使用する

  AWS ベースイメージには、Lambda と関数コード間のやり取りを管理するランタイムインターフェースクライアント、ローカルテスト用のランタイムインターフェイスエミュレーターがプリロードされています。

- AWS の OS 専用ベースイメージを使用する

  AWS OS 専用ベースイメージには、Amazon Linux ディストリビューションおよびランタイムインターフェイスエミュレータが含まれています。これらのイメージは、Go や Rust などのコンパイル済み言語や、Lambda がベースイメージを提供していない言語または言語バージョン (Node.js 19 など) のコンテナイメージの作成によく使用されます。OS 専用のベースイメージを使用してカスタムランタイムを実装することもできます。イメージに Lambda との互換性を持たせるには、.NET のランタイムインターフェイスクライアントをイメージに含める必要があります。

- 非 AWS ベースイメージを使用する

  Alpine Linux や Debian など、別のコンテナレジストリの代替ベースイメージを使用することもできます。組織が作成したカスタムイメージを使用することもできます。イメージに Lambda との互換性を持たせるには、.NET のランタイムインターフェイスクライアントをイメージに含める必要があります。

> **Tip**
>
> Lambda コンテナ関数がアクティブになるまでの時間を短縮するには、「Docker ドキュメント」の「マルチステージビルドを使用する」を参照してください。効率的なコンテナイメージを構築するには、「Dockerfiles を記述するためのベストプラクティス」に従ってください。

このページでは、Lambda のコンテナイメージを構築、テスト、デプロイする方法について説明します。

トピック

- .NET 用の AWS ベースイメージ

- .NET の AWS ベースイメージを使用する
- ランタイムインターフェイスクライアントで代替ベースイメージを使用する

.NET 用の AWS ベースイメージ

AWSには、.NET 用に次のようなベースイメージが利用できます。

タグ	ランタイム	オペレーティングシステム	Dockerfile	廃止
8	.NET 8	Amazon Linux 2023	GitHub 上の .NET 8 用の Dockerfile	
6	.NET 6	Amazon Linux 2	GitHub 上の .NET 6 用の Dockerfile	2024 年 11 月 12 日

Amazon ECR リポジトリ: gallery.ecr.aws/lambda/dotnet

.NET の AWS ベースイメージを使用する

前提条件

このセクションの手順を完了するには、以下が必要です。

- 「.NET SDK」 — 以下の手順では .NET 8 ベースイメージを使用します。.NET のバージョンが、Dockerfile で指定した「ベースイメージ」のバージョンと一致することを確認してください。
- Docker

ベースイメージを使用したイメージの作成およびデプロイ

次のステップでは、「Amazon.Lambda.Templates」および「Amazon.Lambda.Tools」を使用して .NET プロジェクトを作成します。次に、Docker イメージを構築し、イメージを Amazon ECR にアップロードして Lambda 関数にデプロイします。

1. Amazon.Lambda.Templates の NuGet パッケージをインストールします。

```
dotnet new install Amazon.Lambda.Templates
```

2. `lambda.image.EmptyFunction` テンプレートを使用して.NET プロジェクトを作成します。

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

3. *MyFunction*/src/*MyFunction* ディレクトリに移動します。ここにプロジェクトファイルが保存されます。次のファイルを確認します。

   - aws-lambda-tools-defaults.json - このファイルで、Lambda 関数をデプロイするときにコマンドラインオプションを指定します。
   - Function.cs - Lambda ハンドラーの関数コード。これは、デフォルトの `Amazon.Lambda.Core` ライブラリおよびデフォルトの `LambdaSerializer` 属性が含まれる C# テンプレートです。シリアル化の要件およびオプションの詳細は、Lambda 関数のシリアル化 を参照してください。提供されるコードをテストに使用することも、独自のコードで置き換えることもできます。
   - MyFunction.csproj - アプリケーションを構成するファイルおよびアセンブリを一覧表示する .NET 「プロジェクトファイル」。
   - Readme.md — このファイルには、サンプル Lambda 関数に関する詳細が含まれています。

4. `src/`*MyFunction* ディレクトリの Dockerfile を調べます。提供される Dockerfile をテストに使用することも、独自の Dockerfile で置き換えることもできます。独自のものを使用する場合、必ず次のことを行ってください。

   - FROM プロパティを「ベースイメージの URI」に設定します。.NET バージョンは、ベースイメージのバージョンと一致する必要があります。
   - CMD 引数を Lambda 関数ハンドラーに設定します。これは「`aws-lambda-tools-defaults.json`」内の `image-command` と一致している必要があります。

   Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
FROM public.ecr.aws/lambda/dotnet:8

# Copy function code to Lambda-defined environment variable
COPY publish/* ${LAMBDA_TASK_ROOT}
```

```
# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "MyFunction::MyFunction.Function::FunctionHandler" ]
```

5. Amazon.Lambda.Tools の「.NET Global Tool」をインストールします。

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools が既にインストールされている場合、最新バージョンであることを確認します。

```
dotnet tool update -g Amazon.Lambda.Tools
```

6. まだディレクトリが MyFunction ではない場合、ディレクトリを *MyFunction*/src/*MyFunction* に変更します。

```
cd src/MyFunction
```

7. Amazon.Lambda.Tools を使用して Docker イメージを構築し、新しい Amazon ECR リポジトリにプッシュして Lambda 関数をデプロイします。

`--function-role` では、Amazon リソースネーム (ARN) ではなく、関数の「実行ロール」のロール名を指定します。例えば、`lambda-role` と指定します。

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

Amazon.Lambda.Tools の .NET Global Tool の詳細については、GitHub の AWS Extensions for .NET CLI リポジトリを参照してください。

8. 関数を呼び出します。

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

すべてが成功すると、次のメッセージが表示されます。

```
Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
```

```
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

9. Lambda 関数を削除する

```
dotnet lambda delete-function MyFunction
```

ランタイムインターフェイスクライアントで代替ベースイメージを使用する

OS 専用ベースイメージまたは代替のベースイメージを使用する場合、イメージにランタイムインターフェイスクライアントを含める必要があります。ランタイムインターフェイスクライアントは、Lambda と関数コード間の相互作用を管理する Lambda Runtime API を拡張します。

次の例は、非 AWS ベースイメージを使用して .NET 用のコンテナイメージを構築する方法と、.NET 用の Lambda ランタイムインターフェースクライアントである Amazon.Lambda.RuntimeSupport パッケージを追加する方法を示しています。Dockerfile の例では、Microsoft .NET 8 ベースイメージを使用します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- 「.NET SDK」 — 以下の手順では .NET 8 ベースイメージを使用します。.NET のバージョンが、Dockerfile で指定した「ベースイメージ」のバージョンと一致することを確認してください。
- Docker

代替的なベースイメージを使用したイメージの作成およびデプロイ

1. Amazon.Lambda.Templates の NuGet パッケージをインストールします。

```
dotnet new install Amazon.Lambda.Templates
```

2. `lambda.CustomRuntimeFunction` テンプレートを使用して.NET プロジェクトを作成します。このテンプレートには Amazon.Lambda.RuntimeSupport パッケージが含まれています。

```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
```

3. *MyFunction*/src/*MyFunction* ディレクトリに移動します。ここにプロジェクトファイルが保存されます。次のファイルを確認します。

   - aws-lambda-tools-defaults.json - このファイルで、Lambda 関数をデプロイするときにコマンドラインオプションを指定します。
   - Function.cs — コードには、Amazon.Lambda.RuntimeSupport ライブラリをブートストラップとして初期化する Main メソッドを持つクラスが含まれています。Main メソッドは、関数のプロセスのエントリポイントです。この Main メソッドは、ブートストラップが処理できるラッパーで関数ハンドラをラップします。詳細については、GitHub リポジトリの「Amazon.Lambda.RuntimeSupport をクラスライブラリとして使用する」を参照してください。
   - MyFunction.csproj - アプリケーションを構成するファイルおよびアセンブリを一覧表示する .NET 「プロジェクトファイル」。
   - Readme.md — このファイルには、サンプル Lambda 関数に関する詳細が含まれています。

4. aws-lambda-tools-defaults.json ファイルを開き、次の行を追加します。

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

   - package-type: デプロイパッケージをコンテナイメージとして定義します。
   - docker-host-build-output-dir: ビルドプロセスの出力ディレクトリを設定します。

   Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
```

```
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. *MyFunction*/src/*MyFunction* ディレクトリに Dockerfile を作成します。次の Dockerfile の例では、AWS ベースイメージの代わりに Microsoft .NET ベースイメージを使用します。

   - ベースイメージ識別子に FROM プロパティを設定します。.NET バージョンは、ベースイメージのバージョンと一致する必要があります。
   - COPY コマンドを使用して、関数を /var/task ディレクトリにコピーします。
   - ENTRYPOINT を、Docker コンテナの起動時に実行させるモジュールに設定します。この場合、モジュールは Amazon.Lambda.RuntimeSupport ライブラリを初期化するブートストラップです。

   Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
FROM mcr.microsoft.com/dotnet/runtime:8.0

# Set the image's internal work directory
WORKDIR /var/task

# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net8.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. Amazon.Lambda.Tools の「.NET Global Tool」拡張機能をインストールします。

```
dotnet tool install -g Amazon.Lambda.Tools
```

   Amazon.Lambda.Tools が既にインストールされている場合、最新バージョンであることを確認します。

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Amazon.Lambda.Tools を使用して Docker イメージを構築し、新しい Amazon ECR リポジトリにプッシュして Lambda 関数をデプロイします。

   `--function-role` では、Amazon リソースネーム（ARN）ではなく、関数の「実行ロール」のロール名を指定します。例えば、`lambda-role` と指定します。

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   Amazon.Lambda.Tools の .NET CLI 拡張機能の詳細については、GitHub の AWS Extensions for .NET CLI リポジトリを参照してください。
8. 関数を呼び出します。

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   すべてが成功すると、次のメッセージが表示されます。

   ```
   Payload:
   "TESTING THE FUNCTION"

   Log Tail:
   START RequestId: id Version: $LATEST
   END RequestId: id
   REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
    Size: 256 MB     Max Memory Used: 12 MB
   ```
9. Lambda 関数を削除する

   ```
   dotnet lambda delete-function MyFunction
   ```

.NET Lambda 関数コードをネイティブランタイム形式にコンパイルする

.NET 8 はネイティブ ahead-of-time (AOT) コンパイルをサポートします。ネイティブ AOT を使用することで、Lambda 関数コードをネイティブなランタイム形式にコンパイルできるので、実行時に .NET コードをコンパイルする必要がなくなります。ネイティブ AOT コンパイルは、.NET で記述する Lambda 関数のコールドスタート時間を短縮できます。詳細については、AWS コンピューティングブログの「Introducing the .NET 8 runtime for AWS Lambda」を参照してください。

セクション

Lambda ランタイム

ネイティブ AOT コンパイルで構築された Lambda 関数をデプロイするには、 マネージド .NET 8 Lambda ランタイムを使用します。このランタイムは、x86_64 アーキテクチャと arm64 アーキテクチャの両方の使用をサポートします。

AOT を使用せずに .NET Lambda 関数をデプロイすると、アプリケーションが中間言語 (IL) コードにコンパイルされます。実行時、Lambda ランタイム just-in-time (JIT) コンパイラが、必要に応じて IL コードを取得してマシンコードにコンパイルします。ネイティブ AOT で事前にコンパイルされた Lambda 関数では、関数をデプロイするときにコードをマシンコードにコンパイルするため、Lambda ランタイムの .NET ランタイムや SDK に依存しずに、実行前にコードをコンパイルできます。

AOT の制限の 1 つは、アプリケーションコードを、.NET 8 ランタイムが使用するのと同じ Amazon Linux 2023 (AL2023) オペレーティングシステムを使用する環境でコンパイルする必要があることです。.NET Lambda CLI には、AL2023 イメージを使用して Docker コンテナでアプリケーションをコンパイルする機能があります。

アーキテクチャ間の互換性に関する潜在的な問題を避けるため、関数用に設定したのと同じプロセッサアーキテクチャの環境でコードをコンパイルすることを強くお勧めします。クロスアーキテクチャコンパイルの制限の詳細については、Microsoft .NET ドキュメントの「クロスコンパイル」を参照してください。

前提条件

Docker

ネイティブ AOT を使用するには、関数コードを、 ランタイムと同じ AL2023 オペレーティングシステムの環境でコンパイルする必要があります。以下のセクションで説明する .NET CLI コマンドは、Docker を使用して AL2023 環境で Lambda 関数を開発および構築します。

.NET 8 SDK

ネイティブ AOT コンパイルは .NET 8 の機能です。ビルドマシンには、ランタイムだけでなく、.NET 8 SDK もインストールする必要があります。

Amazon.Lambda.Tools

Lambda 関数を作成するには、Amazon.Lambda.Tools .NET Global Tools 拡張機能を使用します。Amazon.Lambda.Tools をインストールするには、以下のコマンドを実行します。

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 拡張機能の詳細については、GitHub の「AWS Extensions for .NET CLI」リポジトリを参照してください。

Amazon.Lambda.Templates

Lambda 関数コードを生成するには、Amazon.Lambda.Templates NuGet パッケージを使用します。このテンプレートパッケージをインストールするには、以下のコマンドを実行します。

```
dotnet new install Amazon.Lambda.Templates
```

開始

.NET Global CLI と AWS Serverless Application Model (AWS SAM) はどちらも、ネイティブ AOT を使用してアプリケーションを構築するための開始用テンプレートを提供します。最初のネイティブ AOT Lambda 関数を構築するには、以下の手順を実行します。

ネイティブ AOT コンパイル済み Lambda 関数を初期化してデプロイするには

1. ネイティブ AOT テンプレートを使用して新しいプロジェクトを初期化し、作成した .cs ファイルと .csproj ファイルが含まれるディレクトリに移動します。この例では、関数に NativeAotSample という名前を付けています。

```
dotnet new lambda.NativeAOT -n NativeAotSample
cd ./NativeAotSample/src/NativeAotSample
```

ネイティブ AOT テンプレートによって作成された Function.cs ファイルには、次の関数コードが含まれています。

```
using Amazon.Lambda.Core;
using Amazon.Lambda.RuntimeSupport;
using Amazon.Lambda.Serialization.SystemTextJson;
using System.Text.Json.Serialization;

namespace NativeAotSample;

public class Function
{
    /// <summary>
    /// The main entry point for the Lambda function. The main function is called
 once during the Lambda init phase. It
    /// initializes the .NET Lambda runtime client passing in the function handler
 to invoke for each Lambda event and
    /// the JSON serializer to use for converting Lambda JSON format to the .NET
 types.
    /// </summary>
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new
 SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
            .Build()
            .RunAsync();
    }

    /// <summary>
    /// A simple function that takes a string and does a ToUpper.
    ///
```

```
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
```

```
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

ネイティブ AOT は、アプリケーションを単一のネイティブバイナリにコンパイルします。このバイナリのエントリポイントは `static Main` メソッドです。`static Main` 内部では、Lambda ランタイムがブートストラップされ、`FunctionHandler` メソッドが設定されます。ランタイムブートストラップの一部として、ソース生成のシリアライザーは `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` を使用して設定されます。

2. アプリケーションを Lambda にデプロイするには、Docker がローカル環境で実行されていることを確認し、次のコマンドを実行します。

```
dotnet lambda deploy-function
```

バックグラウンドでは、.NET Global CLI が AL2023 Docker イメージをダウンロードし、実行中のコンテナ内でアプリケーションコードをコンパイルしています。コンパイルされたバイナリは、Lambda にデプロイされる前にローカルファイルシステムに出力されます。

3. 次のコマンドを実行して関数をテストします。`<FUNCTION_NAME>` を、デプロイウィザードで関数に選択した名前に置き換えます。

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

CLI からの応答には、コールドスタートのパフォーマンスの詳細 (初期化時間) と関数呼び出しの合計実行時間が含まれます。

4. 前の手順で作成した AWS リソースを削除するには、次のコマンドを実行します。`<FUNCTION_NAME>` を、デプロイウィザードで関数に選択した名前に置き換えます。使用しなくなった AWS リソースを削除することで、AWS アカウントに請求される料金の発生を防ぎます。

```
dotnet lambda delete-function <FUNCTION_NAME>
```

シリアル化

ネイティブ AOT を使用して Lambda に関数をデプロイするには、関数コードでソース生成のシリアル化を使用する必要があります。ランタイムリフレクションを使用し、シリアル化のためにオブジェクトプロパティにアクセスするのに必要なメタデータを収集する代わりに、ソースジェネレーターは、アプリケーションの構築時にコンパイルされる C# ソースファイルを生成します。ソース生成のシリアライザーを正しく設定するには、関数が使用する入力オブジェクトと出力オブジェクト、およびカスタムタイプが含まれていることを確認します。例えば、API Gateway REST API からイベントを受信してカスタム `Product` タイプを返す Lambda 関数には、次のように定義されたシリアライザーが含まれます。

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

トリミング

ネイティブ AOT は、バイナリができるだけ小さくなるように、コンパイルの一部としてアプリケーションコードをトリミングします。.NET 8 for Lambda は、以前のバージョンの .NET と比較して、トリミングサポートが改善されています。Lambda ランタイムライブラリ、AWS .NET SDK 、.NET Lambda アノテーション 、.NET 8 自体にサポートが追加されました。

これらの改善により、ビルド時のトリミング警告を排除できる可能性がありますが、.NET が完全にトリムセーフになることはありません。つまり、関数が依存するライブラリの一部が、コンパイルステップの一部としてトリミングされる可能性があるということです。これを管理するには、次の例に示すように、`TrimmerRootAssemblies` を `.csproj` ファイルの一部として定義することでこれを管理できます。

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

トリミング警告が表示された場合は、警告を生成するクラスを `TrimmerRootAssembly` に追加しても、問題は解決しない可能性があります。トリミング警告は、ランタイムまで決定できない他のクラスにクラスがアクセスしようとしていることを示します。ランタイムエラーを回避するには、この 2 番目のクラスを `TrimmerRootAssembly` に追加します。

トリミング警告の管理の詳細については、Microsoft .NET ドキュメントの「Introduction to trim warnings」を参照してください。

トラブルシューティング

Error: Cross-OS native compilation is not supported.(エラー: クロス OS のネイティブコンパイルはサポートされません。)

> 使用している Amazon.Lambda.Tools .NET Core グローバルツールのバージョンが古くなっています。最新のバージョンにアップデートしてから、再試行してください。

Docker: image operating system "linux" cannot be used on this platform. (Docker: このプラットフォームでイメージオペレーティングシステム「linux」を使用することはできません。)

> システム上の Docker は、Windows コンテナを使用するように設定されています。Linux コンテナに切り替えて、ネイティブ AOT 構築環境を実行してください。

一般的なエラーの詳細については、GitHub の AWS NativeAOT for .NET リポジトリを参照してください。

C# の AWS Lambda context オブジェクト

Lambda で関数が実行されると、コンテキストオブジェクトがハンドラーに渡されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を含むプロパティです。

context プロパティ

- `FunctionName` - Lambda 関数の名前。
- `FunctionVersion` - 関数のバージョン。
- `InvokedFunctionArn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `MemoryLimitInMB` - 関数に割り当てられたメモリの量。
- `AwsRequestId` - 呼び出しリクエストの ID。
- `LogGroupName` - 関数のロググループ。
- `LogStreamName` - 関数インスタンスのログストリーム。
- `RemainingTime` (`TimeSpan`) - 実行がタイムアウトするまでの残りのミリ秒数。
- `Identity` - (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
- `ClientContext` - (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。
- `Logger` 関数のロガーオブジェクト。

モニタリング目的として、`ILambdaContext` オブジェクトの情報を使用して関数の呼び出しに関する情報を出力できます。次のコードは、構造化ログ記録フレームワークにコンテキスト情報を追加する方法の例を示しています。この例では、関数はログ出力に `AwsRequestId` を追加します。また、この関数は `RemainingTime` プロパティを使用して、Lambda 関数のタイムアウトに達しそうな場合にインフライトタスクをキャンセルします。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        finally
        {
            cts.Cancel();

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
    }
}
```

C# での Lambda 関数のログ記録

AWS Lambda は、Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログエントリを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda のランタイム環境は、各呼び出しの詳細や、関数のコードからのその他の出力をログストリームに送信します。CloudWatch Logs の詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

セクション

ログを返す関数の作成

関数コードからログを出力するには、コンソールクラスのメソッドか、`stdout` または `stderr` に書き込む任意のログ記録のライブラリを使用します。

.NET ランタイムは、呼び出しごとに `START`、`END`、`REPORT` の各行を記録します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。
- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。

- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- サンプリング済み - トレースされたリクエストの場合、サンプリング結果。

ツールとライブラリ

Powertools for AWS Lambda (.NET) は、サーバーレスのベストプラクティスを実装し、デベロッパーの作業速度を向上させるためのデベロッパーツールキットです。ログ記録ユーティリティには、Lambda に最適化されたロガーを提供し、すべての関数を通して関数コンテキストに関する追加情報が含まれ、JSON 形式で構成した出力を行います。このユーティリティを使用して次のことができます。

- Lambda の コンテキスト、コールドスタート、構造から主要キーをキャプチャし、JSON 形式でログ出力する
- 指示された場合 Lambda 呼び出しイベントをログ記録する (デフォルトでは無効)
- ログサンプリングにより、特定の割合の呼び出しにのみすべてのログを出力する (デフォルトでは無効)
- 任意のタイミングで、構造化されたログにキーを追加する
- カスタムログフォーマッター (Bring Your Own Formatter) を使用して、組織の ログ記録 RFC と互換性のある構造でログを出力する

構造化されたログ記録に Powertools for AWS Lambda (.NET) と AWS SAM を使用する

以下の手順に従い、AWS SAM を使用する統合済み Powertools for AWS Lambda (.NET) モジュールを使用して、Hello World C# アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は `hello world` のメッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- .NET 6 または .NET 8

- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World TypeScript テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のログを取得するには、sam logs を実行します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「ログの使用」を参照してください。

```
sam logs --stack-name sam-app
```

ログ出力は次のようになります。

```
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2023-02-20T14:15:27.988000 INIT_START Runtime Version:
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:29.259000
 2023-02-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.479000
 2023-02-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2023-02-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.599000
 2023-02-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2023/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2023-02-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2451 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

ログ保持の管理

関数を削除しても、ロググループは自動的には削除されません。ログを無期限に保存しないようにするには、ロググループを削除するか、CloudWatch がログを自動的に削除するまでの保持期間を設定します。ログ保持を設定するには、AWS SAM テンプレートに以下を追加します。

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Lambda コンソールを使用する

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。

2. 機能のロググループを選択します(/aws/lambda/###)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。 X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
```

```
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

AWS Lambda での C# コードの作成

Lambda アプリケーションのトレース、デバッグ、最適化を行うために、Lambda は AWS X-Ray と統合されています。X-Ray を使用すると、Lambda 関数や他の AWS のサービスが含まれるアプリケーション内で、リソースを横断するリクエストをトレースできます。

トレーシングデータを X-Ray に送信するには、以下に表示された 3 つの SDK ライブラリのいずれかを使用できます。

- AWS Distro for OpenTelemetry (ADOT) – 安全で本番環境に対応し、AWS でサポートされている OpenTelemetry (OTel) SDK のディストリビューションです。
- AWS X-Ray SDK for .NET - トレースデータを生成して X-Ray に送信するための SDK です。
- Powertools for AWS Lambda (.NET) — サーバーレスのベストプラクティスを実装し、開発者の作業速度を向上させるための開発者ツールキットです。

各 SDK は、テレメトリデータを X-Ray サービスに送信する方法を提供します。続いて、X-Ray を使用してアプリケーションのパフォーマンスメトリクスの表示やフィルタリングを行い、インサイトを取得することで、問題点や最適化の機会を特定できます。

> **Important**
>
> X-Ray および Powertools for AWS Lambda SDK は、AWS が提供する、密接に統合された計測ソリューションの一部です。ADOT Lambda レイヤーは、一般的により多くのデータを収集するトレーシング計測の業界標準の一部ですが、すべてのユースケースに適しているわけではありません。これらのソリューションのいずれかを使用して、X-Ray でエンドツーエンドのトレーシングを実装することができます。選択方法の詳細については、「Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs」(Distro for Open Telemetry または X-Ray SDK の選択) を参照してください。

セクション

- AWS CloudFormation によるトレースのアクティブ化
- X-Ray トレースの解釈

トレーシングに Powertools for AWS Lambda (.NET) と AWS SAM を使用する

以下の手順に従い、AWS SAM を使用する統合済み Powertools for AWS Lambda (.NET) モジュールを使用して、Hello World C# アプリケーションのサンプルをダウンロード、構築、デプロイします。このアプリケーションは基本的な API バックエンドを実装し、Powertools を使用してログ、メトリクス、トレースを生成します。Amazon API Gateway エンドポイントと Lambda 関数で構成されています。API Gateway エンドポイントに GET リクエストを送信すると、Lambda 関数は呼び出し、Embedded Metric Format を使用してログおよびメトリクスを CloudWatch に送信、トレースを AWS X-Ray に送信します。関数は「hello world」メッセージを返します。

前提条件

このセクションの手順を完了するには、以下が必要です。

- .NET 6 または .NET 8
- AWS CLI バージョン 2
- 「AWS SAM CLI バージョン 1.75 以降」 AWS SAM CLI のバージョンが古い場合は、「AWS SAM CLI のアップグレード」を参照してください。

AWS SAM サンプルアプリケーションをデプロイする

1. Hello World TypeScript テンプレートを使用して、アプリケーションを初期化します。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. アプリケーションを構築します。

```
cd sam-app && sam build
```

3. アプリケーションをデプロイします。

```
sam deploy --guided
```

4. 画面に表示されるプロンプトに従ってください。インタラクティブな形式で提供されるデフォルトオプションを受け入れるには、Enter を押します。

> **Note**
>
> [HelloWorldFunction には権限が定義されていない場合がありますが、問題ありませんか？] には、必ず y を入力してください。

5. デプロイされたアプリケーションの URL を取得します。

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API エンドポイントを呼び出します。

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功すると、次のレスポンスが表示されます。

```
{"message":"hello world"}
```

7. 関数のトレースを取得するには、sam traces を実行します。

```
sam traces
```

トレース出力は次のようになります。

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
```

```
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. これは、インターネット経由でアクセス可能なパブリック API エンドポイントです。テスト後にエンドポイントを削除することを推奨します。

```
sam delete
```

X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray SDK を使用して .NET 関数を計装する

関数コードを実装して、メタデータを記録し、ダウンストリームコールをトレースできます。関数が他のリソースやサービスに対して行うコールの詳細を記録するには、AWS X-Ray SDK for .NET を使用します。SDK を取得するには、`AWSXRayRecorder` パッケージをプロジェクトファイルに追加します。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

AWS SDK、Entity Framework、HTTP リクエストに自動計測を提供するさまざまな Nuget パッケージがあります。設定オプションの完全なセットについては、「AWS X-Ray 開発者ガイド」の「AWS X-Ray SDK for .NET」を参照してください。

目的の Nuget パッケージを追加したら、自動計測を設定します。ベストプラクティスは、この設定を関数のハンドラー関数の外部で実行することです。これにより、実行環境の再利用を活用して関数のパフォーマンスを向上させることができます。次のコード例では、`RegisterXRayForAllServices` メソッドを関数コンストラクタで呼び出して、すべての AWS SDK 呼び出しに計測を追加しています。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Lambda コンソールを使用してトレースを有効化する

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。

4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [保存] をクリックします。

Lambda API でのトレースのアクティブ化

AWS CLI または AWS SDK で Lambda 関数のトレースを設定するには、次の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation によるトレースのアクティブ化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray トレースの解釈

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

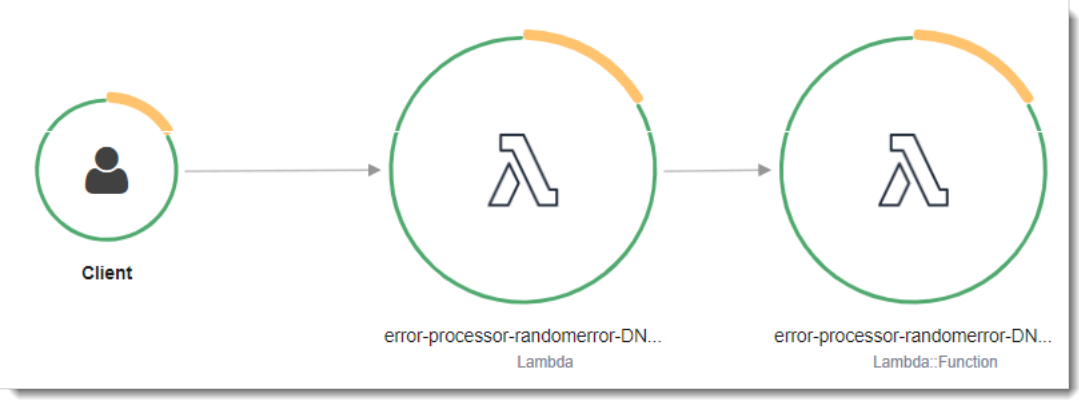
アクティブトレースの設定後は、アプリケーションを通じて特定のリクエストの観測が行えるようになります。[X-Ray サービスグラフ] には、アプリケーションとそのすべてのコンポーネントに関する情報が表示されます。次の図は、2 つの関数を持つアプリケーションを示しています。プライマリ関数はイベントを処理し、エラーを返す場合があります。上位 2 番目の関数は、最初のロググループに表示されるエラーを処理し、AWS SDKを使用してX-Ray、Amazon Simple Storage Service (Amazon S3)、および Amazon CloudWatch Logs を呼び出します。

X-Ray は、アプリケーションへのすべてのリクエストをトレースするわけではありません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエ

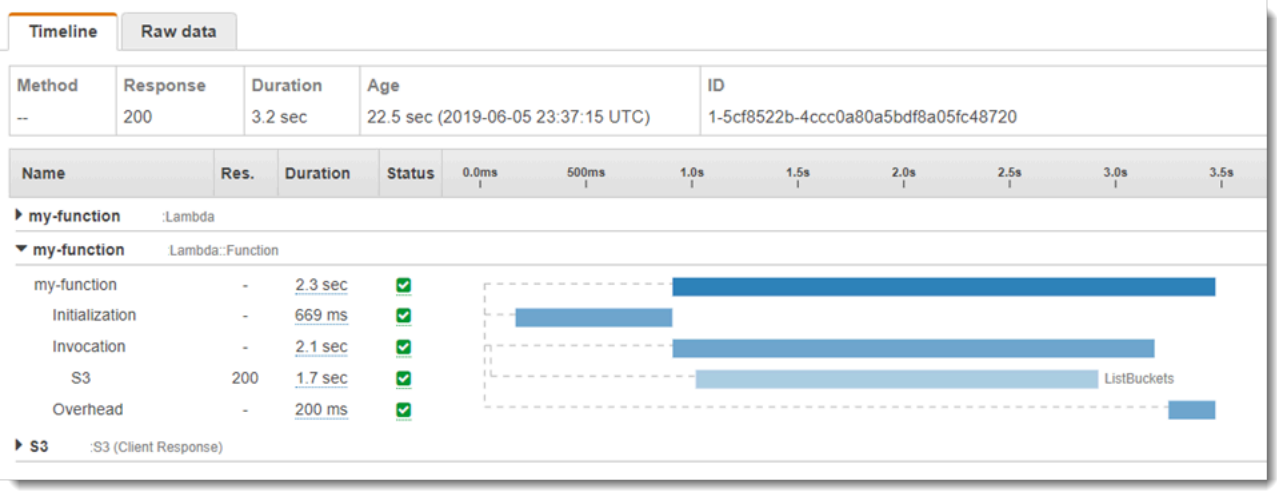
ストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは `AWS::Lambda` の起点があり、もう 1 つは `AWS::Lambda::Function` の起点があります。`AWS::Lambda` セグメントにエラーが表示される場合は、Lambda サービスに問題があります。`AWS::Lambda::Function` セグメントにエラーが表示される場合、関数に問題があります。



この例では、AWS::Lambda::Function セグメントを展開して、それの 3 つのサブセグメントが表示されています。

- 初期化 - 関数のロードと初期化コードの実行に要した時間を表します。このサブセグメントは、関数の各インスタンスが処理する最初のイベントに対してのみ表示されます。
- [呼び出し] - ハンドラーコードの実行に要した時間を表します。
- [オーバーヘッド] - Lambda ランタイムが次のイベントを処理するための準備に要する時間を表します。

HTTP クライアントをインストルメント化し、SQL クエリを記録して、注釈とメタデータからカスタムサブセグメントを作成することもできます。詳細については、「AWS X-Ray デベロッパーガイド」の「AWS X-Ray SDK for .NET」を参照してください。

> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

C# での AWS Lambda 関数テスト

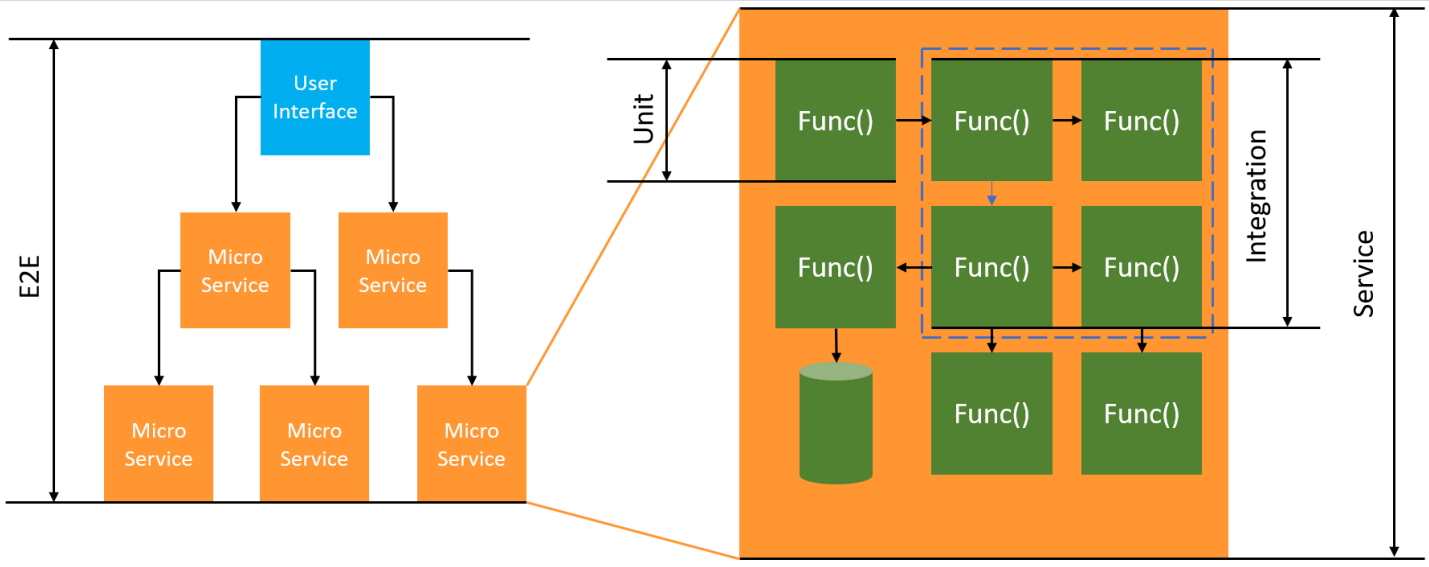
> **Note**
>
> サーバーレスソリューションをテストするための手法とベストプラクティスの詳細については、「関数のテスト」の章を参照してください。

サーバーレス関数のテストでは、従来のテストタイプと手法を使用しますが、サーバーレスアプリケーション全体のテストも検討する必要があります。クラウドベースのテストでは、関数とサーバーレスアプリケーションの両方の品質を最も正確に測定できます。

サーバーレスアプリケーションアーキテクチャには、API 呼び出しを通じて重要なアプリケーション機能を提供するマネージドサービスが含まれます。このため、開発サイクルには、関数とサービスが相互に作用する際に機能を検証する自動テストを含める必要があります。

クラウドベースのテストを作成しない場合、ローカル環境とデプロイされた環境の違いにより問題が発生する可能性があります。継続的な統合プロセスでは、コードをQA、ステージング、本番稼働などの次のデプロイ環境に昇格する前に、クラウドにプロビジョニングされた一連のリソースに対してテストを実行する必要があります。

サーバーレスアプリケーションのテスト戦略に関する詳細については、このショートガイドを引き続きご覧ください。また、サーバーレステストサンプルリポジトリにアクセスして、選択した言語とランタイムに固有の実用的な例を調べることもできます。



サーバーレステストでも、ユニット 、 統合、end-to-endテストを記述します。

- ユニットテスト - 分離されたコードブロックに対して実行されるテスト。例えば、特定の商品と配送先を指定して配送料を計算するビジネスロジックを検証する場合です。
- 統合テスト - 通常はクラウド環境で相互作用する 2 つ以上のコンポーネントまたはサービスを対象としたテスト。例えば、キューからのイベントを処理する関数を検証する場合です。
- E nd-to-end テスト - アプリケーション全体の動作を検証するテスト。例えば、インフラストラクチャが正しくセットアップされ、顧客の注文を記録するためにイベントがサービス間で想定どおりに流れることを確認する場合です。

サーバーレスアプリケーションのテスト

通常、サーバーレスアプリケーションコードのテストには、クラウドでのテスト、モックを使ったテスト、場合によってはエミュレーターでのテストなど、さまざまな方法を組み合わせます。

クラウドでのテスト

クラウドでのテストは、ユニットテスト、統合テスト、テストなど、 end-to-end テストのすべてのフェーズで重要です。クラウドにデプロイされたコードやクラウドベースのサービスとやり取りするコードに対してテストを実行します。この方法では、コードの品質を最も正確に測定できます。

クラウドで Lambda 関数をデバッグする便利な方法は、コンソールからテストイベントを行うことです。テストイベントとは、関数への JSON 入力のことです。関数が入力を必要としない場合、イベントは空の JSON ドキュメント ({}) にすることができます。コンソールには、さまざまなサービス統合のサンプルイベントが用意されています。コンソールでイベントを作成したら、それをチームと共有して、テストを簡単かつ一貫性のあるものにすることができます。

> **Note**
>
> コンソールで関数をテストするのが簡単な方法ですが、テストサイクルを自動化することでアプリケーションの品質と開発スピードが保証されます。

テストツール

開発サイクルを能率化するために、関数のテスト時に使用できるツールやテクニックは数多くあります。例えば、AWS SAM Accelerate と AWS CDK 監視モードは、いずれもクラウド環境の更新に要する時間を短縮します。

Lambda 関数コードを定義する方法により、ユニットテストを簡単に追加できます。Lambda では、クラスを初期化するためのパラメータなしのパブリックコンストラクタが必要です。2 つ目の内部コンストラクタを導入すると、アプリケーションが使用する依存関係を制御できるようになります。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

この関数のテストを作成するには、Function クラスの新しいインスタンスを初期化し、IDatabaseRepository のモック実装を渡します。以下の例では、XUnit、Moq、FluentAssertions を使用して、FunctionHandler がステータスコード 200 を返すことを確認する簡単なテストを記述しています。

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}
```

非同期テストの例など、より詳細な例については、「」の「.NET テストサンプルリポジトリ」を参照してください GitHub。

PowerShell による Lambda 関数の構築

以下のセクションでは、Lambda 関数のコードを PowerShell で記述する際に、一般的なプログラミングパターンと主要概念がどのように適用されるかについて説明します。

Lambda は、PowerShell 用の次のサンプルアプリケーションを提供します。

- blank-powershell – ログ記録、環境変数、AWS SDK の使用方法を示す PowerShell 関数。

開始する前に、PowerShell の開発環境をまず設定する必要があります。これを行う手順については、「PowerShell 開発環境のセットアップ」を参照してください。

AWSLambdaPSCore モジュールを使用してテンプレートからサンプルの PowerShell プロジェクトをダウンロードし、PowerShell デプロイパッケージを作成して AWS クラウドに PowerShell 関数をデプロイする方法については、「.zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする」を参照してください。

Lambda は、次の .NET 言語のランタイムをサポートしています。

.NET

名前	識別子	オペレーティングシステム	廃止日	関数の作成をブロックする	関数の更新をブロックする
.NET 8	`dotnet8`	Amazon Linux 2023			
.NET 6	`dotnet6`	Amazon Linux 2	2024 年 11 月 12 日	2025 年 2 月 28 日	2025 年 3 月 31 日

トピック

- PowerShell 開発環境のセットアップ
- .zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする
- PowerShell の Lambda 関数ハンドラーの定義
- AWS Lambda の context オブジェクト PowerShell
- PowerShell の AWS Lambda 関数ログ作成

PowerShell 開発環境のセットアップ

Lambda は、 PowerShell ランタイム用のツールとライブラリのセットを提供します。インストール手順については、「」の「 用の Lambda ツール PowerShell」を参照してください GitHub。

AWSLambdaPSCore モジュールには、 PowerShell Lambda 関数の作成とパブリッシュに役立つ次のコマンドレットが含まれています。

- Get-AWSPowerShellLambdaTemplate — 開始用テンプレートのリストを返します。
- New-AWSPowerShellLambda - テンプレートに基づいて初期 PowerShell スクリプトを作成します。
- Publish-AWSPowerShellLambda - 指定された PowerShell スクリプトを Lambda に発行します。
- 新規 —AWSPowerShellLambdaPackage デプロイ用の CI/CD システムで使用できる Lambda デプロイパッケージを作成します。

.zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする

PowerShell ランタイムのデプロイパッケージには、 PowerShell スクリプト、 PowerShell スクリプトに必要な PowerShell モジュール、 PowerShell Core をホストするために必要なアセンブリが含まれています。

Lambda 関数の作成

Lambda で PowerShell スクリプトの記述と呼び出しを開始するには、 `New-AWSPowerShellLambda` コマンドレットを使用して、テンプレートに基づいてスタータースクリプトを作成します。Lambda にスクリプトをデプロイするには、`Publish-AWSPowerShellLambda` コマンドレットを使用します。その後、コマンドラインあるいは Lambda コンソールから、スクリプトをテストできます。

新しい PowerShell スクリプトを作成してアップロードし、テストするには、次の手順を実行します。

1. 使用可能なテンプレートの一覧を表示するには、次のコマンドを実行します。

   ```
   PS C:\> Get-AWSPowerShellLambdaTemplate

   Template                Description
   --------                -----------
   Basic                   Bare bones script
   CodeCommitTrigger       Script to process AWS CodeCommit Triggers
   ...
   ```

2. `Basic` テンプレートに基づいてサンプルスクリプトを作成するには、次のコマンドを実行します。

   ```
   New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
   ```

   `MyFirstPSScript.ps1` という名前の新しいファイルが現在のディレクトリの新しいサブディレクイトリに作成されます。ディレクトリの名前は、`-ScriptName` パラメータに基づきます。別のディレクトリを選択するには、 `-Directory` パラメータを使用できます。

   新しいファイルの内容は次のとおりに表示されます。

```
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. PowerShell スクリプトからのログメッセージが Amazon CloudWatch Logs にどのように送信されるかを確認するには、サンプルスクリプトの `Write-Host`行のコメントを解除します。

   Lambda 関数からデータを返す方法を示すには、スクリプトの末尾に新しい行 (`$PSVersionTable`) を追加します。これにより、 が PowerShell パイプライン`$PSVersionTable`に追加されます。 PowerShell スクリプトが完了すると、 PowerShell パイプラインの最後のオブジェクトは Lambda 関数の戻りデータです。 `$PSVersionTable` は、実行中の環境に関する情報も提供する PowerShell グローバル変数です。

   上述の変更を行った後のサンプルスクリプトの最後の 2 行は次のようになります。

```
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
$PSVersionTable
```

4. `MyFirstPSScript.ps1` ファイルを編集したら、スクリプトの場所にディレクトリを変更します。次に、次のコマンドを実行して、Lambda にスクリプトを発行します。

```
Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
 MyFirstPSScript -Region us-east-2
```

-Name パラメータは Lambda 関数名を指定しますが、これは Lambda コンソールに表示されます。この関数を使用して、手動でスクリプトを呼び出すことができます。

5. AWS Command Line Interface (AWS CLI) invoke コマンドを使用して関数を呼び出します。

```
> aws lambda invoke --function-name MyFirstPSScript out
```

PowerShell の Lambda 関数ハンドラーの定義

Lambda 関数が呼び出されると、Lambda ハンドラは PowerShell スクリプトを呼び出します。

PowerShell スクリプトが呼び出された場合、以下の変数は事前定義されます。

- *$LambdaInput* - ハンドラへの入力を含む PSObject。この入力は、イベントデータ (イベントソースによって公開される)、あるいは文字列やカスタムデータオブジェクトなど、ユーザーが提供するカスタム入力とすることができます。
- *$LambdaContext* - 現在の呼び出しに関する情報にアクセスするために使用できる、Amazon.Lambda.Core.ILambdaContext オブジェクト。アクセスできる情報には、現在の関数名、メモリ制限、残りの実行時間、ログ記録があります。

例えば、次の PowerShell コードの例を考えてみます。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

このスクリプトは、$LambdaContext 変数から取得した FunctionName プロパティを返します。

> **Note**
>
> PowerShell スクリプト内の `#Requires` ステートメントを使用して、スクリプトが依存するモジュールを指定するように求められます。このステートメントは 2 つの重要なタスクを実行します。1) スクリプトが使用するモジュールを他のデベロッパーに伝え、そして 2) デプロイの一部として、スクリプトでパッケージングするために AWS PowerShell ツールが必要とする依存モジュールを識別します。PowerShell の `#Requires` ステートメントに関する詳細については、「要件について」を参照してください。PowerShell デプロイパッケージについての詳細は、.zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイするを参照してください。
>
> PowerShell Lambda 関数が AWS PowerShell コマンドレットを使用する場合、`#Requires` ステートメントが、Windows PowerShell のみをサポートする `AWSPowerShell` モジュールではなく、PowerShell Core をサポートする `AWSPowerShell.NetCore` モジュールを参照するように設定されていることを確認します。また、コマンドレットインポートプロセスを最適化する `AWSPowerShell.NetCore` のバージョン 3.3.270.0 以降を使用していることも確認します。古いバージョンを使用している場合、より長いコールドスタートが発生します。詳細については、「AWS Tools for PowerShell」を参照してください。

データを返す

一部の Lambda 呼び出しには、呼び出し元にデータを返す目的があります。例えば、呼び出しが API Gateway から送信されたウェブリクエストへの返答である場合、Lambda 関数はレスポンスを返す必要があります。PowerShell Lambda の場合、PowerShell パイプラインに最後に追加されるオブジェクトは Lambda 呼び出しから返されたデータです。オブジェクトが文字列の場合、返されるデータも文字列です。それ以外の場合、`ConvertTo-Json` コマンドレットを使用して、オブジェクトは JSON に変換されます。

例えば、PowerShell パイプラインに `$PSVersionTable` を追加する次の PowerShell ステートメントを検討します。

```
$PSVersionTable
```

PowerShell スクリプトの終了後の PowerShell パイプラインの最後のオブジェクトは、Lambda 関数に返されたデータです。`$PSVersionTable` は、実行されている環境の情報も提供する PowerShell グローバル変数です。

AWS Lambda の context オブジェクト PowerShell

関数を実行すると、Lambda は、`$LambdaContext` 変数をハンドラで使用できるようにして、コンテキスト情報を渡します。この変数は、呼び出し、関数、および実行関数に関する情報を含むメソッドおよびプロパティです。

context プロパティ

- `FunctionName` - Lambda 関数の名前。
- `FunctionVersion` - 関数のバージョン。
- `InvokedFunctionArn` - 関数を呼び出すために使用される Amazon リソースネーム (ARN)。呼び出し元でバージョン番号またはエイリアスが指定されているかどうかを示します。
- `MemoryLimitInMB` - 関数に割り当てられたメモリの量。
- `AwsRequestId` - 呼び出しリクエストの ID。
- `LogGroupName` - 関数のロググループ。
- `LogStreamName` — 関数インスタンスのログストリーム。
- `RemainingTime` — 実行がタイムアウトするまでの残りのミリ秒数。
- `Identity` — (モバイルアプリケーション) リクエストを認可した Amazon Cognito ID に関する情報。
- `ClientContext` — (モバイルアプリケーション) クライアントアプリケーションが Lambda に提供したクライアントコンテキスト。
- `Logger` - 関数のロガーオブジェクト。

次の PowerShell コードスニペットは、コンテキスト情報の一部を出力する単純なハンドラー関数を示しています。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

PowerShell の AWS Lambda 関数ログ作成

AWS Lambda は、ユーザーに代わって Lambda 関数を自動的にモニタリングし、Amazon CloudWatch にログを送信します。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが用意されています。Lambda ランタイム環境は、各呼び出しの詳細をログストリームに送信し、関数のコードからのログやその他の出力を中継します。詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。

このページでは、AWS Command Line Interface、Lambda コンソール、または CloudWatch コンソールを使用して、Lambda 関数のコードからログ出力を生成する方法、またはアクセスログを生成する方法について説明します。

セクション

ログを返す関数の作成

関数コードからログを出力するには、Microsoft.PowerShell.Utility で cmdlets を使用するか、あるいは `stdout` または `stderr` に書き込む任意のログ作成モジュールを使用できます。次の例では `Write-Host` を使用しています。

Example function/Handler.ps1 - ログ記録

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example ログの形式

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 4000 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

.NET ランタイムは、呼び出しごとに START、END、REPORT の各行を記録します。レポート行には、次の詳細が示されます。

REPORT 行のデータフィールド

- RequestId - 呼び出しの一意のリクエスト ID。
- 所要時間 - 関数のハンドラーメソッドがイベントの処理に要した時間。

- 課金期間 - 呼び出しの課金対象の時間。
- メモリサイズ - 関数に割り当てられたメモリの量。
- 使用中の最大メモリ - 関数によって使用されているメモリの量。
- 初期所要時間 - 最初に処理されたリクエストについて、ハンドラーメソッド外で関数をロードしてコードを実行するためにランタイムにかかった時間。
- XRAY TraceId - トレースされたリクエストの場合、AWS X-Ray のトレース ID。
- SegmentId - トレースされたリクエストの場合、X-Ray のセグメント ID。
- Sampled - トレースされたリクエストの場合、サンプリング結果。

Lambda コンソールの使用

Lambda コンソールを使用して、Lambda 関数を呼び出した後のログ出力を表示できます。

組み込み Code エディタからコードがテスト可能である場合、[実行結果] でログを確認できます。コンソールのテスト機能を使用して関数を呼び出すと、[詳細] セクションで [ログ出力] を確認できます。

CloudWatch コンソールの使用

Amazon CloudWatch コンソールを使用して、すべての Lambda 関数呼び出しのログを表示できます。

CloudWatch コンソールでログを表示するには

1. CloudWatch コンソールの [Log groups (ロググループ)] ページを開きます。
2. 機能のロググループを選択します(/aws/lambda/**###**)
3. ログストリームを選択します

各ログストリームは、関数のインスタンスに相当します。ログストリームは、Lambda 関数を更新したとき、および複数の同時呼び出しを処理するために追加のインスタンスが作成されたときに表示されます。特定の呼び出しのログを検索するために、AWS X-Ray を使って関数をインストルメント化することをお勧めします。　X-Ray は、リクエストとログストリームの詳細をトレースに記録します。

AWS Command Line Interface (AWS CLI) を使用する

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - aws configure によるクイック設定

AWS CLI および --log-type コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、LogResultフィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out

を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


ログの削除

関数を削除しても、ロググループは自動的には削除されません。ログが無期限に保存されないようにするには、ロググループを削除するか、ログが自動的に削除されるまでの保存期間を設定します。

Rust で Lambda 関数を構築する

Rust はネイティブコードにコンパイルするため、Lambda で Rust コードを実行するために専用のランタイムは必要ありません。代わりに、「Rust ランタイムクライアント」を使用してプロジェクトをローカルで構築し、`provided.al2023` または `provided.al2` ランタイムを使用して Lambda にデプロイします。`provided.al2023` または `provided.al2` を使用するとき、Lambda はオペレーティングシステムを最新のパッチで自動的に最新の状態に保ちます。

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

Rust 用のツールおよびライブラリ

- AWS SDK for Rust: AWS SDK for Rust は、アマゾンウェブサービスのインフラストラクチャサービスとやり取りするための Rust API を提供します。
- 「Lambda 用 Rust ランタイムクライアント」: Rust ランタイムクライアントは実験的なパッケージです。大幅に変更される可能性があるため、本番環境での使用は推奨しません。
- 「Cargo Lambda」: このライブラリは、Rust で構築された Lambda 関数と連動するコマンドラインアプリケーションを提供します。
- 「Lambda HTTP」: このライブラリは、HTTP イベントと連動するラッパーを提供します。
- 「Lambda 拡張機能」: このライブラリは、Rust で Lambda 拡張機能を作成するためのサポートを提供します。
- 「AWS Lambda イベント」: このライブラリは、一般的なイベントソース統合のタイプ定義を提供します。

Rust 用のサンプル Lambda アプリケーション

- 「基本的な Lambda 関数」: 基本的なイベントの処理方法を示す Rust 関数。
- 「エラー処理機能付き Lambda 関数」: Lambda でカスタム Rust エラーの処理方法を示す Rust 関数。
- 「共有リソースを使用する Lambda 関数」: Lambda 関数を作成する前に共有リソースを初期化する Rust プロジェクト。

- 「Lambda HTTP イベント」: HTTP イベントを処理する Rust 関数。
- 「CORS ヘッダー付き Lambda HTTP イベント」: Tower を使用して CORS ヘッダーを挿入する Rust 関数。
- 「Lambda REST API」: Axum および Diesel を使用して PostgreSQL データベースに接続する REST API。
- 「サーバーレス Rust デモ」: Lambda のライブラリ、ログ記録、環境変数、AWS SDK の使い方を示す Rust プロジェクト。
- 「基本的な Lambda 拡張機能」: 基本的な拡張イベントの処理方法を示す Rust 拡張機能。
- 「Lambda ログ Amazon Data Firehose 拡張機能」: Kinesis Data Firehose に Lambda ログの送信方法を示す Rust 拡張機能。

トピック

- Rust の Lambda 関数ハンドラーの定義
- Rust の Lambda コンテキストオブジェクト
- Rust での HTTP イベントの処理
- .zip ファイルアーカイブを使用して Rust Lambda 関数をデプロイする
- Rust での Lambda 関数のログ記録

Rust の Lambda 関数ハンドラーの定義

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

Lambda 関数ハンドラーは、イベントを処理する関数コード内のメソッドです。関数が呼び出されると、Lambda はハンドラーメソッドを実行します。関数は、ハンドラーが応答を返すか、終了するか、タイムアウトするまで実行されます。

Lambda 関数コードを Rust 実行ファイルとして記述します。ハンドラー関数コードとメイン関数を実装し、以下のものを含めます。

- Rust の Lambda プログラミングモデルを実装する crates.io の lambda_runtime クレート。
- 依存関係には Tokio を含めます。Lambda 用の Rust ランタイムクライアントは、Tokio を使用して非同期呼び出しを処理します。

Example — JSON イベントを処理する Rust ハンドラー

次の例では、serde_json クレートを使用して基本的な JSON イベントを処理しています。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
 let payload = event.payload;
 let first_name = payload["firstName"].as_str().unwrap_or("world");
 Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
 lambda_runtime::run(service_fn(handler)).await
}
```

次の点に注意してください。

- use: Lambda 関数に必要なライブラリをインポートします。

- `async fn main`: Lambda 関数コードが実行するエントリポイント。Rust ランタイムクライアントは Tokio を非同期ランタイムとして使用するため、`#[tokio::main]` のメイン関数にアノテーションを付ける必要があります。
- `async fn handler(event: LambdaEvent<Value>) -> Result<Value,Error>`: これは Lambda ハンドラーの署名です。関数が呼び出されたときに実行するコードが含まれています。
  - `LambdaEvent<Value>`: これは、Lambda ランタイムが受け取るイベントと Lambda 関数コンテキストを記述するジェネリックタイプです。
  - `Result<Value, Error>`: この関数は `Result` タイプを返します。関数が成功すると、結果は JSON 値になります。関数が成功しなかった場合、結果はエラーになります。

共有ステートを使用する

Lambda 関数のハンドラーコードとは共有された変数を宣言して変更することができます。これらの変数は、関数がイベントを受け取る前に、初期化フェーズ 中の状態情報をロードするのに役立ちます。

Example — Amazon S3 クライアントを関数インスタンス間で共有

次の点に注意してください。

- `use aws_sdk_s3::Client`: この例では、`Cargo.toml` ファイル内の依存関係のリストに `aws-sdk-s3 = "0.26.0"` を追加する必要があります。
- `aws_config::from_env`: この例では、`Cargo.toml` ファイル内の依存関係のリストに `aws-config = "0.55.1"` を追加する必要があります。

```
use aws_sdk_s3::Client;
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};

#[derive(Deserialize)]
struct Request {
    bucket: String,
}

#[derive(Serialize)]
struct Response {
    keys: Vec<String>,
}
```

```
async fn handler(client: &Client, event: LambdaEvent<Request>) -> Result<Response,
 Error> {
    let bucket = event.payload.bucket;
    let objects = client.list_objects_v2().bucket(bucket).send().await?;
    let keys = objects
        .contents()
        .map(|s| s.iter().flat_map(|o| o.key().map(String::from)).collect())
        .unwrap_or_default();
    Ok(Response { keys })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
}
```

Rust の Lambda コンテキストオブジェクト

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

Lambda が関数を実行すると、ハンドラー LambdaEvent が受け取る にコンテキストオブジェクトが追加されます。このオブジェクトは、呼び出し、関数、および実行関数に関する情報を含むプロパティです。

context プロパティ

- `request_id`: Lambda サービスによって生成された AWS リクエスト ID。
- `deadline`: 現在の呼び出しの実行期限 (ミリ秒単位)。
- `invoked_function_arn`: Lambda 関数の Amazon リソースネーム (ARN) が呼び出されます。
- `xray_trace_id`: 現在の呼び出しの AWS X-Ray トレース ID。
- `client_content`:AWS モバイル SDK によって送信されたクライアントコンテキストオブジェクト。AWS モバイル SDK を使用して関数を呼び出さない限り、このフィールドは空です。
- `identity`: 関数を呼び出した Amazon Cognito ID。Lambda API への呼び出しリクエストが Amazon Cognito ID プールによって発行された AWS 認証情報を使用して行われた場合を除き、このフィールドは空です。
- `env_config`: ローカル環境変数からの Lambda 関数の構成。このプロパティには、関数名、メモリ割り当て、バージョン、ログストリームなどの情報が含まれます。

context の呼び出し情報へのアクセス

Lambda 関数は、環境と呼び出しリクエストに関するメタデータにアクセスできます。関数ハンドラが受け取る `LambaEvent` オブジェクトには、`context` メタデータが含まれます。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
```

```
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Rust での HTTP イベントの処理

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

Amazon API Gateway、アプリケーションロードバランサー、および Lambda 関数 URL は HTTP イベントを Lambda に送信できます。crates.io の aws_lambda_events クレートを使用して、これらのソースからのイベントを処理できます。

Example — API Gateway プロキシリクエストの処理

次の点に注意してください。

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: aws_lambda_events クレートには、多くの Lambda イベントが含まれています。コンパイル時間を短縮するには、機能フラグを使用して必要なイベントをアクティブにします。例えば、`aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }` などです。
- `use http::HeaderMap`: このインポートでは、依存関係に http クレートを追加する必要があります。

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
```

```
    };
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

また、Lambda 用の Rust ランタイムクライアントでは、これらのイベントタイプを抽象化することもでき、どのサービスがイベントを送信するかに関係なく、ネイティブ HTTP タイプを使用できます。次のコードは前の例と同等で、Lambda 関数 URL、Application Load Balancer、API Gateway ですぐに使用できます。

> **Note**
>
> lambda_http クレートは、その下の lambda_runtime クレートを使用します。`lambda_runtime` を個別にインポートする必要はありません。

Example — HTTP リクエストの処理

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

`lambda_http` の使用方法の別の例については、AWS Labs GitHub リポジトリにある「http-axum コードサンプル」を参照してください。

Rust 用サンプル HTTP Lambda イベント

- 「Lambda HTTP イベント」: HTTP イベントを処理する Rust 関数。
- 「CORS ヘッダー付き Lambda HTTP イベント」: Tower を使用して CORS ヘッダーを挿入する Rust 関数。
- 共有リソースを使用する Lambda HTTP イベント: 関数ハンドラーが作成される前に初期化された共有リソースを使用する Rust 関数。

.zip ファイルアーカイブを使用して Rust Lambda 関数をデプロイする

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

このページでは、Rust 関数をコンパイルし、Cargo Lambda を使用して コンパイルしたバイナリを AWS Lambda にデプロイする方法について説明します。また、AWS Command Line Interface と AWS Serverless Application Model CLI を使用してコンパイルしたバイナリをデプロイする方法も示します。

セクション

前提条件

- Rust
- AWS Command Line Interface (AWS CLI) バージョン 2

MacOS、Windows、Linux で Rust 関数をビルドする

次のステップでは、Rust で最初の Lambda 関数のプロジェクトを作成し、Cargo Lambda でコンパイルする方法を示しています。

1. Cargo サブコマンドである Cargo Lambda をインストールします。これは macOS、Windows、Linux で Lambda 用の Rust 関数をコンパイルします。

   Python 3 がインストールされているシステムに Cargo Lambda をインストールするには、次の pip を使用してください。

```
pip3 install cargo-lambda
```

macOS または Linux に Cargo Lambda をインストールするには、Homebrew を使用します。

```
brew tap cargo-lambda/cargo-lambda
brew install cargo-lambda
```

Windows に Cargo Lambda をインストールし、Scoop を使用します。

```
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

その他のオプションについては、Cargo Lambda ドキュメントの「インストール」を参照してください。

2. パッケージ構造を作成します。このコマンドは、`src/main.rs` に基本的な関数コードを作成します。このコードをテストに使用することも、独自のコードで置き換えることもできます。

```
cargo lambda new my-function
```

3. パッケージのルートディレクトリ内で build サブコマンドを実行して、関数内のコードをコンパイルします。

```
cargo lambda build --release
```

(オプション) Lambda でAWS Graviton2 を使用する場合は、`--arm64` フラグを追加して ARM CPU 用のコードをコンパイルします。

```
cargo lambda build --release --arm64
```

4. Rust 関数をデプロイする前に、マシンに AWS 認証情報を設定します。

```
aws configure
```

Cargo Lambda による Rust 関数バイナリのデプロイ

deploy サブコマンドを使用して、コンパイルされたバイナリを Lambda にデプロイします。このコマンドは実行ロールを作成し、次に Lambda 関数を作成します。既存の実行ロールを指定するには、--iam-role フラグを使用します。

```
cargo lambda deploy my-function
```

AWS CLI を使用して Rust 関数バイナリをデプロイする

AWS CLI を使用してバイナリをデプロイすることもできます。

1. .zip デプロイパッケージをビルドするには、ビルドサブコマンドを使用します。

   ```
   cargo lambda build --release --output-format zip
   ```

2. .zip パッケージを Lambda にデプロイします。--role には、実行ロールの ARN を指定します。

   ```
   aws lambda create-function --function-name my-function \
        --runtime provided.al2023 \
        --role arn:aws:iam::111122223333:role/lambda-role \
        --handler rust.handler \
        --zip-file fileb://target/lambda/my-function/bootstrap.zip
   ```

AWS SAM CLI による Rust 関数バイナリのデプロイ

AWS SAM CLI を使用してバイナリをデプロイすることもできます。

1. リソースとプロパティの定義を含む AWS SAM テンプレートを作成します。詳細については、「AWS Serverless Application Model デベロッパーガイド」の「AWS::Serverless::Function」を参照してください。

   Example Rust バイナリの SAM リソースとプロパティ定義

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: SAM template for Rust binaries
   Resources:
     RustFunction:
   ```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: target/lambda/my-function/
      Handler: rust.handler
      Runtime: provided.al2023
Outputs:
  RustFunction:
    Description: "Lambda Function ARN"
    Value: !GetAtt RustFunction.Arn
```

2. build サブコマンドを使用して関数をコンパイルします。

```
cargo lambda build --release
```

3. sam deploy コマンドを使用して、関数を Lambda にデプロイします。

```
sam deploy --guided
```

AWS SAM CLI を使用した Rust 関数をビルドする方法の詳細については、AWS Serverless Application Model 開発者ガイドの「Cargo Lambda による Rust Lambda 関数のビルド」を参照してください。

Cargo Lambda を使用して Rust 関数を呼び出す

invoke サブコマンドを使用して、ペイロードを用いて関数をテストします。

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

AWS CLI を使用して Rust 関数を呼び出すには

また、AWS CLI を使用して関数を呼び出すこともできます。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

Rust での Lambda 関数の口グ記録

> **Note**
>
> 「Rust ランタイムクライアント」は実験的なパッケージです。これは変更される可能性があり、評価のみを目的としています。

AWS Lambda は、ユーザーに代わって Lambda 関数を自動的にモニタリングし、Amazon にログを送信します CloudWatch。Lambda 関数には、関数のインスタンスごとに CloudWatch Logs ロググループとログストリームが付属しています。Lambda ランタイム環境は、各呼び出しの詳細をログストリームに送信し、関数のコードからのログやその他の出力を中継します。詳細については、「AWS Lambda での Amazon CloudWatch Logs の使用」を参照してください。このページでは、Lambda 関数のコードからログ出力を生成する方法について説明します。

ログを書く関数の作成

関数コードからログを出力するには、`println!` マクロなどの `stdout` または `stderr` に書き込む任意のログ記録ライブラリを使用できます。次の例は、`println!`を使用して、関数ハンドラーの開始時と終了前にメッセージを印刷します。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

トレーシングクレートによる高度なロギ記録

トレーシングは、構造化されたイベントベースの診断情報を収集するために Rust プログラムをインストルメントするためのフレームワークです。このフレームワークには、構造化された JSON ログ

メッセージの作成など、ロギング出力レベルと形式をカスタマイズするユーティリティが用意されています。このフレームワークを使用するには、関数ハンドラーを実装する前に subscriber を初期化する必要があります。次に、debug、info、error などのトレースマクロを使用して、各シナリオに必要なログのレベルを指定できます。

Example — トレーシングクレートの使用

次の点に注意してください。

- tracing_subscriber::fmt().json(): このオプションが含まれる場合、ログは JSON でフォーマットされます。このオプションを使用するには、tracing-subscriber の json 機能を依存関係に含める必要があります (例: tracing-subscriber = { version = "0.3.11", features = ["json"] })。
- #[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]: このアノテーションは、ハンドラが呼び出されるたびにスパンを生成します。このスパンは、各ログ行にリクエスト ID を追加します。
- { %first_name }: このコンストラクトは、使用されているログ行に first_name フィールドを追加します。このフィールドの値は、同じ名前の変数に対応します。

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
```

```
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

この Rust 関数が呼び出されると、次のような口グ行が 2 行出力されます。

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

他の AWS サービスからのイベントを使用した Lambda の呼び出し

一部の AWS サービスは、トリガーを使用して Lambda 関数を直接呼び出すことができます。これらのサービスはイベントを Lambda にプッシュし、指定されたイベントが発生すると即時に関数が呼び出されます。トリガーは、個別のイベントやリアルタイム処理に適しています。Lambda コンソールを使用してトリガーを作成すると、コンソールは対応する AWS サービスと連携して、そのサービスでイベント通知を設定します。実際には、トリガーは Lambda ではなく、イベントを生成するサービスによって保存および管理されます。

イベントは JSON 形式で構造化されたデータです。JSON 構造は、それを生成するサービスとイベントタイプによって異なりますが、すべて関数がイベントを処理するために必要なデータを含んでいます。

関数には複数のトリガーを持つことができます。各トリガーは、関数を単独に呼び出すクライアントとして機能し、Lambda が関数に渡す各イベントには、1 つのトリガーからのデータしかありません。Lambda は、イベントドキュメントをオブジェクトに変換して関数ハンドラに渡します。

サービスに応じて、イベント駆動型呼び出しは同期または非同期になります。

- 同期呼び出しの場合、イベントを生成するサービスは、関数からのレスポンスを待機します。そのサービスは、関数がレスポンスで返す必要があるデータを定義します。サービスはエラー戦略を制御します (エラー発生時に再試行するかどうかなど)。
- 非同期呼び出しの場合、Lambda は関数に渡す前に、イベントをキューに入れます。Lambda はイベントをキューに入れると、イベントを生成したサービスにすぐに成功レスポンスを送信します。関数がイベントを処理した後、Lambda はイベント生成サービスにレスポンスを返しません。

トリガーの作成

トリガーを作成する最も簡単な方法は、Lambda コンソールを使用することです。コンソールを使用してトリガーを作成すると、Lambda によって、必要なアクセス許可が自動的に関数のリソースベースのポリシーに追加されます。

Lambda コンソールを使用してトリガーを作成するには

1. Lambda コンソールの [関数ページ] を開きます。
2. トリガーを作成する関数を選択します。

3. [関数の概要] ペインで、[トリガーを追加] を選択します。
4. 関数を呼び出す AWS サービスを選択します。
5. トリガー設定 ペインにオプションを入力し、 [追加] を選択します。関数を呼び出すために選択した AWS のサービス に応じて、トリガー設定オプションは異なります。

Lambda 関数を呼び出すことができるサービス

次の表は、Lambda 関数を呼び出すことができるサービスをリストしています。

サービス	呼び出し方法
Amazon Alexa	イベント駆動型、同期呼び出し
Amazon Managed Streaming for Apache Kafka	イベントソースマッピング
セルフマネージド Apache Kafka	イベントソースマッピング
Amazon API Gateway	イベント駆動型、同期呼び出し
AWS CloudFormation	イベント駆動型;、非同期呼び出し
Amazon CloudFront (Lambda@Edge)	イベント駆動型、同期呼び出し
Amazon CloudWatch Logs	イベント駆動型;、非同期呼び出し
AWS CodeCommit	イベント駆動型;、非同期呼び出し
AWS CodePipeline	イベント駆動型;、非同期呼び出し
Amazon Cognito	イベント駆動型、同期呼び出し
AWS Config	イベント駆動型;、非同期呼び出し
Amazon Connect	イベント駆動型、同期呼び出し
Amazon DynamoDB	イベントソースマッピング

サービス	呼び出し方法
Amazon Elastic File System	特殊な統合
Elastic Load Balancing (Application Load Balancer)	イベント駆動型、同期呼び出し
AWS IoT	イベント駆動型;、非同期呼び出し
Amazon Kinesis	イベントソースマッピング
Amazon Data Firehose	イベント駆動型、同期呼び出し
Amazon Lex	イベント駆動型、同期呼び出し
Amazon MQ	イベントソースマッピング
Amazon Simple Email Service	イベント駆動型;、非同期呼び出し
Amazon Simple Notification Service	イベント駆動型;、非同期呼び出し
Amazon Simple Queue Service	イベントソースマッピング
Amazon Simple Storage Service (Amazon S3)	イベント駆動型;、非同期呼び出し
Amazon Simple Storage Service Batch	イベント駆動型、同期呼び出し
シークレットマネージャー	イベント駆動型、同期呼び出し
Amazon VPC Lattice	イベント駆動型、同期呼び出し
AWS X-Ray	特殊な統合

一般的な Lambda アプリケーションの種類とユースケース

AWS Lambda でアプリケーションを構築する場合のコアコンポーネントは、Lambda 関数とトリガーです。Lambda 関数とは、イベントを処理するコードやランタイムです。一方、トリガーとは、関数を呼び出す AWS のサービスまたはアプリケーションです。説明のため、以下のシナリオを想定してください。

- ファイル処理 - 写真共有アプリケーションがあるとします。ユーザーはこのアプリケーションを使用して、写真をアップロードし、アプリケーションがユーザーの写真を Amazon S3 バケットに保存します。そこで、アプリケーションは各ユーザーの写真のサムネイルバージョンを作成し、ユーザーのプロフィールページに表示します。このシナリオでは、自動的にサムネイルを作成する Lambda 関数の作成を選択できます。Amazon S3 はオブジェクト作成イベントを公開し、Lambda 関数を呼び出すことのできる、サポートされた AWS イベントソースの 1 つです。Lambda 関数コードは S3 バケットから写真オブジェクトを読み取り、サムネイルバージョンを作成して、それを別の S3 バケットに保存できます。
- データと分析 - 分析アプリケーションを構築して、raw データを DynamoDB テーブルに保存するとします。テーブル内の項目を書き込み、更新、削除した場合、DynamoDB Streams は、項目の更新イベントをテーブルに関連付けられたストリーミングに発行できます。この場合、イベントデータは、項目キー、イベント名 (挿入、更新、削除など)、その他関連する詳細情報を提供します。raw データを集約することで、カスタムメトリクスを生成する Lambda 関数を記述できます。
- ウェブサイト - ウェブサイトを作成して、Lambda でバックエンドロジックをホストするとします。Amazon API Gateway を HTTP エンドポイントとして使用し、HTTP 経由で Lambda 関数を呼び出せます。これにより、ウェブクライアントが API を呼び出し、API Gateway がリクエストを Lambda にルーティングできるようになります。
- モバイルアプリケーション - イベントを作成するカスタムモバイルアプリケーションがあるとします。カスタムアプリケーションによって、発行されるイベントを処理するための Lambda 関数を作成できます。例えば、カスタムモバイルアプリケーション内のクリックを処理する Lambda 関数を設定できます。

AWS Lambda では、イベントソースとして、多くの AWS のサービスをサポートしています。詳細については、「他の AWS サービスからのイベントを使用した Lambda の呼び出し」を参照してください。これらのイベントソースを、Lambda 関数をトリガーするように設定する場合、Lambda 関数はイベントが発生すると自動的に呼び出されます。追跡するイベントや、呼び出す Lambda 関数を識別するイベントソースマッピングを定義します。

以下は、イベントソースの概要と end-to-end エクスペリエンスの仕組みの例です。

例 1: Amazon S3 はイベントをプッシュし、Lambda 関数を呼び出します。

Amazon S3 は、PUT、POST、COPY、DELETE オブジェクトイベントなど、異なる種類のイベントをバケットに公開できます。バケットの通知機能を使用して、特定の種類のイベントが発生した場合に Lambda 関数を呼び出し、Amazon S3 に指示するイベントソースマッピングを設定できます。

一般的な手順は以下のとおりです。

1. ユーザーはバケット内にオブジェクトを作成します。
2. Amazon S3 がオブジェクト作成イベントを検出します。
3. Amazon S3 は、実行ロールによって付与されたアクセス許可を使用して、Lambda 関数を呼び出します。
4. AWS Lambda はイベントをパラメータとして指定し、Lambda 関数を実行します。

関数をバケット通知アクションとして呼び出すように、Amazon S3 を設定します。関数を呼び出すアクセス許可を Amazon S3 に付与するには、関数のリソースベースのポリシーを更新します。

例 2: AWS Lambda は Kinesis ストリームからイベントを取り出し、Lambda 関数を呼び出します。

ポーリングベースのイベントソースの場合、AWS Lambda がソースをポーリングし、ソースでレコードが検出されると、Lambda 関数が呼び出されます。

- CreateEventSourceMapping
- UpdateEventSourceMapping

次の手順は、カスタムアプリケーションが Kinesis ストリーミングにどのようにレコードを書き込むかを示します。

1. カスタムアプリケーションは Kinesis ストリームにレコードを書き込みます。
2. AWS Lambda はストリーミングのポーリングを継続し、サービスによってストリーミングに新しいレコードが検出されると Lambda 関数を呼び出します。AWS Lambda は、どのストリーミングをポーリングし、どの Lambda 関数を呼び出すかを、Lambda でユーザーが作成したイベントソースマッピングに基づいて判断します。
3. Lambda 関数は、受信イベントに伴って呼び出されます。

ストリーミングベースのイベントソースを使用する場合、イベントソースマッピングを AWS Lambda に作成します。Lambda は、ストリームから項目を読み込み、関数を同期的に呼び出します。この関数を呼び出すためのアクセス許可を Lambda に付与する必要はありませんが、ストリーミングから読み取るためのアクセス許可は必要です。

Alexa で AWS Lambda を使用する

Lambda 関数を使用して、Amazon Echo の音声アシスタント Alexa に新しいスキルを与えるサービスをビルドできます。Alexa Skills Kit は、Lambda 関数として実行している独自のサービスを使用して、新しいスキルを作成するための API、ツール、およびドキュメントを提供します。Amazon Echo ユーザーは、Alexa に質問やリクエストを行うことで、これらの新しいスキルにアクセスできます。

Alexa Skills Kit は で利用できます GitHub。

- Alexa Skills Kit SDK for Java
- Alexa Skills Kit SDK for Node.js
- Alexa Skills Kit SDK for Python

Example Alexa スマートホームイベント

```
{
  "header": {
    "payloadVersion": "1",
    "namespace": "Control",
    "name": "SwitchOnOffRequest"
  },
  "payload": {
    "switchControlAction": "TURN_ON",
    "appliance": {
      "additionalApplianceDetails": {
        "key2": "value2",
        "key1": "value1"
      },
      "applianceId": "sampleId"
    },
    "accessToken": "sampleAccessToken"
  }
}
```

詳細については、Alexa Skills Kit を使用したスキルの構築ガイドの「AWS Lambda Lambda 関数としてカスタムスキルをホストする」を参照してください。

Amazon API Gateway エンドポイントを使用した Lambda 関数の呼び出し

Amazon API Gateway を使用して、Lambda 関数の HTTP エンドポイントを持つウェブ API を作成できます。API Gateway は、HTTP リクエストを Lambda 関数にルーティングするウェブ API を、作成および文書化するためのツールを提供します。認証および承認のコントロールにより、API へのアクセスを保護できます。API は、インターネット経由でトラフィックを処理することも、VPC 内でのみアクセス可能にすることもできます。

API 内のリソースは、GET や POST などの 1 つ以上のメソッドを定義します。メソッドには、Lambda 関数または別の統合タイプにリクエストをルーティングする統合があります。各リソースとメソッドを個別に定義することも、特殊なリソースとメソッドタイプを使用して、パターンに一致するすべてのリクエストを照合することもできます。プロキシリソースは、リソースの下にあるすべてのパスをキャッチします。ANY メソッドは、すべての HTTP メソッドをキャッチします。

セクション

API タイプの選択

API Gateway は、Lambda 関数を呼び出す 3 種類の API をサポートしています。

- HTTP API: 軽量で低レイテンシーの RESTful API。
- REST API: カスタマイズ可能で機能豊富な RESTful API。
- WebSocket API: 全二重通信のためにクライアントとの永続的な接続を維持するウェブ API。

HTTP API と REST API は、両方とも HTTP リクエストを処理し、レスポンスを返す RESTful API です。HTTP API は新しいバージョンであり、API Gateway バージョン 2 API を使用して構築されています。HTTP API では、次の機能が新しく追加されました。

HTTP API の機能

- 自動デプロイ - ルートまたは統合を変更した場合、変更は自動デプロイが有効になっているステージに自動的にデプロイされます。
- デフォルトステージ - API の URL のルートパスでリクエストを処理するデフォルトステージ (`$default`) を作成できます。名前付きステージの場合は、パスの先頭にステージ名を含める必要があります。
- CORS 設定 - CORS ヘッダーを関数コードで手動で追加する代わりに、送信レスポンスに CORS ヘッダーを追加するように API を設定できます。

REST API は、当初から API Gateway でサポートされている従来の RESTful API です。REST APIには現在、より多くのカスタマイズ、統合、および管理機能があります。

REST API の機能

- 統合タイプ - REST API は、カスタム Lambda 統合をサポートします。カスタム統合では、リクエストのボディだけを関数に送信するか、関数に送信する前にリクエストボディに変換テンプレートを適用できます。
- アクセスコントロール - REST API では、認証と認可のためのより多くのオプションがサポートされています。
- モニタリングとトレース - REST API では、AWS X-Ray のトレースと追加のログ記録オプションがサポートされます。

詳細な比較については、API Gateway デベロッパーガイドの HTTP API と REST API 間で選択する を参照してください。

WebSocket API も API Gateway バージョン 2 API を使用し、同様の機能セットをサポートします。クライアントと API 間の永続的な接続を活用できるアプリケーションには、WebSocket API を使用します。WebSocket API は全二重通信を提供します。つまり、クライアントと API の両方がレスポンスを待たずにメッセージを継続的に送信できます。

HTTP API では、簡略化されたイベント形式 (バージョン 2.0) がサポートされています。次の例は、HTTP API からのイベントを示しています。

Example event-v2.json - API Gateway プロキシイベント (HTTP API)

```
{
    "version": "2.0",
    "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
    "rawPath": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
    "rawQueryString": "",
    "cookies": [
        "s_fid=7AABXMPL1AFD9BBF-0643XMPL09956DE2",
        "regStatus=pre-register"
    ],
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
        "accept-encoding": "gzip, deflate, br",
        ...
    },
    "requestContext": {
        "accountId": "123456789012",
        "apiId": "r3pmxmplak",
        "domainName": "r3pmxmplak.execute-api.us-east-2.amazonaws.com",
        "domainPrefix": "r3pmxmplak",
        "http": {
            "method": "GET",
            "path": "/default/nodejs-apig-function-1G3XMPLZXVXYI",
            "protocol": "HTTP/1.1",
            "sourceIp": "205.255.255.176",
            "userAgent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36"
        },
        "requestId": "JKJaXmPLvHcESHA=",
        "routeKey": "ANY /nodejs-apig-function-1G3XMPLZXVXYI",
        "stage": "default",
        "time": "10/Mar/2020:05:16:23 +0000",
        "timeEpoch": 1583817383220
    },
    "isBase64Encoded": true
}
```

詳細については、API Gateway デベロッパーガイドの AWS Lambda integrations を参照してください。

エンドポイントの Lambda 関数への追加

パブリックエンドポイントを Lambda 関数に追加するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [機能の概要] で、[トリガーを追加] を選択します。
4. [API Gateway] を選択します。
5. [Create an API] (API の作成) または [Use an existing API] (既存の API の使用) を選択します。
   a. [New API] (新しい API): [API type] (API タイプ) で、[HTTP API] を選択します。詳細については、「API タイプ」を参照してください。
   b. [Existing API] (既存の API): ドロップダウンメニューから API を選択するか、API ID (r3pmxmplak など) を入力します。
6. [Security (セキュリティ)] で、[Open (開く)] を選択します。
7. [Add] (追加) をクリックします。

プロキシ統合

API Gateway API は、ステージ、リソース、メソッド、および統合で構成されています。ステージとリソースによって、エンドポイントのパスが決まります。

API パス形式

- `/prod/` - `prod` ステージおよびルートリソース。
- `/prod/user` - `prod` ステージおよび `user` リソース。
- `/dev/{proxy+}` - `dev` ステージ内の任意のルート。
- `/` - (HTTP API) デフォルトのステージおよびルートリソース。

Lambda 統合は、パスと HTTP メソッドの組み合わせを Lambda 関数にマッピングします。HTTP リクエストのボディをそのまま渡すように (カスタム統合)、またはヘッダー、リソース、パス、メソッドなどすべてのリクエスト情報を含むドキュメントにリクエストボディをカプセル化するように、API Gateway を設定できます。

詳細については、「API Gateway で Lambda プロキシ統合を設定する」を参照してください。

イベント形式

Amazon API Gateway は、HTTP リクエストの JSON 表現を含むイベントを使用して、関数を同期的に呼び出します。カスタム統合の場合、イベントはリクエストのボディです。プロキシ統合の場合、イベントは定義された構造を持ちます。次の例は、API Gateway REST API からのプロキシイベントを示しています。

Example event.json API Gateway プロキシイベント (REST API)

```
{
      "resource": "/",
      "path": "/",
      "httpMethod": "GET",
      "requestContext": {
          "resourcePath": "/",
          "httpMethod": "GET",
          "path": "/Prod/",
          ...
      },
      "headers": {
          "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9",
          "accept-encoding": "gzip, deflate, br",
          "Host": "70ixmpl4fl.execute-api.us-east-2.amazonaws.com",
          "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/80.0.3987.132 Safari/537.36",
          "X-Amzn-Trace-Id": "Root=1-5e66d96f-7491f09xmpl79d18acf3d050",
          ...
      },
      "multiValueHeaders": {
          "accept": [
              "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
          ],
          "accept-encoding": [
              "gzip, deflate, br"
          ],
          ...
      },
      "queryStringParameters": null,
      "multiValueQueryStringParameters": null,
      "pathParameters": null,
      "stageVariables": null,
```

```
      "body": null,
      "isBase64Encoded": false
  }
```

レスポンスの形式

API Gateway は関数からのレスポンスを待ち、その結果を発信者に中継します。カスタム統合の場合は、統合レスポンスとメソッドレスポンスを定義して、関数からの出力を HTTP レスポンスに変換します。プロキシ統合の場合、関数はレスポンスを特定の形式で表現して応答する必要があります。

次の例は、Node.js 関数からのレスポンスオブジェクトを示しています。レスポンスオブジェクトは、JSON ドキュメントを含む成功した HTTP レスポンスを表します。

Example index.js - プロキシ統合レスポンスオブジェクト (Node.js)

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
    }
```

Lambda ランタイムは、レスポンスオブジェクトを JSON にシリアル化し、API に送信します。API はレスポンスを解析し、それを使用して HTTP レスポンスを作成します。次に、そのレスポンスを元のリクエストを実行したクライアントに送信します。

Example HTTP レスポンス

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
```

```
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

アクセス許可

Amazon API Gateway は、関数のリソースベースのポリシーから関数を呼び出すアクセス許可を取得します。API 全体に対する呼び出しアクセス許可を付与したり、ステージ、リソース、またはメソッドに対する制限付きアクセスを付与したりできます。

Lambda コンソールまたは API Gateway コンソールを使用するか、AWS SAM テンプレートで API を関数に追加すると、関数のリソースベースのポリシーが自動的に更新されます。関数ポリシーの例を以下に示します。

Example 関数ポリシー

```
{
  "Version": "2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-east-2:111122223333:ktyvxmpls1/*/
GET/"
        }
      }
```

```
    }
  ]
}
```

次の API オペレーションを使用して、関数ポリシーのアクセス許可を手動で管理できます。

- AddPermission
- RemovePermission
- GetPolicy

既存の API への呼び出しアクセス許可を付与するには、add-permission コマンドを使用します。

```
aws lambda add-permission --function-name my-function \
--statement-id apigateway-get --action lambda:InvokeFunction \
--principal apigateway.amazonaws.com \
--source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

以下の出力が表示されます。

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> 関数と API が異なる AWS リージョンにある場合、ソース ARN のリージョン識別子は、API のリージョンではなく、関数のリージョンと一致している必要があります。API Gateway が関数を呼び出すと、API の ARN に基づくリソース ARN が使用されますが、その関数のリージョンと一致するように変更されます。

この例のソース ARN は、API のデフォルトステージにあるルートリソースの GET メソッドでの統合に ID mnh1xmpli7 のアクセス許可を付与します。ソース ARN でアスタリスクを使用すると、複数のステージ、メソッド、またはリソースにアクセス許可を付与できます。

リソースパターン

- `mnh1xmpli7/*/GET/*` - すべてのステージのすべてのリソースの GET メソッド。
- `mnh1xmpli7/prod/ANY/user` - `prod` ステージの `user` リソースにおけるすべてのメソッド。
- `mnh1xmpli7/*/*/*` - すべてのステージのすべてのリソースの任意のメソッド。

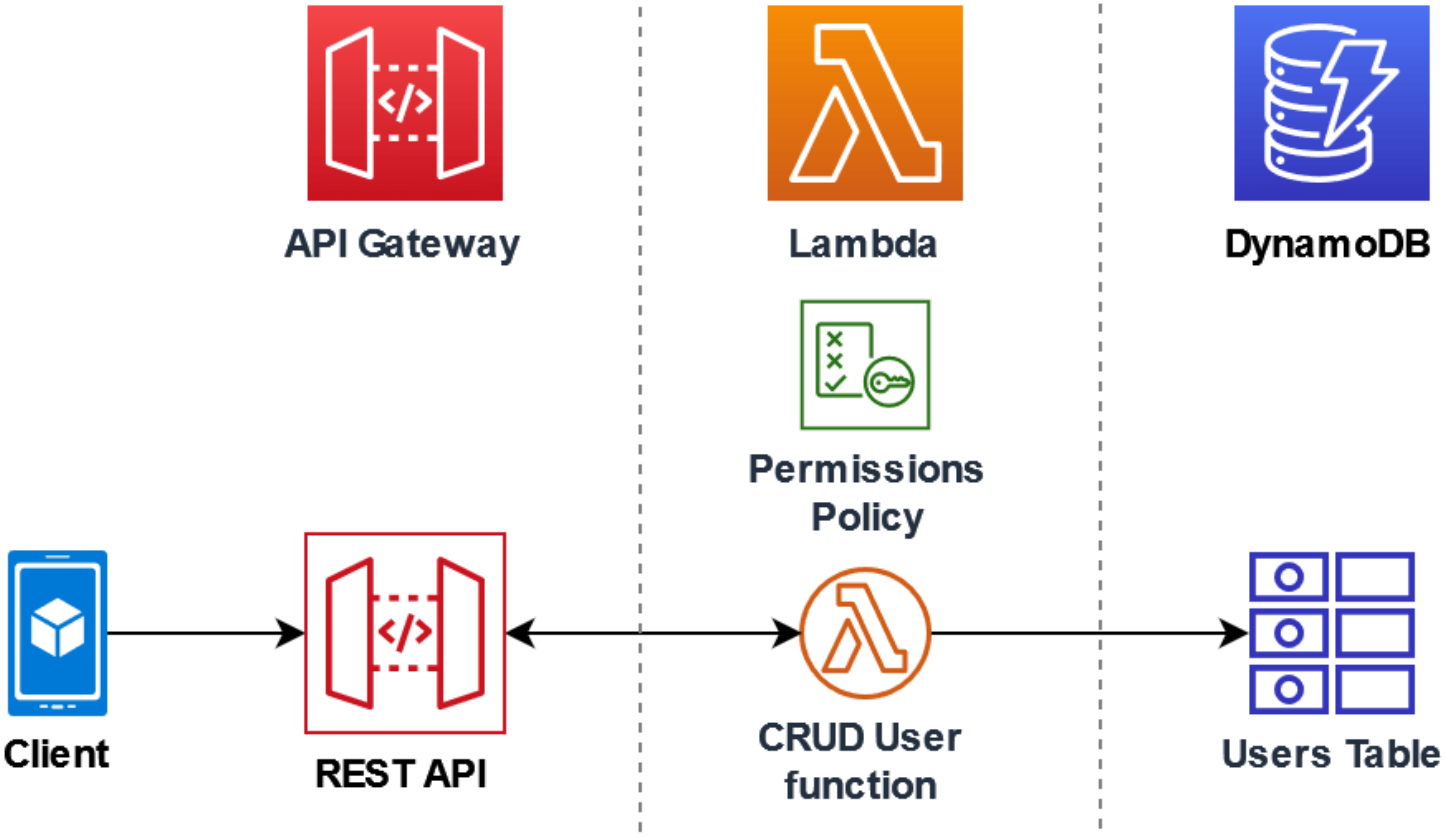
ポリシーの表示とステートメントの削除の詳細については、「」を参照してくださいリソースベースのポリシーのクリーンアップ

サンプルアプリケーション

Node.js を使用した API Gateway のサンプルアプリには、AWS SAM テンプレートを使用した関数が含まれています。このテンプレートは、AWS X-Ray のトレースが有効にされた REST API を作成します。これには、デプロイ、関数の呼び出し、API のテスト、およびクリーンアップ用のスクリプトも含まれます。

チュートリアル: API Gateway で Lambda を使用する

このチュートリアルでは、HTTP リクエストを使用して Lambda 関数を呼び出す REST API を作成します。Lambda 関数は、DynamoDB テーブルで作成、読み取り、更新、および削除 (CRUD) 操作を実行します。このチュートリアルで提供される関数はデモ用ですが、任意の Lambda 関数を呼び出すことができる API Gateway REST API を設定する方法を学びます。



API Gateway を使用することで、Lambda 関数を呼び出すためのセキュアな HTTP エンドポイントがユーザーに提供されるとともに、トラフィックのスロットリングと、API 呼び出しの自動的な検証と承認によって、関数に対する大量の呼び出しを管理するためにも役立ちます。API Gateway は、AWS Identity and Access Management (IAM) と Amazon Cognito を使用した柔軟なセキュリティコントロールも提供します。これは、アプリケーションへの呼び出しに事前承認が必要なユースケースに役立ちます。

このチュートリアルは、以下の段階を通じて完了します。

1. DynamoDB テーブルで操作を実行するための Lambda 関数をPython または Node.js で作成し、設定する。
2. API Gateway で、Lambda 関数に接続するための REST API を作成する。
3. DynamoDB テーブルを作成し、コンソールで Lambda 関数を使用してテーブルをテストする。
4. API をデプロイし、ターミナルで curl を使用してセットアップ全体をテストする。

これらの段階を完了することにより、あらゆる規模で Lambda 関数をセキュアに呼び出すことができる HTTP エンドポイントを作成するために API Gateway を使用する方法を学びます。また、API

をデプロイする方法と、それをコンソールでテスト、およびターミナルを使用して HTTP リクエストを送信することでテストする方法も学びます。

セクション

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] を選んで、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理できます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウントのメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

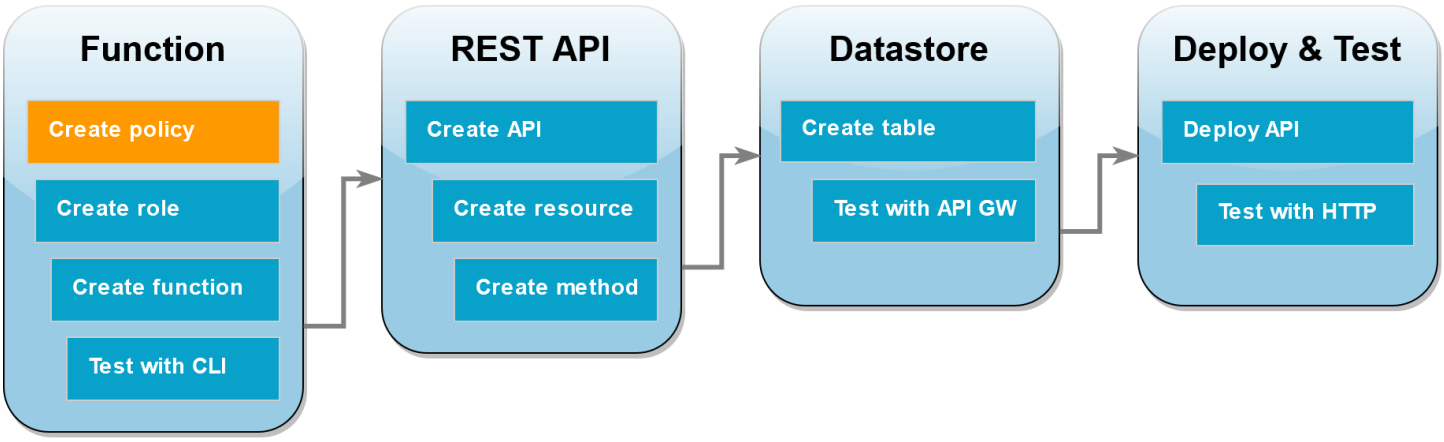
AWS Command Line Interface のインストール

AWS Command Line Interface をまだインストールしていない場合は、「最新バージョンの AWS CLI のインストールまたは更新」にある手順に従ってインストールしてください。

このチュートリアルでは、コマンドを実行するためのコマンドラインターミナルまたはシェルが必要です。Linux および macOS では、任意のシェルとパッケージマネージャーを使用してください。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (`zip` など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。

許可ポリシーを作成する



Lambda 関数の実行ロールを作成する前に、必要な AWS リソースにアクセスするための許可を関数に付与する許可ポリシーを作成する必要があります。このチュートリアルでは、このポリシーが、DynamoDB テーブルで CRUD 操作を実行し、Amazon CloudWatch Logs に書き込むことを Lambda に許可します。

ポリシーを作成する

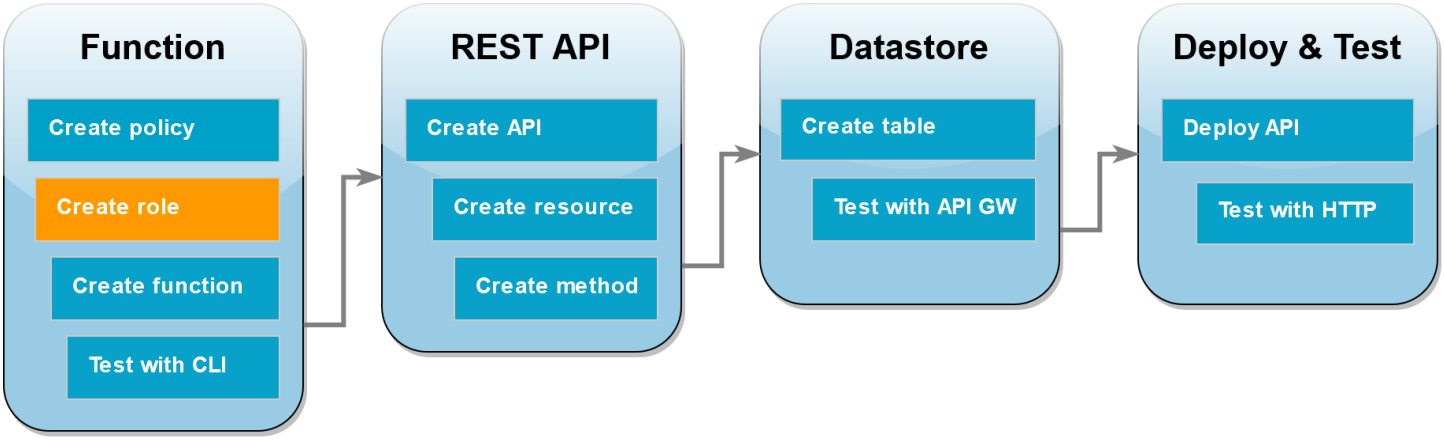
1. IAM コンソールのポリシーページを開きます。
2. [ポリシーの作成] を選択します。
3. [JSON] タブを選択して、次のカスタムポリシーを JSON エディタに貼り付けます。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
```

```
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. [次へ: タグ] を選択します。
5. [次へ: レビュー] を選択します。
6. [ポリシーの確認] でポリシーの [名前] に「**`lambda-apigateway-policy`**」と入力します。
7. [ポリシーの作成] を選択します。

実行ロールを作成する



実行ロールとは、AWS サービスとリソースにアクセスする許可を Lambda 関数に付与する AWS Identity and Access Management (IAM) ロールです。関数が DynamoDB テーブルで操作を実行できるようにするには、前のステップで作成した許可ポリシーをアタッチします。

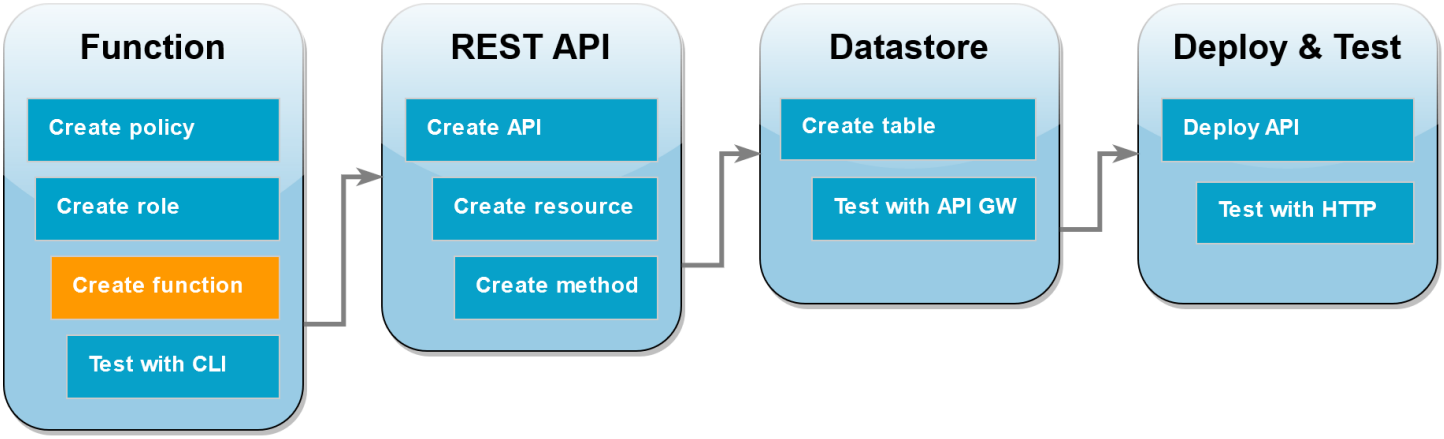
実行ロールを作成して、カスタム許可ポリシーをアタッチする

1. IAM コンソールのロールページを開きます。
2. [ロールの作成] を選択します。
3. 信頼されたエンティティには、[AWS サービス] を選択し、ユースケースには [Lambda] を選択します。

4. [次へ] をクリックします。
5. ポリシー検索ボックスに、「**lambda-apigateway-policy**」と入力します。
6. 検索結果で作成したポリシー (lambda-apigateway-policy) を選択し、[次へ] を選択します。
7. [Role details] (ロールの詳細)で [Role name] (ロール名) に **lambda-apigateway-role** を入力してから、[Create role] (ロールを作成) を選択します。

このチュートリアルの後半で、先ほど作成したロールの Amazon リソースネーム (ARN) が必要になります。IAM コンソールの [Roles] (ロール) ページでロール名 (lambda-apigateway-role) を選択し、[Summary] (概要) ページに表示されている [Role ARN] (ロールの ARN) をコピーします。

関数を作成する



以下のコード例は、作成される DynamoDB テーブルで実行する操作と、いくつかのペイロードデータを指定する、API Gateway からのイベント入力を受け取ります。関数が受け取るパラメータが有効な場合、リクエストされた操作をテーブルで実行します。

Node.js

Example index.mjs

```
console.log('Loading function');

import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-west-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);
```

```
// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *              to perform the operation on
 */
export const handler = async (event, context) => {

     const operation = event.operation;

     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;

        switch (operation) {
          case 'create':
               await ddbDocClient.send(new PutCommand(event.payload));
               break;
          case 'read':
               var table_item = await ddbDocClient.send(new
 GetCommand(event.payload));
               console.log(table_item);
               break;
          case 'update':
               await ddbDocClient.send(new UpdateCommand(event.payload));
               break;
          case 'delete':
               await ddbDocClient.send(new DeleteCommand(event.payload));
               break;
          default:
            return ('Unknown operation: ${operation}');
          }
    }
};
```

> **Note**
>
> この例では、DynamoDB テーブルの名前が関数コード内の変数として定義されます。実際のアプリケーションでは、このパラメータを環境変数として渡し、テーブル名をハードコーディングしないことがベストプラクティスです。詳細については、「AWS Lambda 環境変数の使用」を参照してください。

関数を作成する

1. コード例を `index.mjs` という名前のファイルとして保存し、必要な場合は、コードで指定されている AWS リージョンを編集します。コード内で指定されているリージョンは、このチュートリアルの後半で DynamoDB テーブルを作成するリージョンと同じものにする必要があります。
2. 以下の `zip` コマンドを使用して、デプロイパッケージを作成します。

```
zip function.zip index.mjs
```

3. `create-function` AWS CLI コマンドを使用して、Lambda 関数を作成します。`role` パラメータには、先ほどコピーした実行ロールの Amazon リソースネーム (ARN) を入力します。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler index.handler \
--runtime nodejs20.x \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

Python 3

Example LambdaFunctionOverHttps.py

```
import boto3
import json

# define the DynamoDB table that Lambda will connect to
tableName = "lambda-apigateway"
```

```
# create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(tableName)

print('Loading function')

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:

      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
                 operation being performed
    '''

    # define the functions used to perform the CRUD operations
    def ddb_create(x):
        dynamo.put_item(**x)

    def ddb_read(x):
        dynamo.get_item(**x)

    def ddb_update(x):
        dynamo.update_item(**x)

    def ddb_delete(x):
        dynamo.delete_item(**x)

    def echo(x):
        return x

    operation = event['operation']

    operations = {
        'create': ddb_create,
        'read': ddb_read,
        'update': ddb_update,
        'delete': ddb_delete,
        'echo': echo,
    }

    if operation in operations:
        return operations[operation](event.get('payload'))
    else:
        raise ValueError('Unrecognized operation "{}"'.format(operation))
```

> **Note**
>
> この例では、DynamoDB テーブルの名前が関数コード内の変数として定義されます。実際のアプリケーションでは、このパラメータを環境変数として渡し、テーブル名をハードコーディングしないことがベストプラクティスです。詳細については、「AWS Lambda 環境変数の使用」を参照してください。

関数を作成する

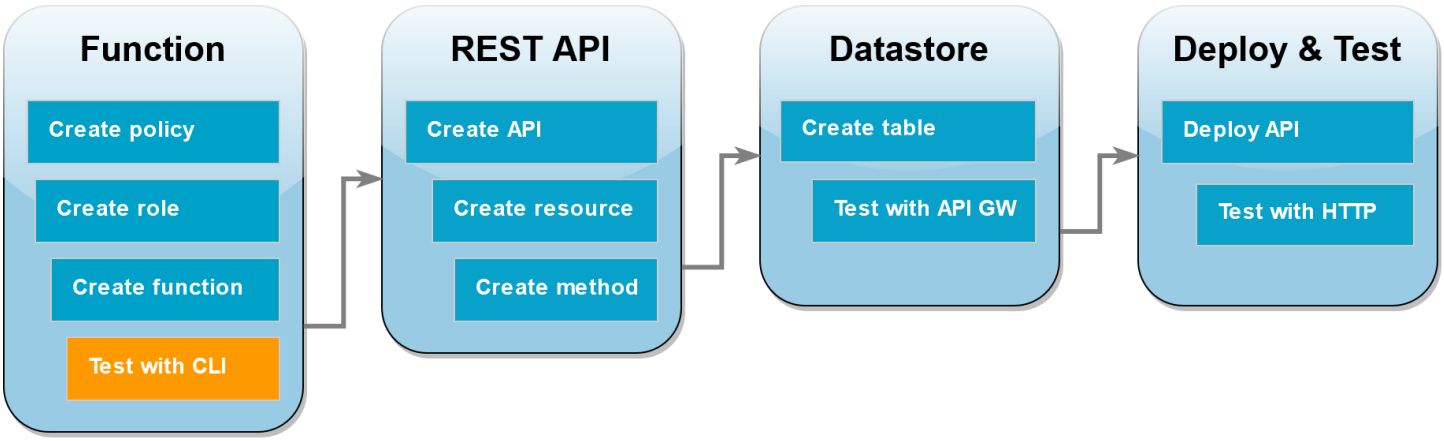
1. サンプルコードを `LambdaFunctionOverHttps.py` のファイル名で保存します。
2. 以下の `zip` コマンドを使用して、デプロイパッケージを作成します。

```
zip function.zip LambdaFunctionOverHttps.py
```

3. `create-function` AWS CLI コマンドを使用して、Lambda 関数を作成します。`role` パラメータには、先ほどコピーした実行ロールの Amazon リソースネーム (ARN) を入力します。

```
aws lambda create-function \
--function-name LambdaFunctionOverHttps \
--zip-file fileb://function.zip \
--handler LambdaFunctionOverHttps.lambda_handler \
--runtime python3.12 \
--role arn:aws:iam::123456789012:role/service-role/lambda-apigateway-role
```

AWS CLI を使用して関数を呼び出す



関数を API Gateway と統合する前に、関数が正常にデプロイされたことを確認します。API Gateway API が Lambda に送信するパラメータが含まれるテストイベントを作成し、AWS CLI `invoke` コマンドを使用して関数を実行します。

AWS CLI を使用して Lambda 関数を呼び出す

1. 次の JSON をファイル名 `input.txt` で保存します。

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

2. 次の `invoke` AWS CLI コマンドを実行します。

```
aws lambda invoke \
--function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt \
--cli-binary-format raw-in-base64-out
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、`aws configure set cli-binary-format raw-in-base64-out` を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

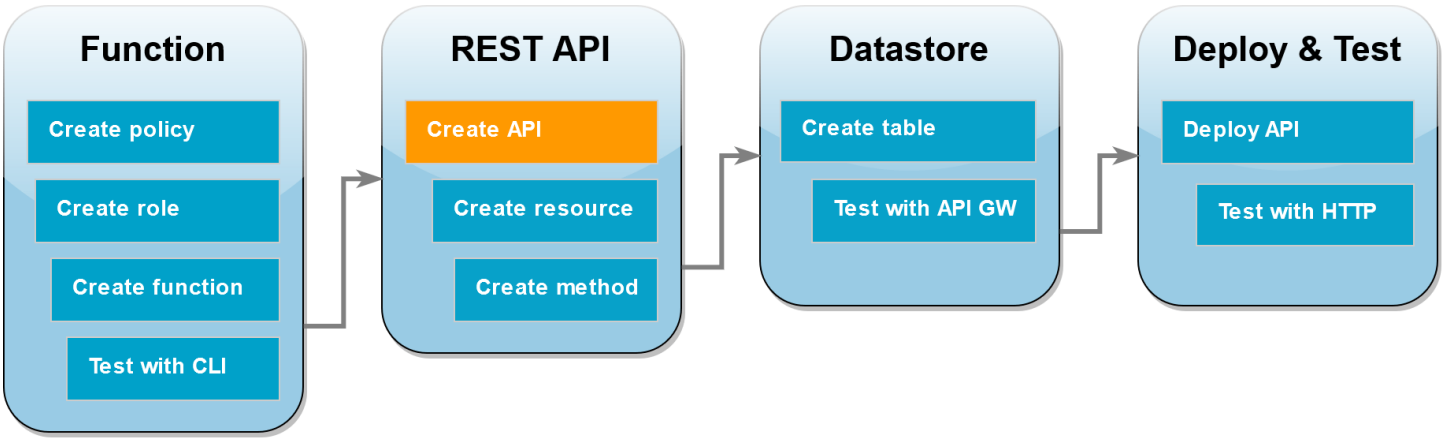
以下のようなレスポンスが表示されます。

```
{
"StatusCode": 200,
"ExecutedVersion": "LATEST"
}
```

3. JSON テストイベントで指定した `echo` 操作を関数が実行したことを確認します。`outputfile.txt` ファイルを調べて、以下が含まれていることを確認します。

```
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

API Gateway を使用して REST API を作成する

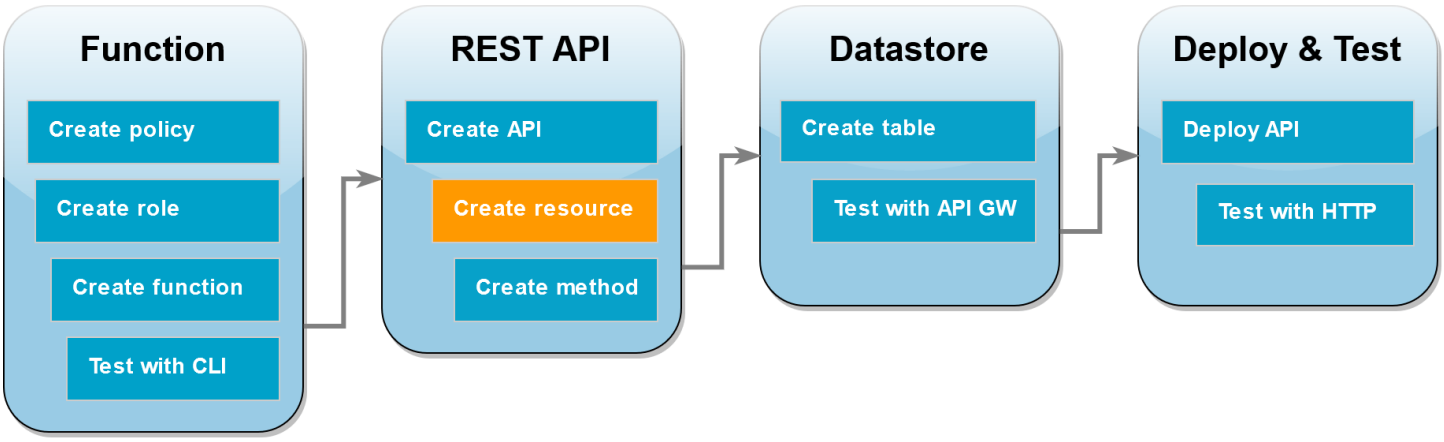


このステップでは、Lambda 関数を呼び出すために使用する API Gateway REST API を作成します。

API を作成するには

1. API Gateway コンソールを開きます。
2. [Create API] を選択します。
3. [REST API] ボックスで、[構築] を選択します。
4. [API の詳細] で [新しい API] を選択したままにし、[API 名] には **`DynamoDBOperations`** と入力します。
5. API の作成 を選択します。

REST API でリソースを作成する

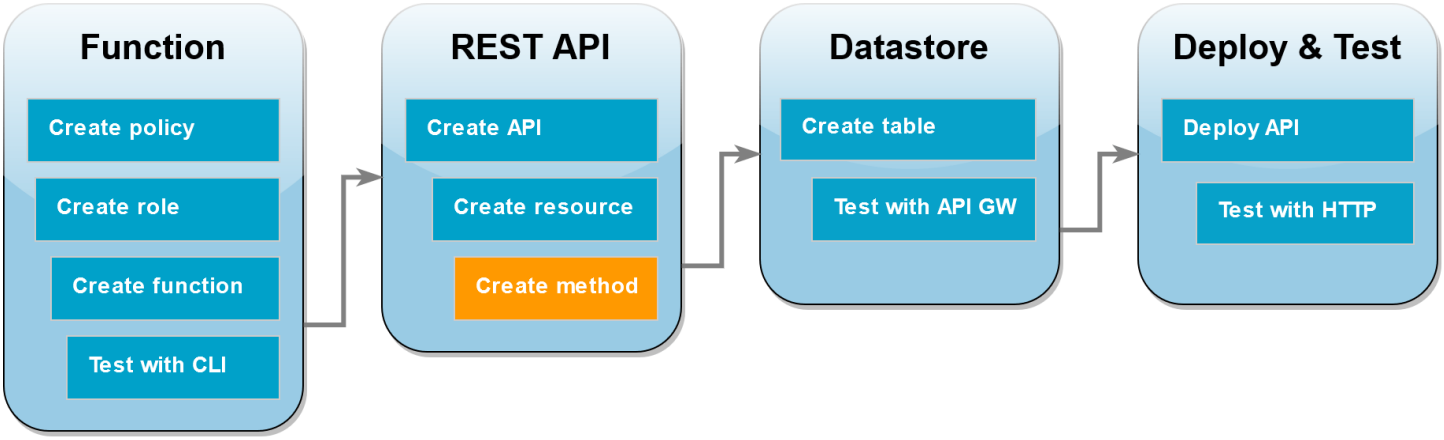


API に HTTP メソッドを追加するには、まずそのメソッドが操作を実行するリソースを作成する必要があります。ここでは、DynamoDB テーブルを管理するためのリソースを作成します。

リソースを作成する

1. API Gateway コンソールの API の [リソース] ページで、[リソースの作成] をクリックします。
2. [リソースの詳細] の [リソース名] に、**DynamoDBManager** と入力します。
3. [リソースの作成] を選択します。

HTTP POST メソッドを作成する



このステップでは、DynamoDBManager リソースのためのメソッド (POST) を作成します。この POST メソッドを Lambda 関数にリンクして、メソッドが HHTTP リクエストを受け取るときに、API Gateway が Lambda 関数を呼び出すようにします。

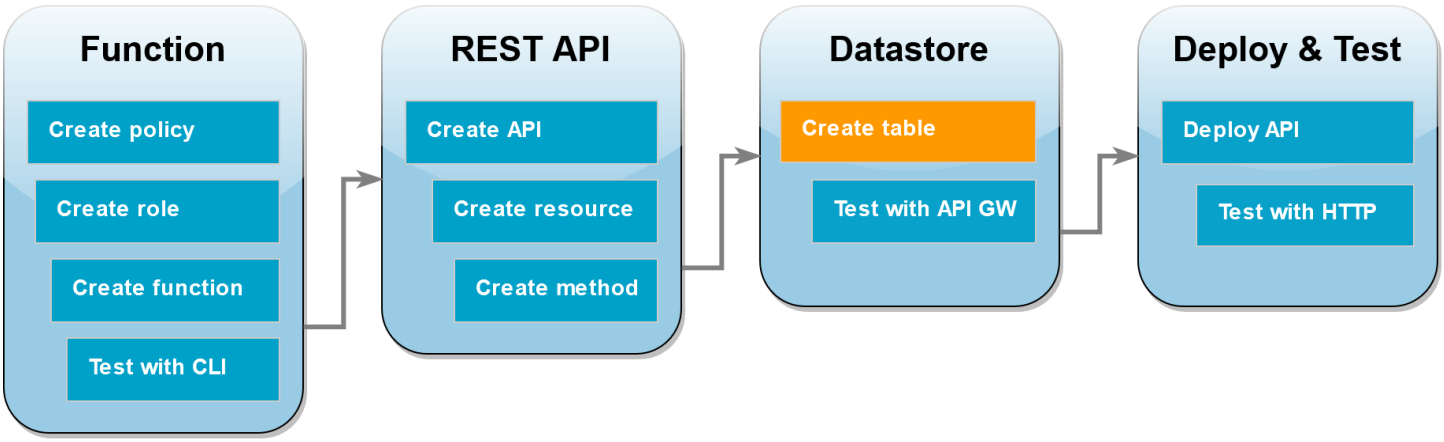
> **Note**
>
> このチュートリアルの目的上、DynamoDB テーブルですべての操作を実行する単一の Lambda 関数を呼び出すために、1 つの HTTP メソッド (POST) が使用されます。実際のアプリケーションでは、操作ごとに異なる Lambda 関数と HTTP メソッドを使用することがベストプラクティスです。詳細については、Serverless Land の「Lambda モノリス」を参照してください。

POST メソッドを作成する

1. API の [リソース] ページで、/DynamoDBManager リソースが強調表示されていることを確認します。次に、[メソッド] ペインで [メソッドの作成] をクリックします。
2. [メソッドタイプ] で、[POST] を選択します。
3. [統合タイプ] で、[Lambda 関数] を選択します。

4. [Lambda 関数] で、使用する関数 (LambdaFunctionOverHttps) の Amazon リソースネーム (ARN)を選択します。
5. [メソッドの作成] を選択します。

DynamoDB テーブルを作成する

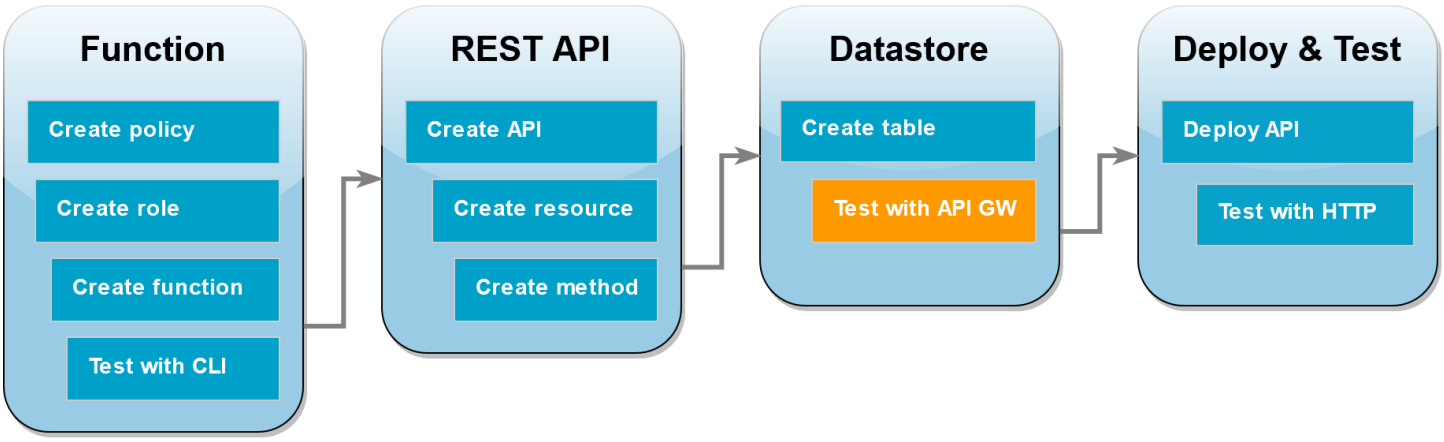


Lambda 関数が CRUD 操作を実行する空の DynamoDB テーブルを作成します。

DynamoDB テーブルを作成する

1. DynamoDB コンソールで [Tables (テーブル)] ページを開きます。
2. [Create table (テーブルの作成)] を選択します。
3. [テーブルの詳細] で、次の操作を行います。
   1. [テーブル名] に「**lambda-apigateway**」と入力します。
   2. [パーティションキー] に「**id**」と入力し、データ型を [文字列] のままにします。
4. [Table settings] (テーブル設定) では、[Default settings] (デフォルト設定) をそのまま使用します。
5. [Create table (テーブルの作成)] を選択します。

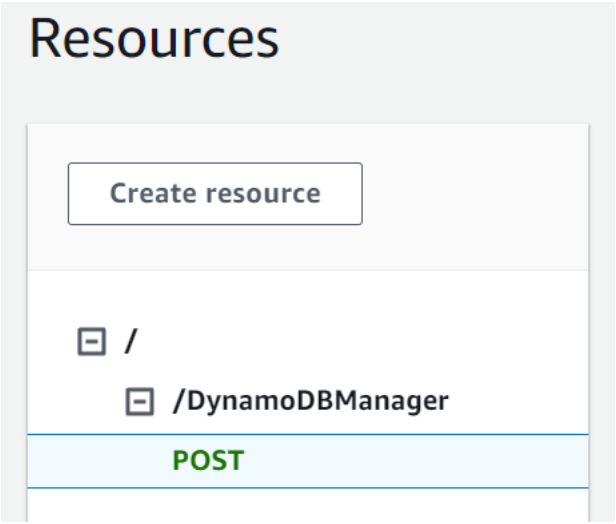
API Gateway、Lambda、および DynamoDB の統合をテストする



これで、API Gateway API メソッドの Lambda 関数および DynamoDB テーブルとの統合をテストするための準備が整いました。API Gateway コンソールで、コンソールのテスト機能を使用して POST メソッドにリクエストを直接送信します。このステップでは、まず `create` 操作を使用して DynamoDB テーブルに新しい項目を追加し、次に `update` 操作を使用してその項目を変更します。

テスト 1 DynamoDB テーブルで新しい項目を作成する

1. API Gateway コンソールで、API (`DynamoDBOperations`) を選択します。
2. `DynamoDBManager` リソースの [POST] メソッドを選択します。



3. [テスト] タブを選択します。タブを表示するには、右矢印ボタンを選択する必要がある場合があります。
4. [テストメソッド] では、[クエリ文字列] と [ヘッダー] を空白のままにします。[リクエスト本文] に以下の JSON を貼り付けます。

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. [テスト] を選択します。

   テストが完了したときに表示される結果には、200 というステータスが表示されるはずです。このステータスコードは、create 操作が正常に実行されたことを示します。

   それを確認するため、DynamoDB テーブルに新しい項目が含まれていることをチェックします。

6. DynamoDB コンソールで [Tables] (テーブル) ページを開き、lambda-apigateway テーブルを選択します。
7. [Explore table items] (テーブルアイテムの探索) を選択します。[Items returned] (返された項目) ペインに、[id] 1234ABCD と [number] (番号) 5 がある 1 つの項目が表示されるはずです。

テスト 2 DynamoDB テーブルの項目を更新する

1. API Gateway コンソールで POST メソッドの [テスト] タブに戻ります。
2. [テストメソッド] では、[クエリ文字列] と [ヘッダー] を空白のままにします。[リクエスト本文] に以下の JSON を貼り付けます。

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "AttributeUpdates": {
            "number": {
                "Value": 10
            }
        }
```

```
    }
}
```

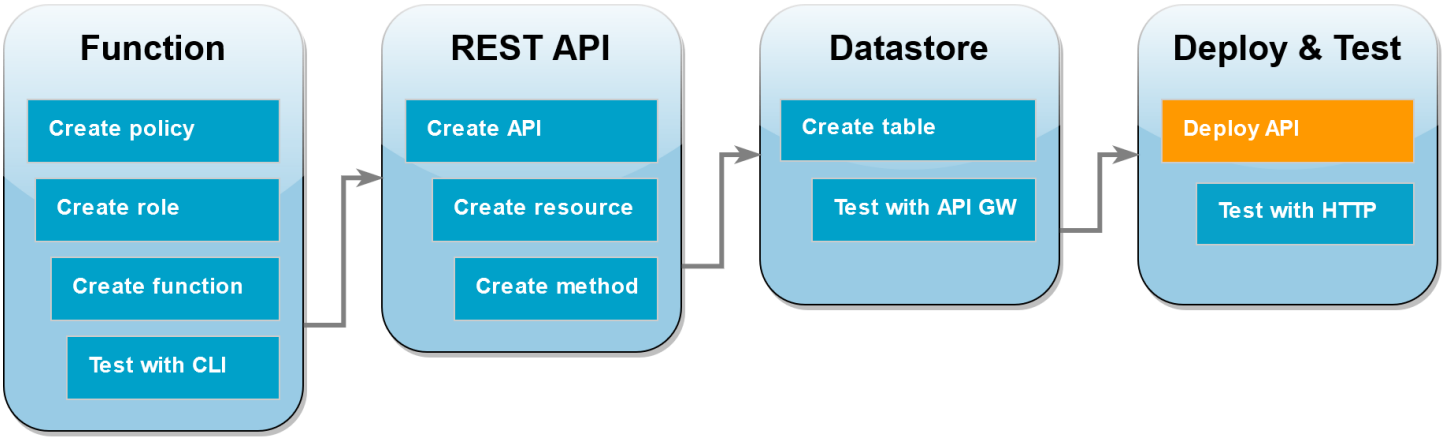
3. [テスト] を選択します。

   テストが完了したときに表示される結果には、200 というステータスが表示されるはずです。このステータスコードは、update 操作が正常に実行されたことを示します。

   それを確認するため、DynamoDB テーブルの項目が変更されていることをチェックします。

4. DynamoDB コンソールで [Tables] (テーブル) ページを開き、lambda-apigateway テーブルを選択します。
5. [Explore table items] (テーブルアイテムの探索) を選択します。[Items returned] (返された項目) ペインに、[id] 1234ABCD と [number] (番号) 10 がある 1 つの項目が表示されるはずです。

API をデプロイする

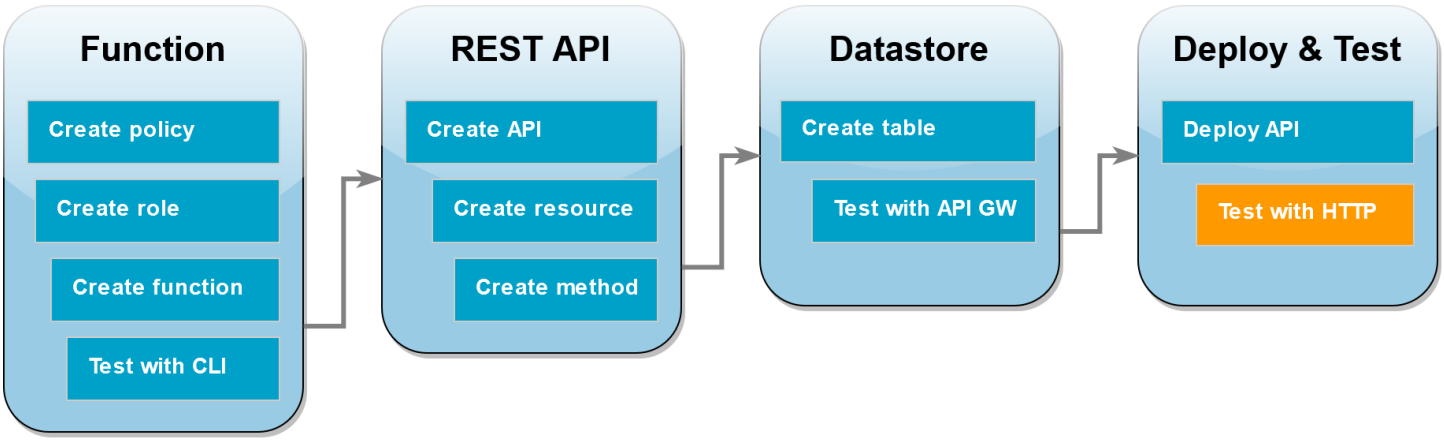


クライアントが API を呼び出すには、デプロイと関連するステージを作成する必要があります。ステージは、メソッドと統合が含まれる API のスナップショットを表します。

API をデプロイする

1. API Gateway コンソールの [API] ページを開き、DynamoDBOperations API を選択します。
2. API の [リソース] ページで [API のデプロイ] をクリックします。
3. [ステージ] で [*新しいステージ*] をクリックし、[ステージ名] には **test** を入力します。
4. [デプロイ] を選択します。
5. [ステージの詳細] ペインで [URL を呼び出す] をコピーします。これは、HTTP リクエストを使用して関数を呼び出すために、次のステップで使用します。

HTTP リクエストを使用して関数を呼び出すために curl を使用する



これで、API に HTTP リクエストを発行することで Lambda 関数を呼び出せるようになりました。このステップでは、DynamoDB テーブルに新しい項目を作成してから、それを削除します。

curl を使用して Lambda 関数を呼び出す

1. 前のステップでコピーした呼び出し URL を使用して、以下の `curl` コマンドを実行します。`-d` (data) オプションと共に curl を使用するときは、自動的に HTTP POST メソッドが使用されます。

   ```
   curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
   ```

2. 作成操作が正常に実行されたことを確認するには、以下の手順を実行します。

   1. DynamoDB コンソールで [Tables] (テーブル) ページを開き、`lambda-apigateway` テーブルを選択します。
   2. [Explore table items] (テーブルアイテムの探索) を選択します。[Items returned] (返された項目) ペインに、[id] `5678EFGH` と [number] (番号) `15` がある項目が表示されるはずです。

3. 以下の `curl` コマンドを実行して、先ほど作成した項目を削除します。独自の呼び出し URL を使用してください。

   ```
   curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
   -d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

4. 削除操作が正常に実行されたことを確認します。DynamoDB コンソールの [Explore items] (項目を探索) ページにある [Items returned] (返された項目) ペインで、id `5678EFGH` がある項目がテーブル内にないことを確認します。

リソースをクリーンアップする (オプション)

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[Delete] (削除) を選択します。

API を削除する

1. API Gateway コンソールで API のページを開きます。
2. 作成した API を選択します。
3. [Actions] で、[Delete] を選択します。
4. [削除] を選択します。

DynamoDB テーブルを削除するには

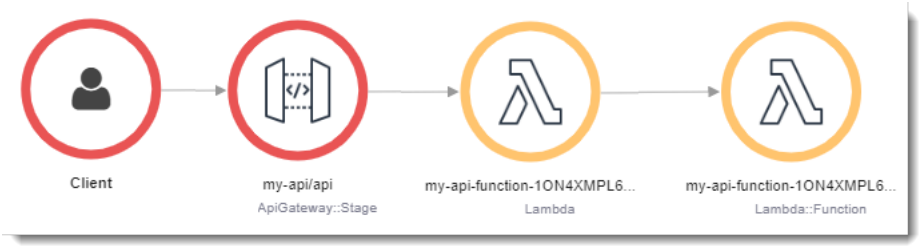
1. DynamoDB コンソールで [Tables (テーブル)] ページを開きます。
2. 作成したテーブルを選択します。
3. [削除] を選択します。
4. テキストボックスに「**delete**」と入力します。
5. [テーブルの削除] を選択します。

API Gateway API を使用した Lambda エラーの処理

API Gateway はすべての呼び出しエラーと関数エラーを内部エラーとして扱います。Lambda API が呼び出しリクエストを拒否した場合、API Gateway は 500 エラーコードを返します。関数が実行されてもエラーが返された場合、または誤った形式でレスポンスが返された場合、API Gateway は 502 を返します。どちらの場合も、API Gateway からのレスポンスの本文は `{"message": "Internal server error"}` です。

> **Note**
>
> API Gateway は、Lambda 呼び出しを再試行しません。Lambda がエラーを返す場合、API Gateway はクライアントにエラーレスポンスを返します。

以下の例では、関数エラーになり API Gateway から 502 が返されたリクエストの X-Ray トレースマップを示しています。クライアントは一般的なエラーメッセージを受け取ります。

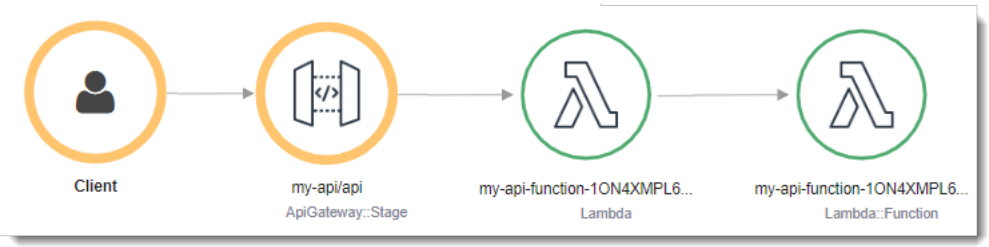


エラーレスポンスをカスタマイズするには、コードでエラーをキャッチし、レスポンスを必要な形式に加工する必要があります。

Example index.js - エラーの形式

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway は、このレスポンスをカスタムステータスコードと本文を含む HTTP エラーに変換します。トレースマップで、関数ノードが緑色なのは、エラーを処理したためです。



AWS Lambdaと使用するAWS Application Composer

AWS Application Composer は、AWS で最新のアプリケーションを設計するためのビジュアルビルダーです。ビジュアルキャンバスで AWS のサービスをドラッグ、グループ化、接続して、アプリケーションアーキテクチャを設計します。Application Composer は、AWS SAM または AWS CloudFormation を使用してデプロイできる設計から Infrastructure as Code (IaC) テンプレートを作成します。

Lambda 関数を Application Composer にエクスポートする

Application Composer の使用を開始するには、Lambda コンソールを使用して既存の Lambda 関数の設定に基づいて新しいプロジェクトを作成します。関数の設定とコードを Application Composer にエクスポートして新しいプロジェクトを作成するには、以下の操作を行います。

1. Lambda コンソールの [関数ページ] を開きます。
2. Application Composer プロジェクトの基礎として使用する関数を選択します。
3. [関数の概要] ペインで、[Application Composer にエクスポート] を選択します。

   関数の設定とコードを Application Composer にエクスポートするには、Lambda でアカウントに Amazon S3 バケットを作成し、このデータを一時的に保存します。
4. ダイアログボックスで [プロジェクトの確認と作成] を選択し、このバケットのデフォルト名をそのまま使用して、関数の設定とコードを Application Composer にエクスポートします。
5. (オプション) Lambda で作成する Amazon S3 バケットに別の名前を選択する場合は、新しい名前を入力して [プロジェクトの確認と作成] を選択します。Amazon S3 バケットの名前は、グローバルに一意で、バケットの命名規則に従ったものである必要があります。
6. プロジェクトファイルと関数ファイルを Application Composer に保存するには、ローカル同期モードをアクティブ化します。

> **Note**
>
> 以前に [Application Composer にエクスポート] 機能を使用し、デフォルト名を使用して Amazon S3 バケットを作成したことがある場合、Lambda はこのバケットがまだ存在していれば再利用できます。既存のバケットを再利用するには、ダイアログボックスのデフォルトのバケット名をそのまま使用してください。

Amazon S3 転送バケット設定

Lambda が関数の設定を転送するために作成する Amazon S3 バケットは、AES 256 暗号化標準を使用してオブジェクトを自動的に暗号化します。また、Lambda はバケット所有者条件を使用するようにバケットを設定して、ユーザーの AWS アカウントだけがバケットにオブジェクトを追加できるようにします。

Lambda は、アップロードされてから 10 日後にオブジェクトを自動的に削除するようにバケットを設定します。ただし、Lambda はバケット自体を自動的に削除しません。AWS アカウントからバケットを削除するには、「バケットの削除」の手順に従います。デフォルトのバケット名には、プレフィックス `lambdasam`、10 桁の英数字の文字列、および関数を作成した AWS リージョンが使用されます。

```
lambdasam-06f22da95b-us-east-1
```

AWS アカウントに追加料金が発生しないように、関数を Application Composer にエクスポートし終えたらすぐに Amazon S3 バケットを削除することをお勧めします。

標準の Amazon S3 の料金が適用されます。

必要なアクセス許可

Lambda と Application Composer の統合機能を使用するには、AWS SAM テンプレートをダウンロードし、関数の設定を Amazon S3 に書き込むための特定の権限が必要です。

AWS SAM テンプレートをダウンロードするには、次の API アクションに対するアクセス権限が必要です。

- GetPolicy
- iam:GetPolicyVersion
- iam:GetRole

- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

IAM ユーザーロールに `AWSLambda_ReadOnlyAccess` AWS マネージドポリシーを追加することで、これらすべてのアクションを使用する権限を付与できます。

Lambda が関数の設定を Amazon S3 に書き込むには、以下の API アクションを使用するアクセス許可が必要です。

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

関数の設定を Application Composer にエクスポートできない場合は、アカウントにこれらのオペレーションに必要なアクセス許可があることを確認してください。必要なアクセス許可を持っていても、関数の設定をエクスポートできない場合は、Amazon S3 へのアクセスを制限している可能性があるリソースベースのポリシーがないか確認します。

その他のリソース

既存の Lambda 関数に基づいて Application Composer でサーバーレスアプリケーションを設計する方法のより詳細なチュートリアルについては、「the section called “Infrastructure as code (IaC) ”」を参照してください。

Application Composer および AWS SAM を使用し、Lambda を使用して完全なサーバーレスアプリケーションを設計およびデプロイするには、「AWS Serverless Patterns Workshop」の「AWS Application Composer tutorial」に従うこともできます。

CloudWatch Logs で Lambda を使用する

Lambda 関数を使用して、Amazon CloudWatch Logs ログストリームのログをモニタリングして分析することができます。ログが作成されたとき、またはログがオプションのパターンに一致した場合に関数が呼び出されるように、1 つ以上のストリームの サブスクリプション を作成します。この関数を使用して通知を送信したり、ログをデータベースまたはストレージに保存したりします。

CloudWatch Logs は、ログデータが含まれたイベントを使用して、関数を非同期的に呼び出します。データフィールドの値は Base64 でエンコードされた .gzip ファイルアーカイブです。

Example CloudWatch Logs メッセージイベント

```
{
  "awslogs": {
    "data":
 "ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUVTU0FHRSIsCiAgICAib3duZXIiOiAiMTIzNDU2Nzg5MDEyIiwKICAgI
  }
}
```

ログデータをデコードおよび圧縮解除すると、次の構造の JSON ドキュメントになります。

Example CloudWatch Logs メッセージデータ (デコード済み)

```
{
    "messageType": "DATA_MESSAGE",
    "owner": "123456789012",
    "logGroup": "/aws/lambda/echo-nodejs",
    "logStream": "2019/03/13/[$LATEST]94fa867e5374431291a7fc14e2f56ae7",
    "subscriptionFilters": [
        "LambdaStream_cloudwatchlogs-node"
    ],
    "logEvents": [
        {
            "id": "34622316099697884706540976068822859012661220141643892546",
            "timestamp": 1552518348220,
            "message": "REPORT RequestId: 6234bffe-149a-b642-81ff-2e8e376d8aff
\tDuration: 46.84 ms\tBilled Duration: 47 ms \tMemory Size: 192 MB\tMax Memory Used: 72
 MB\t\n"
        }
    ]
}
```

AWS Lambdaと使用するAWS CloudFormation

AWS CloudFormation テンプレートでは、Lambda 関数をカスタムリソースのターゲットとして指定できます。パラメータの処理にカスタムリソースを使用したり、設定値を取得したり、スタックのライフサイクルイベント中に他の AWS のサービスを呼び出します。

次の例は、テンプレートの別の場所で定義される関数を呼び出します。

Example - カスタムリソースの定義

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

サービストークンは、スタックの作成、更新あるいは削除時に AWS CloudFormation が呼び出す関数の Amazon リソースネーム (ARN) です。また、FunctionName のように AWS CloudFormation が関数にそのまま渡す追加のプロパティを含めることができます。

AWS CloudFormation では、コールバック URL を含むイベントで非同期的に Lambda 関数を呼び出します。

Example – AWS CloudFormation メッセージイベント

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
```

```
    "ResourceProperties": {
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

関数は、成功あるいは失敗を示すコールバック URL にレスポンスを返す処理を行います 完全なレスポンスの構文については、「カスタムリソースの応答オブジェクト」を参照してください。

Example – AWS CloudFormation カスタムリソース応答

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

AWS CloudFormation は応答の送信を処理する `cfn-response` という名前のライブラリを提供します。テンプレートで関数を定義する場合、ライブラリを名前で使用できます。AWS CloudFormation はこのライブラリを作成する関数のデプロイパッケージに追加します。

カスタムリソースが使用する関数に Elastic Network Interface がアタッチされている場合、次のリソースを **`region`** が関数が属するリージョンにダッシュがない VPC ポリシーに追加します。たとえば、`us-east-1` は `useast1` です。これにより、カスタムリソースは、AWS CloudFormation スタックにシグナルを送り返すコールバック URL に応答できるようになります。

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

次の関数例では、2 番目の関数を呼び出しています。呼び出しが成功すると、この関数は成功レスポンスを AWS CloudFormation に送信し、スタックの更新が継続します。テンプレートは、AWS Serverless Application Model から提供される AWS:: Serverless:: Function リソースタイプを使用します。

Example – カスタムリソース関数

```
Transform: 'AWS::Serverless-2016-10-31'
```

```
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

カスタムリソースが呼び出す関数がテンプレートで定義されていない場合、AWS CloudFormation の cfn-response モジュールから `cfn-response` のリソースコードを取得できます。

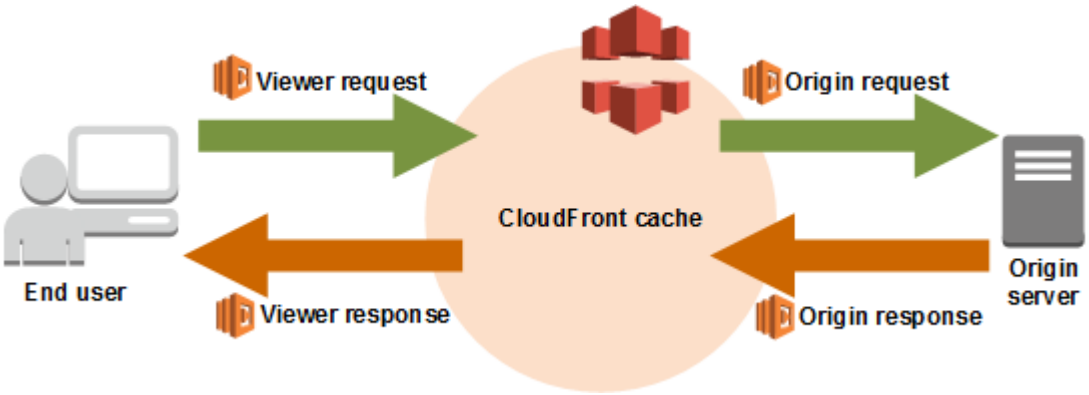
カスタムリソースの詳細については、AWS CloudFormationユーザーガイドのカスタムリソースを参照してください。

CloudFront Lambda@Edge AWS Lambdaで を使用する

Lambda@Edge は、Amazon CloudFront エッジロケーションに Python 関数と Node.js 関数をデプロイAWS Lambdaできる の拡張機能です。Lambda@Edge の一般的なユースケースは、関数を使用して、 CloudFront ディストリビューションがエンドユーザーに配信するコンテンツをカスタマイズすることです。オリジンサーバーではなくビューワーの近くでそれらの関数を呼び出すことで、レイテンシーが大幅に軽減され、ユーザーエクスペリエンスが向上します。

CloudFront ディストリビューションを Lambda@Edge 関数に関連付けると、 は CloudFront エッジロケーションでリクエストとレスポンスを CloudFront インターセプトします。 CloudFront その後、イベントを送信して Lambda 関数を呼び出します。次のイベントが発生したときに、 で Lambda 関数を CloudFront 呼び出すことができます。

- がビューワーからリクエスト CloudFront を受信したとき (ビューワーリクエスト)
- がリクエストをオリジン CloudFront に転送する前に (オリジンリクエスト)
- がオリジンからレスポンス CloudFront を受信する場合 (オリジンレスポンス)
- がビューワーにレスポンスを CloudFront 返す前 (ビューワーレスポンス)



> **Note**
>
> Lambda@Edge は、限定数のランタイムと機能をサポートしています。詳細については、「Amazon CloudFront デベロッパーガイド」の「Lambda 関数の要件と制限」を参照してください。

CloudFront イベントの例を次に示します。

Example CloudFront メッセージイベント

```
{
  "Records": [
    {
      "cf": {
        "config": {
          "distributionId": "EDFDVBD6EXAMPLE"
        },
        "request": {
          "clientIp": "2001:0db8:85a3:0:0:8a2e:0370:7334",
          "method": "GET",
          "uri": "/picture.jpg",
          "headers": {
            "host": [
              {
                "key": "Host",
                "value": "d111111abcdef8.cloudfront.net"
              }
            ],
            "user-agent": [
              {
                "key": "User-Agent",
                "value": "curl/7.51.0"
              }
            ]
          }
        }
      }
    }
  ]
}
```

Lambda@Edge の使用の詳細については、「Lambda@Edge CloudFront での の使用」を参照してください。

AWS Lambdaと使用するAWS CodeCommit

AWS CodeCommit リポジトリのトリガーを作成して、リポジトリのイベントから Lambda 関数を呼び出すことができます。例えば、ブランチまたはタグが作成されたときや既存のブランチに対してプッシュが行われたときに、Lambda 関数を呼び出すことができます。

Example AWS CodeCommit メッセージイベント

```
{
    "Records": [
        {
            "awsRegion": "us-east-2",
            "codecommit": {
                "references": [
                    {
                        "commit": "5e493c6f3067653f3d04eca608b4901eb227078",
                        "ref": "refs/heads/master"
                    }
                ]
            },
            "eventId": "31ade2c7-f889-47c5-a937-1cf99e2790e9",
            "eventName": "ReferenceChanges",
            "eventPartNumber": 1,
            "eventSource": "aws:codecommit",
            "eventSourceARN": "arn:aws:codecommit:us-east-2:123456789012:lambda-
pipeline-repo",
            "eventTime": "2019-03-12T20:58:25.400+0000",
            "eventTotalParts": 1,
            "eventTriggerConfigId": "0d17d6a4-efeb-46f3-b3ab-a63741badeb8",
            "eventTriggerName": "index.handler",
            "eventVersion": "1.0",
            "userIdentityARN": "arn:aws:iam::123456789012:user/intern"
        }
    ]
}
```

詳細については、「AWS CodeCommit リポジトリのトリガーを管理する」を参照してください。

Amazon Cognito との AWS Lambda の使用

Amazon Cognito イベント機能では、Amazon Cognito のイベントへの応答として Lambda 関数を実行できます。Amazon Cognito は、ウェブおよびモバイルアプリの認証、認可、およびユーザー管理機能を提供します。Lambda 関数は、Amazon Cognito での重要なイベントに対応して呼び出すことができます。例えば、Sync Trigger イベントを使用して、データセットが同期されるたびに発行される Lambda 関数を呼び出すことができます。詳細および例の解説については、モバイル開発ブログの Amazon Cognito イベントの概要: 同期トリガーを参照してください。

Example Amazon Cognito メッセージイベント

```
{
  "datasetName": "datasetName",
  "eventType": "SyncTrigger",
  "region": "us-east-1",
  "identityId": "identityId",
  "datasetRecords": {
    "SampleKey2": {
      "newValue": "newValue2",
      "oldValue": "oldValue2",
      "op": "replace"
    },
    "SampleKey1": {
      "newValue": "newValue1",
      "oldValue": "oldValue1",
      "op": "replace"
    }
  },
  "identityPoolId": "identityPoolId",
  "version": 2
}
```

Amazon Cognito のイベントサブスクリプション設定を使用してイベントソースマッピングを設定します。イベントソースマッピングとイベント例については、Amazon Cognito デベロッパーガイドの Amazon Cognito events を参照してください。

Amazon Connect での Lambda の使用

Lambda 関数をを使用して、Amazon Connect からのメッセージを処理することができます。Amazon Connect を使用して、クラウドコンタクトセンターを作成することができます。

Amazon Connect は、リクエストボディおよびメタデータを含むイベントを使用して、Lambda 関数を同期的に呼び出します。

Example Amazon Connect リクエストイベント

```
{
  "Details": {
      "ContactData": {
          "Attributes": {},
          "Channel": "VOICE",
          "ContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "CustomerEndpoint": {
              "Address": "+1234567890",
              "Type": "TELEPHONE_NUMBER"
          },
          "InitialContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "InitiationMethod": "INBOUND | OUTBOUND | TRANSFER | CALLBACK",
          "InstanceARN": "arn:aws:connect:aws-region:1234567890:instance/
c8c0e68d-2200-4265-82c0-XXXXXXXXXX",
          "PreviousContactId": "4a573372-1f28-4e26-b97b-XXXXXXXXXXX",
          "Queue": {
             "ARN": "arn:aws:connect:eu-west-2:111111111111:instance/cccccccc-bbbb-
dddd-eeee-ffffffffffff/queue/aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
             "Name": "PasswordReset"
           },
          "SystemEndpoint": {
              "Address": "+1234567890",
              "Type": "TELEPHONE_NUMBER"
          }
      },
      "Parameters": {
          "sentAttributeKey": "sentAttributeValue"
      }
  },
  "Name": "ContactFlowEvent"
}
```

Lambda で Amazon Connect を使用する方法については、Amazon Connect 管理者ガイドの Invoke Lambda functions を参照してください。

Amazon EC2 で AWS Lambda を使用する

AWS Lambdaを使用して、Amazon Elastic Compute Cloud のライフサイクルイベントを処理し、Amazon EC2 リソースを管理します。Amazon EC2 は、インスタンスの状態が変化したとき、Amazon Elastic Block Store ボリュームのスナップショットが完了したとき、スポットインスタンスの終了が予定されているときなどのライフサイクルイベントにおいて、イベントを Amazon EventBridge (CloudWatch Events) に送信します。これらのイベントを Lambda 関数に転送して処理を行うように、Eventbridge (CloudWatch Events) を設定します。

Eventbridge (CloudWatch Events) は、Amazon EC2 からのイベントドキュメントを使用して、Lambda 関数を非同期的に呼び出します。

Example インスタンスのライフサイクルイベント

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

イベントの設定の詳細については、「Amazon EventBridge スケジューラで Lambda を使用する」を参照してください。Amazon EBS スナップショット通知を処理する関数の例については、「EventBridge Scheduler for Amazon EBS」を参照してください。

AWS SDK を使用して、Amazon EC2 API でインスタンスやその他のリソースを管理することもできます。

アクセス許可

Amazon EC2 のライフサイクルイベントを処理するには、Eventbridge (CloudWatch Events) で関数を呼び出すためのアクセス許可が必要です。このアクセス許可は、関数のリソースベースのポリシーから取得します。Eventbridge (CloudWatch Events) コンソールを使用してイベントトリガーを設定する場合は、コンソールがユーザーに代わってリソースベースのポリシーを更新します。それ以外の場合は、次のようなステートメントを追加します。

Example Amazon EC2 ライフサイクル通知に使用するリソースベースのポリシーステートメント

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

ステートメントを追加するには、`add-permission` AWS CLI コマンドを使用します。

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

関数で AWS SDK を使用して Amazon EC2 リソースを管理する場合は、関数の実行ロールに Amazon EC2 アクセス許可を追加します。

チュートリアル: Amazon VPC の Amazon ElastiCache にアクセスする Lambda 関数の設定

Amazon VPC 内の Amazon ElastiCache にアクセスするように Lambda を設定する方法については、「ElastiCache for Redis ユーザーガイド」の Lambda のチュートリアルを参照してください。

Lambda を使用した Application Load Balancer リクエストの処理

Lambda 関数を使用すると、Application Load Balancer のリクエストを処理することができます。Elastic Load Balancing は、Application Load Balancer のターゲットとして Lambda 関数をサポートしています。パス、またはその他のヘッダー値に基づき、ロードバランサールールを使用して、HTTP リクエストを関数にルーティングします。リクエストを処理して、Lambda 関数の HTTP レスポンスを返します。

Elastic Load Balancing は、リクエストボディおよびメタデータを含むイベントを使用して、Lambda 関数を同期的に呼び出します。

Example Application Load Balancer リクエストイベント

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
```

```
    "isBase64Encoded": False
}
```

関数は、イベントを処理し、応答ドキュメントを JSON でロードバランサーに返します。Elastic Load Balancing は、このドキュメントを HTTP 成功またはエラーの応答に変換し、ユーザーに返します。

Example レスポンスドキュメントの形式

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

Application Load Balancer を関数トリガーとして設定するには、まず、関数を実行するアクセス許可を Elastic Load Balancing に付与します。次に、リクエストを関数にルーティングするターゲットグループを作成し、リクエストをターゲットグループに送信するルールをロードバランサーに追加します。

`add-permission` コマンドを使用して、アクセス許可ステートメントを関数のリソースベースのポリシーに追加します。

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

次のような出力が表示されます。

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

Application Load Balancer のリスナーおよびターゲットグループの設定手順については、Application Load Balancer ユーザーガイドのターゲットとしての Lambda 関数を参照してください。

Lambda で Amazon EFS を使用する

Lambda は Amazon Elastic File System (Amazon EFS) と統合して、Lambda アプリケーションに対するセキュアな共有ファイルシステムアクセスをサポートします。初期化中に、VPC 内のローカルネットワークを介して、NFS プロトコルを使ってファイルシステムをマウントするよう、関数を設定することができます。Lambda は接続を管理し、ファイルシステムとの間で送受信されるすべてのトラフィックを暗号化します。

ファイルシステムと Lambda 関数は同じリージョンに存在している必要があります。あるアカウント内の Lambda 関数は、別のアカウントにファイルシステムをマウントできます。このシナリオでは、関数 VPC とファイルシステム VPC との間で VPC ピア接続を設定します。

> **Note**
>
> ファイルシステムに接続するよう関数を設定する方法については、「」を参照してくださいLambda 関数のファイルシステムアクセスの設定

Amazon EFS は、複数の関数が同時に同じファイルシステムに書き込もうとした場合に、破損を防ぐためにファイルロックをサポートしています。Amazon EFS のロックは NFS v4.1 プロトコルのアドバイザリーロックに従っており、アプリケーションではファイル全体に対するロックとバイト範囲に対するロックの両方を使用できます。

Amazon EFS には、大規模環境で高いパフォーマンスを維持するためのアプリケーションのニーズに基づいて、ファイルシステムをカスタマイズするオプションがあります。考慮すべき主な要因は、接続数、スループット (MiB/秒)、IOPS の 3 つです。

> **クォータ**
>
> ファイルシステムのクォータと制限の詳細については、Amazon Elastic File System ユーザーガイドの Amazon EFS ファイルシステムのクォータ を参照してください。

スケーリング、スループット、および IOPS の問題を回避するには、Amazon EFS が Amazon に送信するメトリクスをモニタリングします CloudWatch。Amazon EFS でのモニタリングの概要については、Amazon Elastic File System ユーザーガイドの Monitoring Amazon EFS を参照してください。

セクション

- 接続
- スループット
- IOPS

接続

Amazon EFS は、ファイルシステムごとに最大 25,000 の接続をサポートします。初期化中、関数の各インスタンスは、呼び出しの間持続するファイルシステムへの単一の接続を作成します。つまり、ファイルシステムに接続されている 1 つまたは複数の関数間では、25,000 の同時接続数に到達する可能性があります。関数が作成する接続数を制限するには、予約済み同時実行を使用します。

ただし、関数のコードまたは設定を大規模に変更すると、関数インスタンスの数が、現在の同時実行数を超えて一時的に増加します。Lambda は、新しいリクエストを処理するために新しいインスタンスをプロビジョニングしますが、古いインスタンスがファイルシステムへの接続を閉じるまでに多少の遅延が生じます。デプロイ中に最大接続数の制限に達するのを回避するには、ローリングデプロイを使用します。ローリングデプロイでは、変更するたびにトラフィックを新しいバージョンに徐々に移行します。

Amazon EC2 などの他のサービスから同じファイルシステムに接続する場合は、Amazon EFS での接続のスケーリング動作にも注意する必要があります。ファイルシステムでは、バーストで最大 3,000 の接続の作成がサポートされ、その後は 1 分あたり 500 の新しい接続がサポートされます。

接続のアラームをモニタリングしてトリガーするには、`ClientConnections` メトリクスを使用します。

スループット

大規模環境では、ファイルシステムの最大スループットを超えることもできます。バーストモード (デフォルト) では、ファイルシステムのベースラインスループットが低く、サイズに応じて直線的にスケールされます。アクティビティのバーストを可能にするために、ファイルシステムにバーストクレジットが付与され、100 MiB/秒以上のスループットを使用できます。クレジットは継続的に蓄積され、読み取りおよび書き込みオペレーションごとに消費されます。ファイルシステムのクレジットが不足すると、ベースラインスループットを超えた読み取りおよび書き込みオペレーションが調整されます。それにより、呼び出しがタイムアウトする可能性があります。

> **Note**
>
> プロビジョニングされた同時実行を使用する場合、関数はアイドル状態でもバーストクレジットを消費できます。プロビジョニングされた同時実行では、関数のインスタンスは呼び出される前に Lambda によって初期化され、数時間ごとにリサイクルされます。初期化中に接続されたファイルシステム上のファイルを使用する場合、このアクティビティではすべてのバーストクレジットを使用できます。

スループットに関するアラームをモニタリングしてトリガーするには、`BurstCreditBalance` メトリクスを使用します。このメトリクスは、関数の同時実行数が低い場合は高くなり、高い場合は低くなります。アクティビティの少ない間にピークトラフィックに対応するために、メトリクスが常に低くなるか蓄積しない場合は、関数の同時実行数の制限や、プロビジョニングされたスループットの有効化が必要になる可能性があります。

IOPS

入出力オペレーション/秒 (IOPS) は、ファイルシステムによって処理される読み取りおよび書き込みオペレーション数の測定値です。汎用モードでは、IOPS は低レイテンシーを実現するために制限されます。これはほとんどのアプリケーションにとって有益です。

汎用モードで IOPS をモニタリングしてアラームを作成するには、`PercentIOLimit` メトリクスを使用します。このメトリクスが 100% に達すると、関数は、読み取りおよび書き込みオペレーションの完了を待機してタイムアウトする可能性があります。

Amazon EventBridge スケジューラで Lambda を使用する

Amazon EventBridge スケジューラはサーバーレススケジューラで、一元化されたマネージドサービスからタスクを作成、実行、管理できます。EventBridge スケジューラでは、繰り返しのパターンに cron やレート式を使ってスケジュールを作成したり、1回限りの呼び出しを設定したりできます。配信の時間枠を柔軟に設定したり、再試行制限を定義したり、未処理のイベントの最大保持時間を設定できます。

Lambda で EventBridge スケジューラを設定すると、EventBridge スケジューラは Lambda 関数を非同期的に呼び出します。このページでは、EventBridge スケジューラを使用してスケジュールに基づき Lambda 関数を呼び出す方法について説明します。

実行ロールを設定する

新しいスケジュールを作成する場合、EventBridge スケジューラにはユーザーに代わってターゲット API オペレーションを呼び出すアクセス許可が必要です。実行ロールを使用して、これらのアクセス許可を EventBridge スケジューラに付与します。スケジュールの実行ロールにアタッチするアクセス許可ポリシーによって、必要なアクセス許可が定義されます。これらのアクセス許可は、EventBridge スケジューラが呼び出すターゲット API によって異なります。

次の手順のように EventBridge スケジューラコンソールを使用してスケジュールを作成すると、EventBridge スケジューラは選択したターゲットに基づき実行ロールを自動的に設定します。EventBridge スケジューラ SDK、AWS CLI、または AWS CloudFormation のいずれかを使用してスケジュールを作成する場合、EventBridge スケジューラがターゲットを呼び出すために必要なアクセス許可を付与する既存の実行ロールが必要です。スケジュールに合わせて実行ロールを手動で設定する方法についての詳細は、「EventBridge スケジューラユーザーガイド」の「実行ロールを設定する」を参照してください。

新しいスケジュールを作成する

コンソールを使用してスケジュールを作成するには

1. Amazon EventBridge スケジューラコンソール (https://console.aws.amazon.com/scheduler/home) を開きます。
2. [スケジュール] ページで、[スケジュールを作成] を選択します。
3. [スケジュールの詳細を指定] ページの [スケジュールの名前と説明] セクションで、次を実行します。

a. [スケジュール名] で、スケジュールの名前を入力します。例えば、**MyTestSchedule** と指定します。

b. (オプション) [説明] で、スケジュールの説明を入力します。例えば、**My first schedule** と指定します。

c. [スケジュールグループ] で、ドロップダウンリストからスケジュールグループを選択します。グループがない場合は、[デフォルト] を選択します。スケジュールグループを作成するには、[独自のスケジュールを作成] を選択します。

スケジュールグループを使用して、スケジュールのグループにタグを追加します。

4. • スケジュールオプションを選択します。

頻度	手順	
[1 回限りのスケジュール] 1 回限りのスケジュールは、指定した日時に 1 回だけターゲットを呼び出します。	[日付と時刻] で、次を実行します。 • 有効な日付を YYYY/MM/DD 形式で入力します。 • タイムスタンプを 24 時間 (hh:mm) 形式で入力します。 • [タイムゾーン] で、タイムゾーンを選択します。	
[繰り返しのスケジュール] 繰り返しのスケジュールは、cron 式またはレート式を使用して指定したレートでターゲットを呼び出します。	a. [スケジュールの種類] では、次のいずれかを実行します。 • Cron 式を使用してスケジュールを定義するには、[cron ベースのスケジュール] を選択して Cron 式を入力します。 • Rate 式を使用してスケジュールを定義する	

頻度	手順	
	には、[rate ベースのスケジュール] を選択して Rate 式を入力します。 cron およびレート式の詳細については、「Amazon EventBridge スケジューラユーザーガイド」の「EventBridge スケジューラのスケジュールタイプ」を参照してください。 b. [フレックスタイムウィンドウ] で、[オフ] を選択してオプションをオフにするか、事前定義された時間枠のいずれかを選択します。例えば、[15 分] を選択し、1 時間に 1 回ターゲットを呼び出す繰り返しのスケジュールを設定した場合、スケジュールは毎時の開始後 15 分以内に実行されます。	

5. (オプション) 前のステップで [定期的なスケジュール] を選択した場合は、[時間枠] セクションで次を実行します。

   a. [タイムゾーン] で、タイムゾーンを選択します。

   b. [開始日時] で、有効な日付を YYYY/MM/DD 形式で入力してから、タイムスタンプを 24 時間 (hh:mm) 形式で指定します。

c. [終了日時] で、有効な日付を YYYY/MM/DD 形式で入力してから、タイムスタンプを 24 時間 (hh:mm) 形式で指定します。

6. [Next] を選択します。
7. [ターゲットを選択] ページで、EventBridge スケジューラが呼び出す AWS API オペレーションを選択します。

   a. [AWS Lambda 呼び出し] を選択します。
   b. [呼び出し] セクションで、関数を選択するか、[新しい Lambda 関数を作成] を選択します。
   c. (オプション) JSON ペイロードを入力します。ペイロードを入力しない場合、EventBridge スケジューラは空のイベントを使用して関数を呼び出します。

8. [Next] を選択します。
9. [Settings] (設定) ページで、以下の操作を行います。

   a. スケジュールをオンにするには、[スケジュールの状態] で [スケジュールを有効にする] をオンに切り替えます。
   b. スケジュールの再試行ポリシーを設定するには、[再試行ポリシーとデッドレターキュー (DLQ)] で次を実行します。

      - [再試行] を切り替えてオンにします。
      - [イベントの最大有効期間] で、EventBridge スケジューラが未処理のイベントを保持しなければならない最大の [時間] と [分] を入力します。
      - 最大 24 時間です。
      - [最大再試行回数] で、ターゲットがエラーを返した場合に EventBridge スケジューラがスケジュールを再試行する最大回数を入力します。

        再試行の最大値は 185 です。

      再試行ポリシーを使用すると、スケジュールがそのターゲットの呼び出しに失敗した場合、EventBridge スケジューラはスケジュールを再実行します。設定されている場合は、スケジュールの最大保持時間と再試行を設定する必要があります。

   c. EventBridge スケジューラが未配信のイベントを保存する場所を選択します。

[デッドレターキュー (DLQ)] オプション	手順
保存しない	[None] を選択します。
スケジュールを作成しようとしている同じ AWS アカウント にイベントを保存する	a. [自分の AWS アカウント の Amazon SQS キューを DLQ として選択] を選択します。 b. Amazon SQS キューの Amazon リソースネーム (ARN) を選択します。
スケジュールを作成しようとしているのとは別の AWS アカウント にイベントを保存する	a. [他の AWS アカウント の Amazon SQS キューを DLQ として指定] を選択します。 b. Amazon SQS キューの Amazon リソースネーム (ARN) を入力します。

d. カスタマーマネージドキーを使用してターゲットの入力を暗号化するには、[暗号化] で [暗号化設定をカスタマイズする (高度)] を選択します。

   このオプションを選択した場合は、既存の KMS キー ARN を入力するか、[AWS KMS key を作成] を選択して AWS KMS コンソールに移動します。EventBridge スケジューラが保管中のデータを暗号化する方法の詳細については、「Amazon EventBridge スケジューラユーザーガイド」の「保管中の暗号化」を参照してください。

e. EventBridge スケジューラに新しい実行ロールを作成させるには、[このスケジュールの新しいロールを作成] を選択します。その後、[ロール名] で名前を入力します。このオプションを選択すると、EventBridge スケジューラは、テンプレート化されたターゲットに必要な許可をロールにアタッチします。

10. [Next] を選択します。
11. [スケジュールの確認と作成] ページで、スケジュールの詳細を確認します。各セクションで、そのステップに戻って詳細を編集するには、[編集] を選択します。
12. [スケジュールを作成] を選択します。

[スケジュール] ページで、新規および既存のスケジュールのリストを表示できます。[ステータス] 列で、新しいスケジュールが [有効] になっていることを確認します。

EventBridge スケジューラが関数を呼び出したことを確認するには、関数の Amazon CloudWatch ログを確認してください。

関連リソース

EventBridge スケジューラに関する詳細については、次を参照してください。

- EventBridge スケジューラユーザーガイド
- EventBridge スキーマ API リファレンス
- EventBridge Scheduler Pricing

で AWS Lambda を使用するAWS IoT

AWS IoT は、インターネットに接続されたデバイス (センサーなど) と AWS クラウドとの安全な通信を提供します。これにより、複数のデバイスからテレメトリデータを収集して保存および分析できます。

デバイスが AWS IoT のサービスとやり取りするための AWS ルールを作成できます。AWS IoTルールエンジンは、メッセージペイロードからデータを選択して他のサービス (Amazon S3、Amazon DynamoDB、AWS Lambda など) に送信するための SQL ベースの言語を提供します。AWS の別のサービスやサードパーティーのサービスを呼び出す場合は、Lambda 関数を呼び出すためのルールを定義します。

着信 IoT メッセージによってルールがトリガーされると、AWS IoT は Lambda 関数を非同期的に呼び出し、IoT メッセージから関数にデータを渡します。

次の例は、温室センサーの湿度値を示しています。row 値と pos 値は、センサーの位置を識別します。このイベント例は、AWS IoT ルールチュートリアルの greenhouse タイプに基づいています。

Example AWS IoT メッセージイベント

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

非同期呼び出しで、関数がエラーを返した場合、Lambda はメッセージをキューに入れ、エラーになった呼び出しを再試行します。関数を設定するには、送信先を使用して、関数が処理できなかったイベントを保持します。

Lambda 関数を呼び出すためのアクセス許可を AWS IoT サービスに付与する必要があります。add-permission コマンドを使用して、アクセス許可ステートメントを関数のリソースベースのポリシーに追加します。

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

次のような出力が表示されます。

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
}
```

AWS IoT で Lambda を使用する方法の詳細については、AWS Lambda ルールの作成を参照してください。

Amazon Data Firehose で AWS Lambda を使用する

Amazon Data Firehose は、ストリーミングデータをキャプチャ、変換し、Apache Flink や Amazon S3 のマネージド サービスなどのダウンストリーム サービスにロードします。Lambda 関数を作成して、データがダウンストリームに送信される前に、カスタマイズされた処理を追加でリクエストすることができます。

Example Amazon Data Firehose メッセージイベント


```
{
  "invocationId": "invoked123",
  "deliveryStreamArn": "aws:lambda:events",
  "region": "us-west-2",
  "records": [
    {
      "data": "SGVsbG8gV29ybGQ=",
      "recordId": "record1",
      "approximateArrivalTimestamp": 1510772160000,
      "kinesisRecordMetadata": {
        "shardId": "shardId-000000000000",
        "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c317a",
        "approximateArrivalTimestamp": "2012-04-23T18:25:43.511Z",
        "sequenceNumber": "49546986683135544286507457936321625675700192471156785154",
        "subsequenceNumber": ""
      }
    },
    {
      "data": "SGVsbG8gV29ybGQ=",
      "recordId": "record2",
      "approximateArrivalTimestamp": 151077216000,
      "kinesisRecordMetadata": {
        "shardId": "shardId-000000000001",
        "partitionKey": "4d1ad2b9-24f8-4b9d-a088-76e9947c318a",
        "approximateArrivalTimestamp": "2012-04-23T19:25:43.511Z",
        "sequenceNumber": "49546986683135544286507457936321625675700192471156785155",
        "subsequenceNumber": ""
      }
    }
  ]
}
```

詳細については、Kinesis Data Firehose デベロッパーガイドの「Amazon Data Firehose data transformation」を参照してください。

Amazon Lex で AWS Lambda を使用する

Amazon Lex を使用すると、会話ボットをアプリケーションに統合できます。Amazon Lex ボットは、ユーザーに会話型インターフェイスを提供します。Amazon Lex では、Lambda との統合が事前に構築されています。したがって、Amazon Lex ボットで Lambda 関数を使用することができます。

Amazon Lex ボットを設定するときに、検証、応答、またはその両方を行う Lambda 関数を指定します。検証の場合、Amazon Lex はユーザーからレスポンスがあった後に Lambda 関数を呼び出します。Lambda 関数はこのレスポンスを検証し、必要に応じて、ユーザーに修正フィードバックを提供します。応答の場合、Amazon Lex は、ボットが必要なすべての情報を正常に収集し、ユーザーから確認を受け取った後に、Lambda 関数を呼び出してユーザーのリクエストに応答します。

Lambda 関数の同時実行数を管理することで、提供する同時ボット会話の最大数を制御できます。関数の同時実行数が最大になると、Amazon Lex API は HTTP 429 ステータスコード (リクエストが多すぎる) を返します。

Lambda 関数が例外をスローした場合、API は HTTP 424 ステータスコード (依存関係で失敗) を返します。

Amazon Lex ボットは Lambda 関数を同期的に呼び出します。イベントパラメータには、ボットに関する情報と会話の各スロットの値が含まれます。イベントフィールドとレスポンスフィールドの定義については、「Amazon Lex デベロッパーガイド」の「Lambda 関数イベントとレスポンスの形式」を参照してください。Amazon Lex メッセージイベントの `invocationSource`パラメータは、Lambda 関数が入力を検証する (DialogCodeHook) か、インテントを達成する () かを示します FulfillmentCodeHook。

Amazon Lex で Lambda を使用する方法のサンプルチュートリアルについては、 Amazon デベロッパーガイドの Exercise 1: Create Amazon Lex bot using a blueprint を参照してください。

ロールとアクセス許可

サービスにリンクされたロールは、関数の実行ロールとして設定する必要があります。Amazon Lex は、事前定義されたアクセス許可を持つ、サービスにリンクされたロールを定義します。コンソールを使用して Amazon Lex ボットを作成すると、サービスにリンクされたロールが自動的に作成されます。AWS CLI でサービスにリンクされたロールを作成するには、`create-service-linked-role` コマンドを使用します。

```
aws iam create-service-linked-role --aws-service-name lex.amazonaws.com
```

このコマンドは以下のロールを作成します。

```
{
  "Role": {
      "AssumeRolePolicyDocument": {
          "Version": "2012-10-17",
          "Statement": [
              {
                  "Action": "sts:AssumeRole",
                  "Effect": "Allow",
                  "Principal": {
                      "Service": "lex.amazonaws.com"
                  }
              }
          ]
      },
      "RoleName": "AWSServiceRoleForLexBots",
      "Path": "/aws-service-role/lex.amazonaws.com/",
      "Arn": "arn:aws:iam::account-id:role/aws-service-role/lex.amazonaws.com/
AWSServiceRoleForLexBots"
}
```

Lambda 関数が他の AWS のサービスを使用する場合、対応するアクセス許可をサービスにリンクされたロールに追加する必要があります。

リソースベースのアクセス許可ポリシーを使用して、Amazon Lex ボットに Lambda 関数の呼び出しを許可します。Amazon Lex コンソールを使用する場合、アクセス許可ポリシーは自動的に作成されます。AWS CLI から、Lambda `add-permission` コマンドを使用してアクセス許可を設定します。

Amazon Lex V2 の場合は、次のコマンドを実行します。ソース ARN で、`us-east-1` を Amazon Lex ボットが存在する AWS リージョン に置き換え、独自の AWS アカウント 番号とボットエイリアスを使用します。

```
aws lambda add-permission \
    --function-name LexCodeHook \
    --statement-id LexInvoke-MyBot \
    --action lambda:InvokeFunction \
    --principal lex.amazonaws.com \
    --source-arn "arn:aws:lex:us-east-1:123456789012:bot-alias/MYBOT/MYBOTALIAS"
```

Amazon Lex V1 を使用して Lambda 関数を呼び出すこともできます。Amazon Lex V1 の場合は、次のコマンドを実行します。ソース ARN で、us-east-1 を Amazon Lex インテントが存在する AWS リージョン に置き換え、独自の AWS アカウント 番号とインテント名を使用します。

```
aws lambda add-permission \
    --function-name LexCodeHook \
    --statement-id LexInvoke-MyIntent \
    --action lambda:InvokeFunction \
    --principal lex.amazonaws.com \
    --source-arn "arn:aws:lex:us-east-1:123456789012 ID:intent:MYINTENT:*"
```

Amazon Lex V1 はメンテナンスされなくなったことに注意してください。Amazon Lex V2 を使用することをお勧めします。

Amazon RDS で AWS Lambda 使用する

Lambda 関数は、Amazon Relational Database Service (Amazon RDS) データベースに直接接続することも、Amazon RDS Proxy 経由で接続することもできます。単純なシナリオでは直接接続が有効で、本番環境ではプロキシの使用が推奨されます。データベースプロキシは、共有されたデータベース接続のプールを管理します。これを使用することで、データベース接続を消耗することなく、関数の同時実行レベルを上げることができます。

データベースへの短時間の接続を頻繁に実行したり、多数のデータベースへの接続を開閉したりする Lambda 関数には、Amazon RDS Proxy を使用することをお勧めします。

関数を設定する

Lambda コンソールでは、Amazon RDS データベースのインスタンスおよびプロキシリソースのプロビジョニングおよび設定することができます。詳細については、[設定] タブの [RDS データベース] を参照してください。または、Amazon RDS コンソールで Lambda 関数への接続を作成して設定することもできます。

- データベースに接続するには、関数をデータベースが実行されているものと同じ Amazon VPC に置く必要があります。
- Amazon RDS データベースは、MySQL、MariaDB、PostgreSQL、または Microsoft SQL Server エンジンで使用できます。
- また、MySQL または PostgreSQL エンジンでは、Aurora DB クラスターも使用できます。
- データベース認証用の Secrets Manager シークレットを用意する必要があります。
- IAM ロールはシークレットを使用するためのアクセス許可を付与する必要があり、信頼ポリシーは Amazon RDS にロールの引き受けを許可する必要があります。
- コンソールを使用して Amazon RDS リソースを設定し、関数に接続する IAM プリンシパルには、次の許可が必要です。

> **Note**
>
> Amazon RDS Proxy の許可は、データベース接続のプールを管理するように Amazon RDS Proxy を設定する場合にのみ必要です。

Example アクセス許可ポリシー

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
```

```
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
        "iam:AttachPolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS は、データベースインスタンスのサイズに基づいたプロキシの料金が時間単位で請求されます。詳細については、「RDS プロキシの料金」をご覧ください。プロキシ接続についての全般的な詳細は、Amazon RDS User Guide の「Using Amazon RDS Proxy」を参照してください。

> **Lambda とAmazon RDS のセットアップ**
>
> Lambda コンソールと Amazon RDS コンソールはどちらも、Lambda と Amazon RDS を接続するために必要なリソースの一部を自動的に設定するサポートを行います。

Lambda 関数での Amazon RDS データベースへの接続

次のコード例は、Amazon RDS データベースに接続する Lambda 関数を実装する方法を示しています。この関数は、シンプルなデータベースリクエストを実行し、結果を返します。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda 関数での Amazon RDS データベースへの接続

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {
```

```
var dbName string = "DatabaseName"
var dbUser string = "DatabaseUser"
var dbHost string = "mysqldb.123456789012.us-east-1.rds.amazonaws.com"
var dbPort int = 3306
var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
var region string = "us-east-1"

cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
 panic("configuration error: " + err.Error())
}

authenticationToken, err := auth.BuildAuthToken(
 context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
if err != nil {
 panic("failed to create authentication token: " + err.Error())
}

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
 dbUser, authenticationToken, dbEndpoint, dbName,
)

db, err := sql.Open("mysql", dsn)
if err != nil {
 panic(err)
}

defer db.Close()

var sum int
err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
if err != nil {
 panic(err)
}
s := fmt.Sprint(sum)
message := fmt.Sprintf("The selected sum is: %s", s)

messageBytes, err := json.Marshal(message)
if err != nil {
 return nil, err
}

messageString := string(messageBytes)
```

```
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

JavaScript

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Javascript を使用した Lambda 関数での Amazon RDS データベースへの接続

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,
```

```
  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Amazon RDS からのイベント通知を処理する

Lambda を使用して、Amazon RDS データベースからのイベント通知を処理できます。Amazon RDS は、Amazon Simple Notification Service (Amazon SNS) トピックに通知を送信します。このトピックは Lambda 関数を呼び出すように設定することができます。Amazon SNS は、Amazon RDS からのメッセージを独自のイベントドキュメントにラップし、これを関数に送信します。

通知を送信するように Amazon RDS データベースを設定する方法の詳細については、「Using Amazon RDS event notifications」を参照してください。

Example Amazon SNS イベントでの Amazon RDS メッセージ

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
```

```
    }
```

Lambda と Amazon RDS チュートリアル

- Lambda 関数を使用して Amazon RDS データベースにアクセスする — Amazon RDS ユーザーガイドで、Lambda 関数を使用して、Amazon RDS Proxy 経由で Amazon RDS データベースにデータを書き込む方法を学びます。Lambda 関数は、メッセージが追加されるたびに Amazon SQS キューからレコードを読み取り、データベース内のテーブルに新しい項目を書き込みます。

Lambda を使用した Amazon S3 イベント通知の処理

Amazon Simple Storage Service からのイベント通知は、Lambda を使用して処理することができます。Amazon S3 は、オブジェクトを作成または削除するときに、イベントを Lambda 関数に送信できます。バケットの通知設定を構成し、関数のリソースベースのアクセス許可ポリシーで関数を呼び出すためのアクセス許可を Amazon S3 に付与します。

> **Warning**
>
> Lambda 関数で使用するバケットが、その関数をトリガーするのと同じバケットである場合、関数はループで実行される可能性があります。たとえば、オブジェクトがアップロードされるたびにバケットで関数をトリガーし、その関数によってオブジェクトがバケットにアップロードされると、その関数によって間接的にその関数自体がトリガーされます。これを回避するには、2 つのバケットを使用するか、受信オブジェクトで使用されるプレフィックスにのみ適用されるようにトリガーを設定します。

Amazon S3 は、オブジェクトに関する詳細を含むイベントで非同期に関数を呼び出します。次の例は、デプロイパッケージが Amazon S3 にアップロードされたときに Amazon S3 から送信されたイベントを示しています。

Example Amazon S3 の通知イベント

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
```

```
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

関数を呼び出すには、Amazon S3 には、関数のリソースベースのポリシーによるアクセス許可が必要です。Lambda コンソールで Amazon S3 トリガーを設定すると、バケット名とアカウント ID が一致した場合に Amazon S3 で関数を呼び出せるように、コンソールでリソースベースのポリシーが変更されます。Amazon S3 の通知を設定する場合は、Lambda API を使用してこのポリシーを更新します。また、Lambda API を使用して、別のアカウントにアクセス許可を付与したり、指定されたエイリアスへのアクセス許可を制限したりできます。

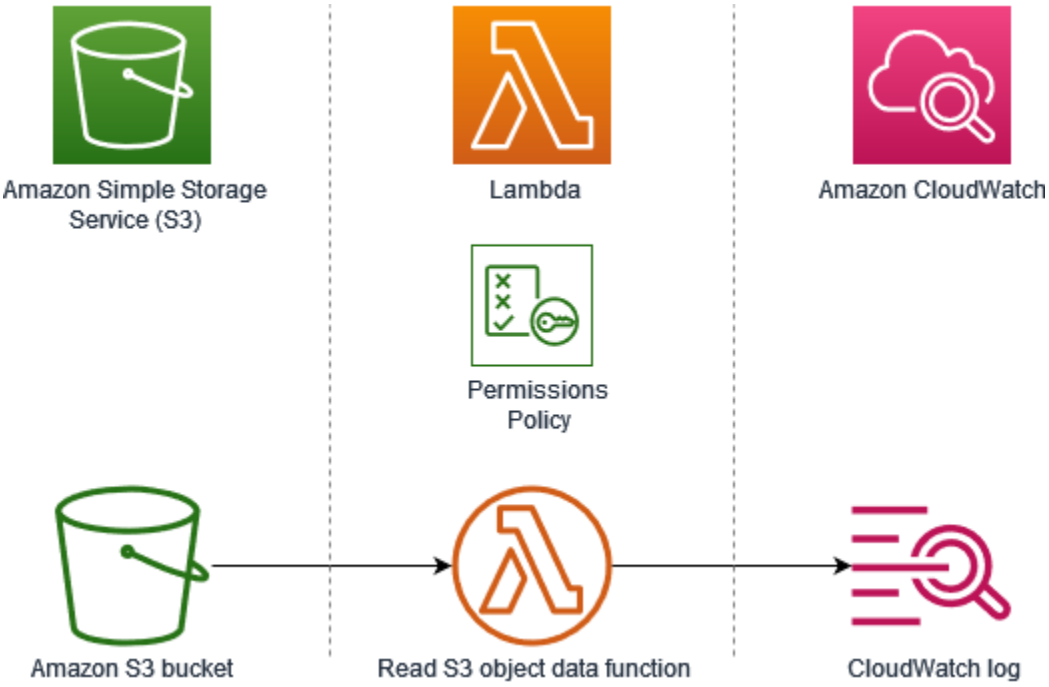
関数で AWS SDK を使用して Amazon S3 リソースを管理する場合は、その実行ロールにも Amazon S3 のアクセス許可が必要です。

トピック

- チュートリアル: Amazon S3 トリガーを使用して Lambda 関数を呼び出す
- チュートリアル: Amazon S3 トリガーを使用してサムネイル画像を作成する

チュートリアル: Amazon S3 トリガーを使用して Lambda 関数を呼び出す

このチュートリアルでは、コンソールを使用して Lambda 関数を作成し、Amazon Simple Storage Service (Amazon S3) バケットのトリガーを設定します。Amazon S3 バケットにオブジェクトを追加するたびに関数を実行し、Amazon CloudWatch Logs にオブジェクトタイプを出力します。



このチュートリアルでは、次の方法を示します。

1. Amazon S3 バケットを作成する。
2. Amazon S3 バケット内のオブジェクトのオブジェクトタイプを返す Lambda 関数を作成します。
3. オブジェクトがバケットにアップロードされたときに関数を呼び出す Lambda トリガーを設定します。
4. 最初にダミーイベントを使用して関数をテストし、次にトリガーを使用してテストします。

これらのステップを完了することにより、Amazon S3 バケットにオブジェクトが追加されたり、Amazon S3 バケットから削除されたりするたびに実行されるように Lambda 関数を設定する方法を学びます。AWS Management Console のみを使って、このチュートリアルを完了できます。

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [アカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウント のメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

  IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

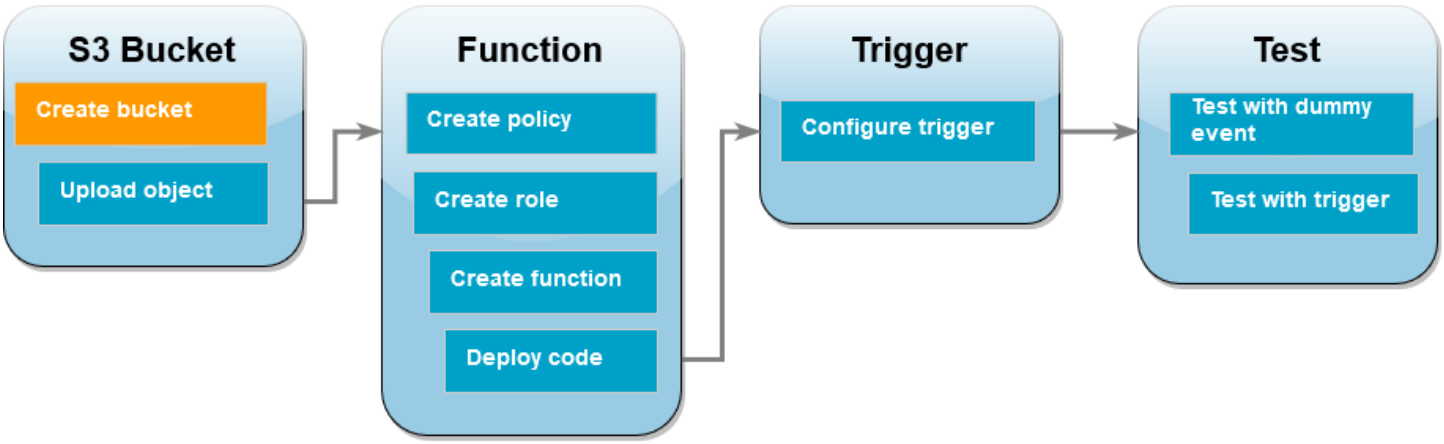
1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

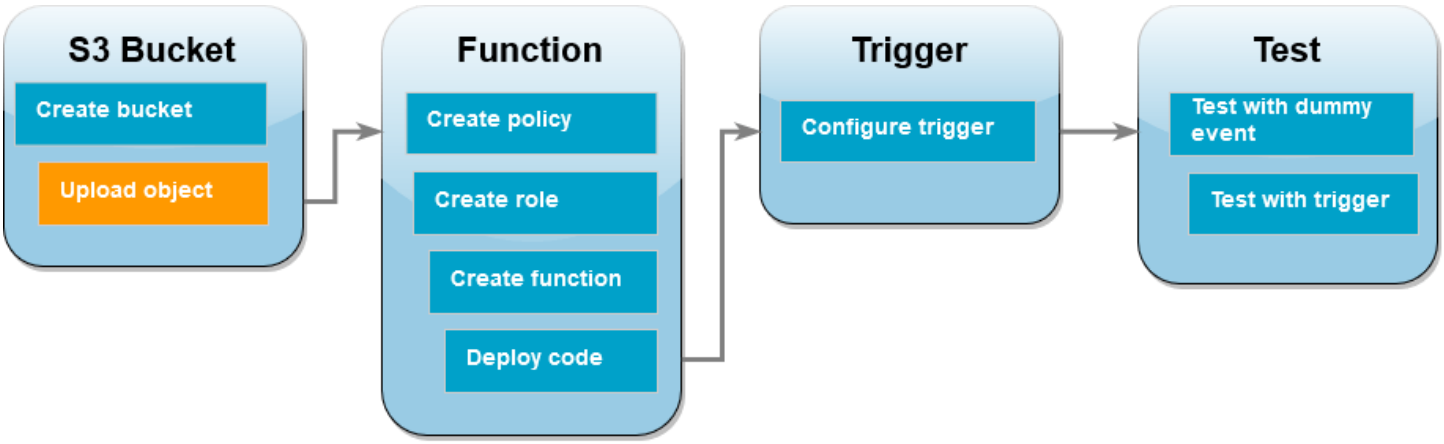
Amazon S3 バケットを作成する



Amazon S3 バケットを作成するには

1. Amazon S3 コンソールを開き、バケットページを選択します。
2. [バケットを作成する] を選択します。
3. [全般設定] で、次の操作を行います。
   a. [バケット名] には、Amazon S3 バケットの命名規則を満たすグローバルに一意な名前を入力します。バケット名は、小文字、数字、ドット (.)、およびハイフン (-) のみで構成できます。
   b. [AWS リージョン] で、リージョンを選択します。チュートリアルの後半では、同じリージョンで Lambda 関数を作成する必要があります。
4. 他のすべてのオプションはデフォルト設定値のままにしておき、[バケットの作成] を選択します。

テストオブジェクトをバケットにアップロードする

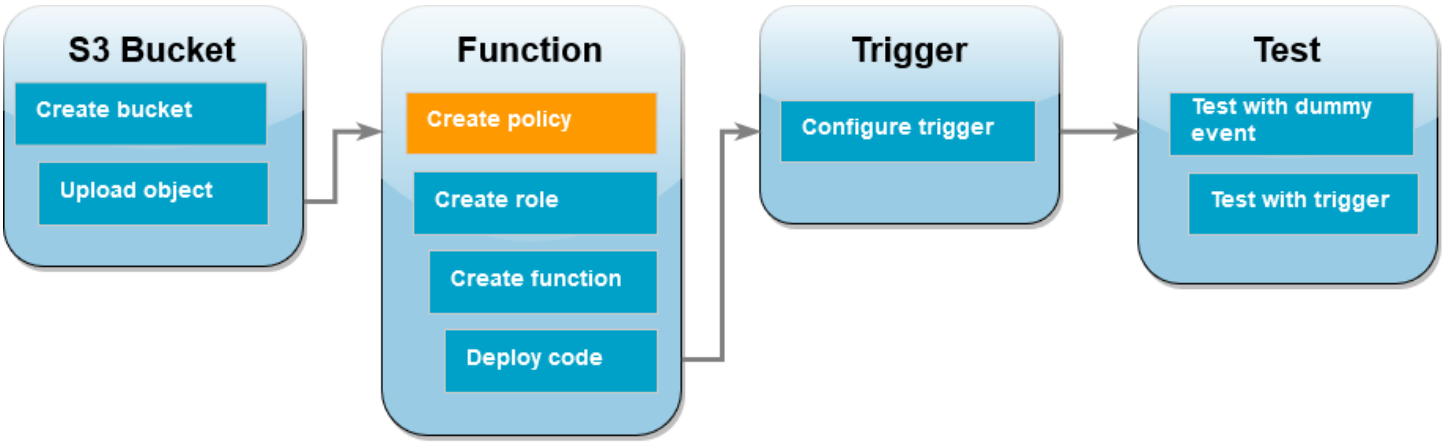


テストオブジェクトをアップロードするには

1. Amazon S3 コンソールのバケットページを開き、前のステップで作成したバケットを選択します。
2. [アップロード] を選択します。
3. [ファイルを追加] を選択し、アップロードするファイルを選択します。任意のファイルを選択できます (例えば、`HappyFace.jpg`)。
4. [開く]、[アップロード] の順に選択します。

チュートリアルの後半では、このオブジェクトを使用して Lambda 関数をテストします。

許可ポリシーを作成する



Lambda が Amazon S3 バケットからオブジェクトを取得し、Amazon CloudWatch Logs に書き込めるようにする許可ポリシーを作成します。

ポリシーを作成するには

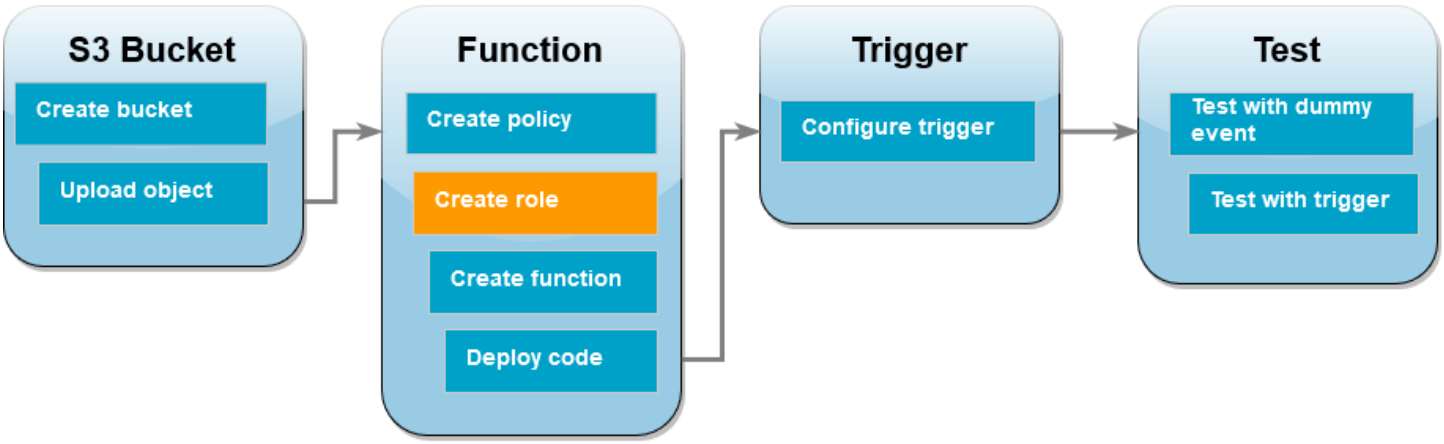
1. IAM コンソールのポリシーページを開きます。
2. [ポリシーの作成] を選択します。
3. [JSON] タブを選択して、次のカスタムポリシーを JSON エディタに貼り付けます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. [次へ: タグ] を選択します。
5. [次へ: レビュー] を選択します。
6. [ポリシーの確認] でポリシーの [名前] に「**s3-trigger-tutorial**」と入力します。
7. [ポリシーの作成] を選択します。

実行ロールを作成する



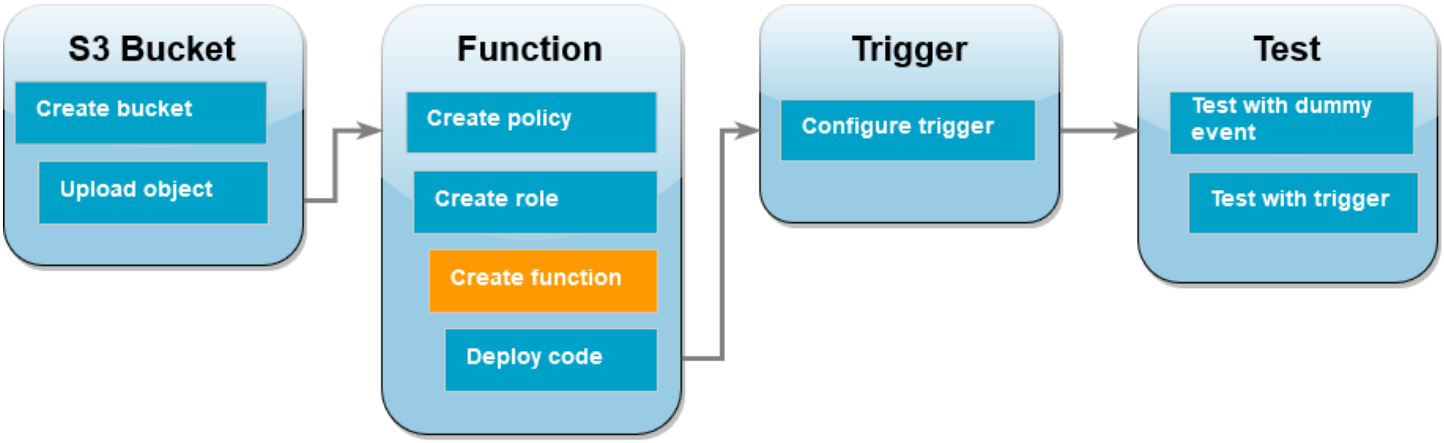
実行ロールとは、AWS サービスとリソースにアクセスする許可を Lambda 関数に付与する AWS Identity and Access Management (IAM) ロールです。この手順では、前のステップで作成したアクセス権限ポリシーを使用して実行ロールを作成します。

実行ロールを作成して、カスタム許可ポリシーをアタッチするには

1. IAM コンソールのロールページを開きます。

2. [ロールの作成] を選択します。
3. 信頼されたエンティティには、[AWS サービス] を選択し、ユースケースには [Lambda] を選択します。
4. [次へ] をクリックします。
5. ポリシー検索ボックスに、「**s3-trigger-tutorial**」と入力します。
6. 検索結果で作成したポリシー (`s3-trigger-tutorial`) を選択し、[次へ] を選択します。
7. [ロールの詳細] で [ロール名] に **lambda-s3-trigger-role** を入力してから、[ロールの作成] を選択します。

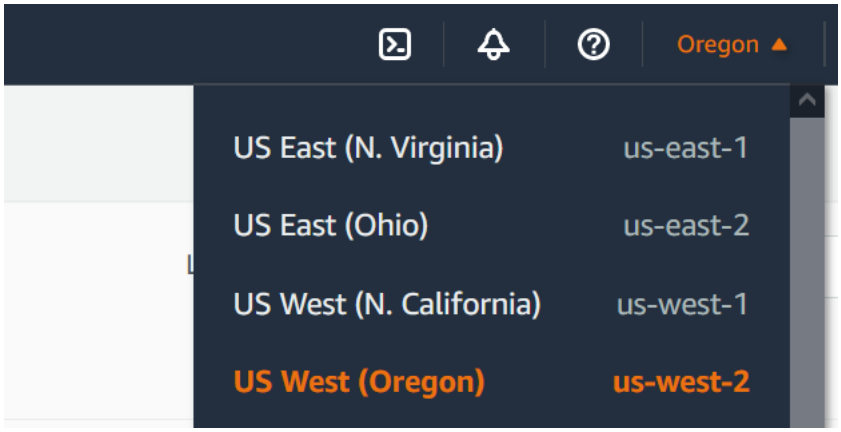
Lambda 関数を作成する



Python 3.12 ランタイムを使用してコンソールで Lambda 関数を作成します。

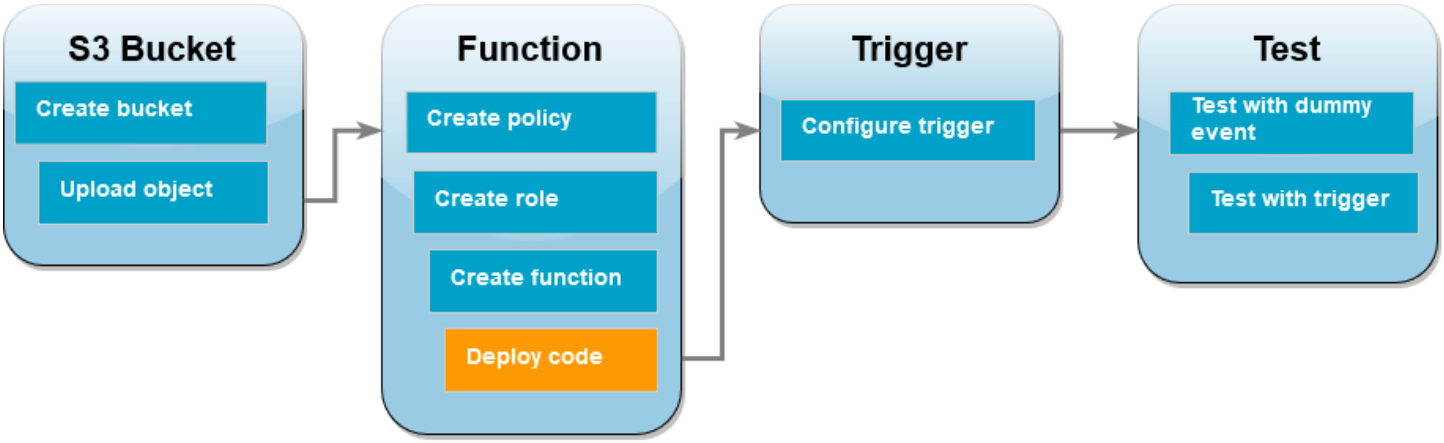
Lambda 関数を作成するには

1. Lambda コンソールの関数ページを開きます。
2. Amazon S3 バケットを作成したときと同じ AWS リージョン で操作していることを確認してください。画面上部にあるドロップダウンリストを使用して、リージョンを変更できます。



3. [関数の作成] を選択します。
4. [一から作成] を選択します。
5. [基本的な情報] で、以下を実行します。
   a. [関数名] に `s3-trigger-tutorial` と入力します。
   b. [ランタイム] には、[Python 3.12] を選択します。
   c. [アーキテクチャ] で [x86_64] を選択します。
6. [デフォルトの実行ロールの変更] タブで、次の操作を行います。
   a. タブを展開し、[既存のロールを使用する] を選択します。
   b. 先ほど作成した `lambda-s3-trigger-role` を選択します。
7. [Create function (関数の作成)] を選択します。

関数コードをデプロイする



このチュートリアルは Python 3.12 ランタイムを使用しますが、他のランタイム用のサンプルコードのファイルも用意しています。次のボックスでタブを選択すると、関心のあるランタイムのコードが表示されます。

Lambda 関数は、Amazon S3 から受信する `event` パラメータから、アップロードされたオブジェクトのキー名およびバケットの名前を取得します。次に、関数は AWS SDK for Python (Boto3) から「get_object」メソッドを使用し、アップロードされたオブジェクトのコンテンツタイプ (MIME タイプ) を含むオブジェクトのメタデータを取得します。

関数コードをデプロイするには

1. 次のボックスで [Python] タブを選択し、コードをコピーします。

   .NET

   AWS SDK for .NET

   > **Note**
   >
   > GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

   .NET を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
```

```
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext
context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key =
HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
{e.Message}");

                return string.Empty;
            }
        }
```

```
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
```

```
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

exports.handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
```

```
    }
};
```

TypeScript を使用して Lambda で S3 イベントを消費する。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用して Lambda で S3 イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
```

```
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で S3 イベントを消費します。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')
```

```
s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda での S3 イベントの消費。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'
```

```
def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

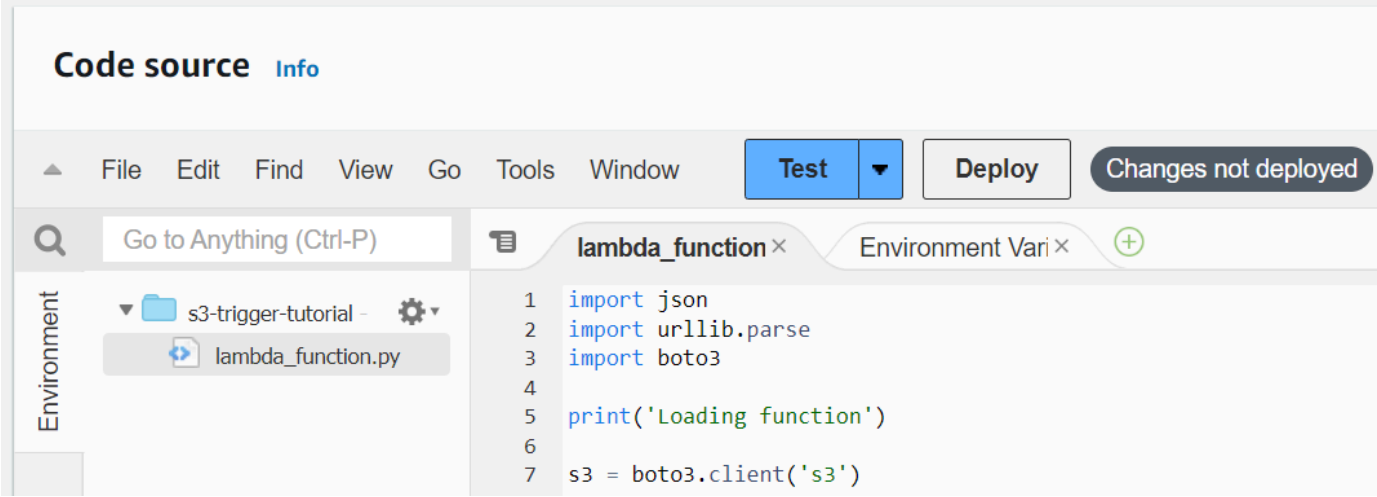
Rust を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;
```

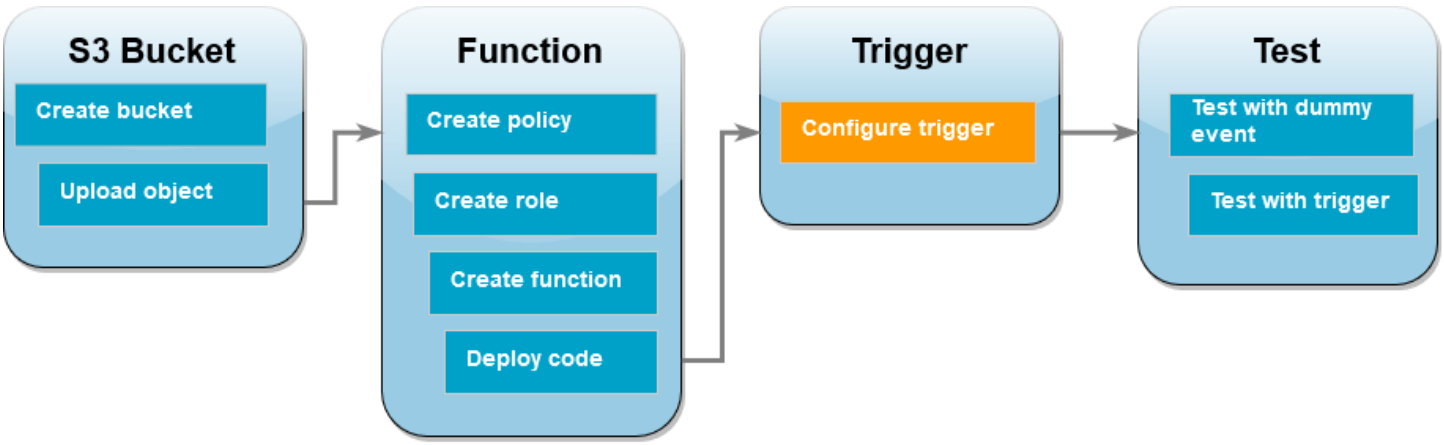
```
    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. Lambda コンソールの [コードソース] ペインで、コードを [lambda_function.py] ファイルに貼り付けます。



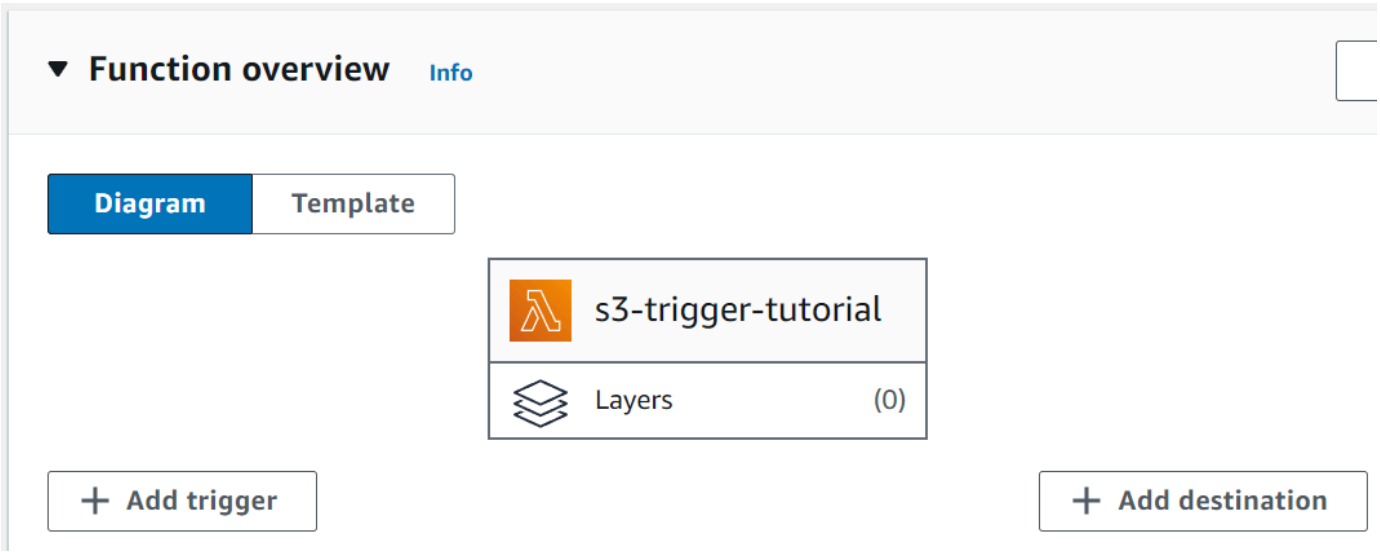
3. [デプロイ] をクリックします。

Amazon S3 トリガーを作成する



Amazon S3 トリガーを作成するには

1. [関数の概要] ペインで、[トリガーを追加] を選択します。



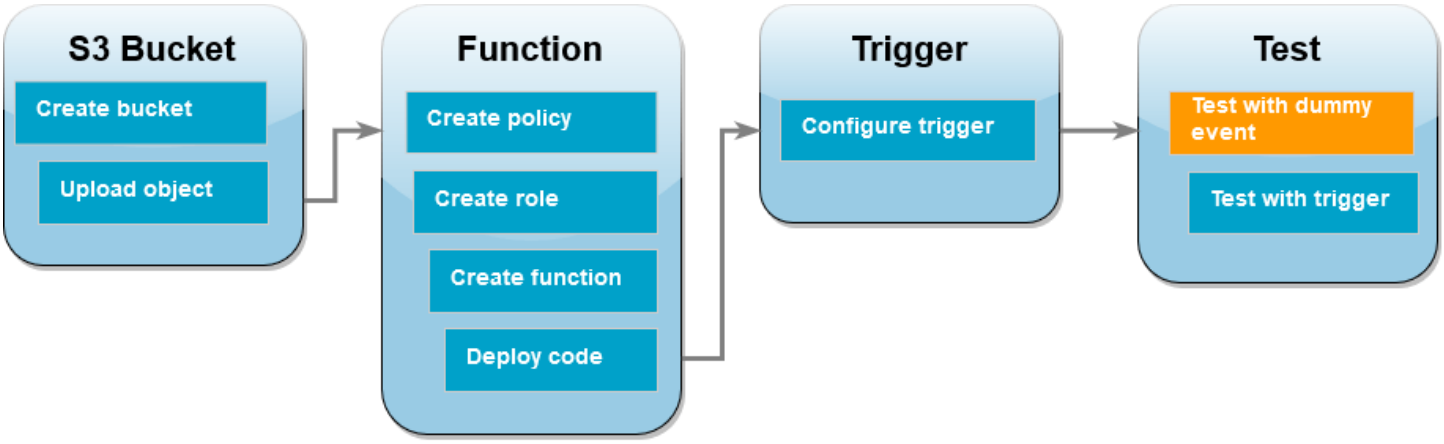
2. [S3] を選択します。
3. [バケット]で、前のチュートリアルで作成したバケットを選択します。
4. [イベントタイプ] で、[すべてのオブジェクトの作成イベント] を選択します。
5. [再帰呼び出し] でチェックボックスを選択して、入力と出力に同じ Amazon S3 バケットを使用することは推奨されないことを確認します。
6. 追加 を選択します。

> **Note**
>
> Lambda コンソールを使用して Lambda 関数の Amazon S3 トリガーを作成すると、Amazon S3 は指定したバケットに対してイベント通知を設定します。このイベント通知を設定する前に、Amazon S3 は一連のチェックを実行して、イベントの送信先が存在し、必要な IAM ポリシーがあることを確認します。また、Amazon S3 は、そのバケットに設定されている他のイベント通知に対してもこれらのテストを実行します。
> このチェックが行われるために、既に存在しないリソースや必要なアクセス許可ポリシーを持たないリソースのイベントの送信先がバケットで既に設定されている場合、Amazon S3 は新しいイベント通知を作成できません。トリガーを作成できなかったことを示す次のエラーメッセージが表示されます。

```
An error occurred when creating the trigger: Unable to validate the following
 destination configurations.
```

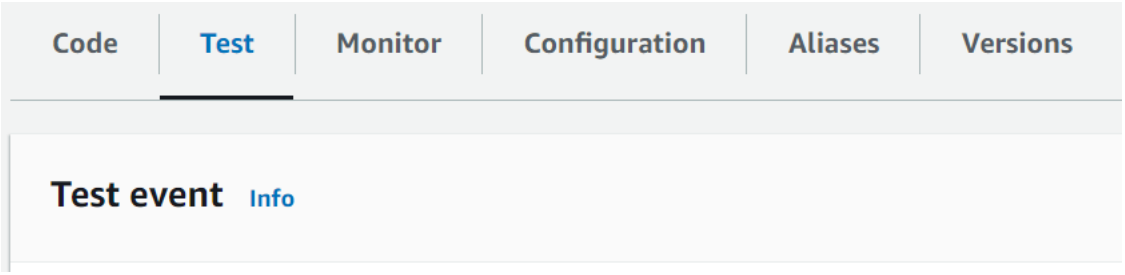
このエラーは、以前に同じバケットを使用して別の Lambda 関数のトリガーを設定しており、その後に関数を削除したり、そのアクセス許可ポリシーを変更したりした場合に表示されます。

Lambda 関数をダミーイベントでテストする



Lambda 関数をダミーイベントでテストするには

1. 関数の Lambda コンソールページで、[テスト] タブを選択します。



2. イベント名()で、`MyTestEvent` と入力します。
3. [イベント JSON] で、次のテストイベントを貼り付けます。次の値を必ず置き換えてください。

   - `us-east-1` をAmazon S3 バケットを作成したリージョンに置き換えます。
   - `DOC-EXAMPLE-BUCKET` の両方のインスタンスをお使いの Amazon S3 バケットの名前に置き換えます。

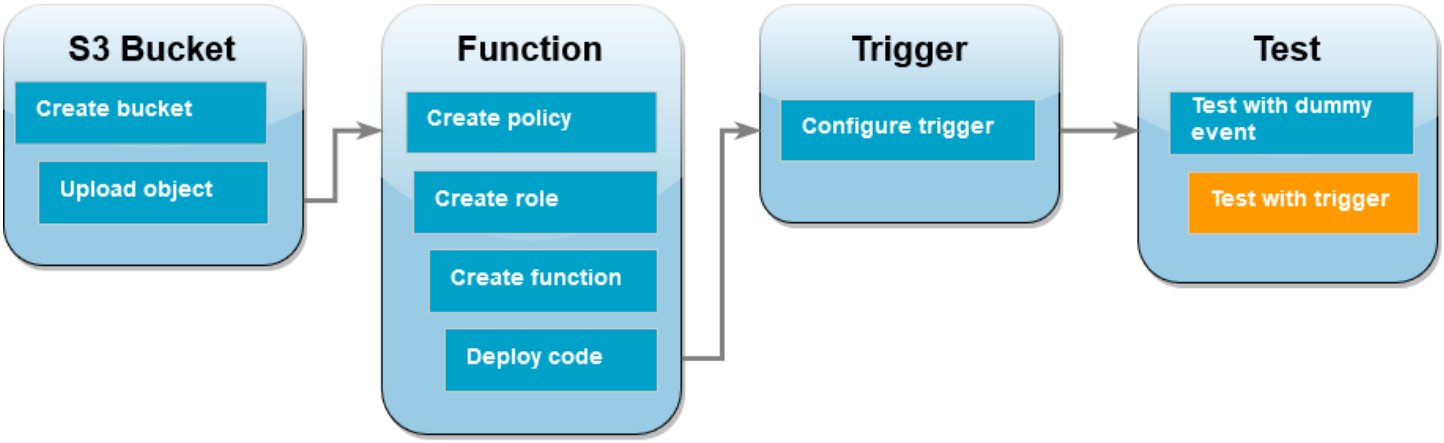
- test%2FKey を前にバケットにアップロードしたテストオブジェクトの名前 (例えば、HappyFace.jpg) に置き換えます。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

4. [Save] を選択します。
5. [Test] を選択します。
6. 関数が正常に実行されると、[実行結果] タブに次のような出力が表示されます。

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'DOC-EXAMPLE-BUCKET', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Amazon S3 トリガーを使用して Lambda 関数をテストする



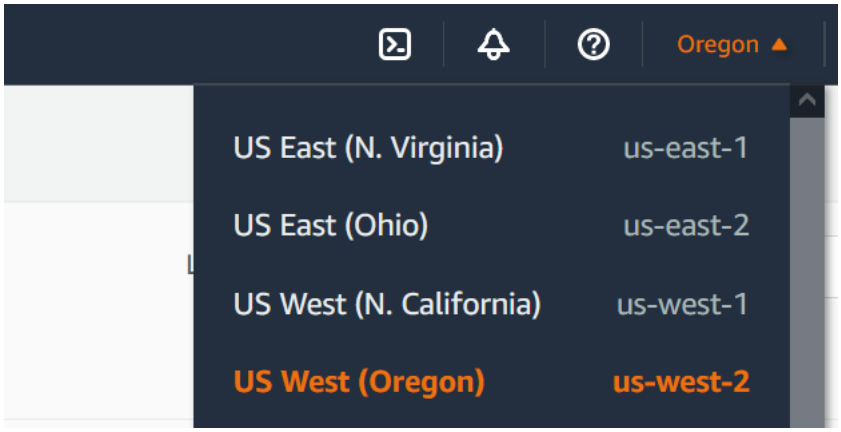
設定したトリガーで関数をテストするには、コンソールを使用して Amazon S3 バケットにオブジェクトをアップロードします。Lambda 関数が予想通りに実行されたことを確認するには、CloudWatch Logs を使用して関数の出力を確認します。

オブジェクトを Amazon S3 バケットにアップロードするには

1. Amazon S3 コンソールの「バケット」ページを開き、先ほど作成したバケットを選択します。
2. [アップロード] を選択します。
3. [ファイルを追加] を選択し、ファイルセレクターを使用してアップロードするオブジェクトを選択します。このオブジェクトには任意のファイルを選択できます。
4. [開く]、[アップロード] の順に選択します。

CloudWatch Logs を使用して関数の呼び出しを確認する方法

1. CloudWatch コンソールを開きます。
2. Lambda 関数を作成したところと同じ AWS リージョン で操作していることを確認してください。画面上部にあるドロップダウンリストを使用して、リージョンを変更できます。



3. [ログ]、[ロググループ] の順に選択します。
4. 関数のロググループの名前を選択します (`/aws/lambda/s3-trigger-tutorial`) 。
5. [ログストリーム] から、最新のログストリームを選択します。
6. Amazon S3 トリガーに応答して関数が正しく呼び出される場合、次のような内容と同じような出力が表示されます。表示される `CONTENT TYPE` は、バケットにアップロードしたファイルのタイプによって異なります。

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

S3 バケットを削除するには

1. Amazon S3 コンソールを開きます。
2. 作成したバケットを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにバケットの名前を入力します。
5. [バケットを削除] を選択します。

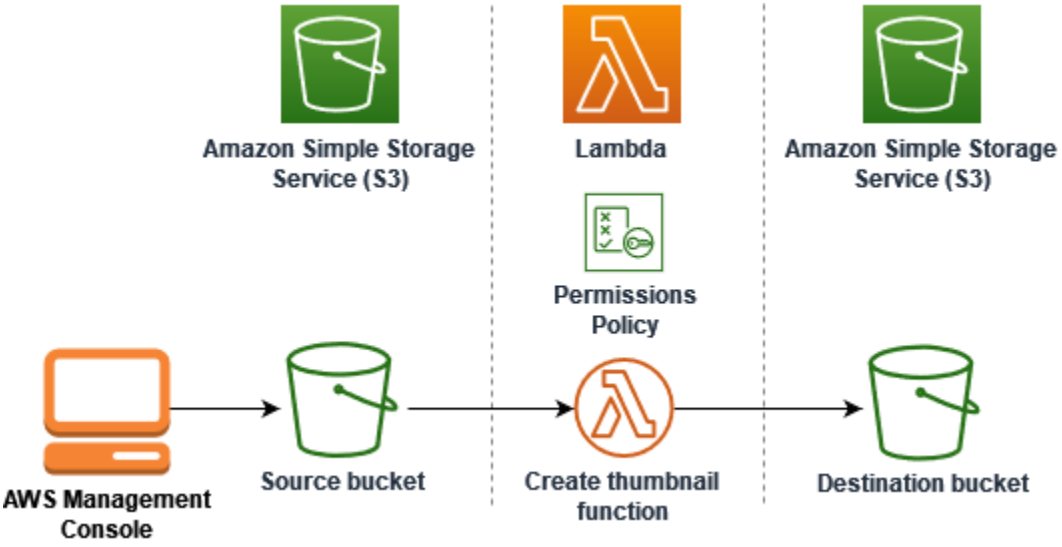
次のステップ

チュートリアル: Amazon S3 トリガーを使用してサムネイル画像を作成する では、Amazon S3 トリガーが関数を呼び出します。この感想は、バケットにアップロードされる各イメージファイルにサムネイルイメージを作成します。このチュートリアルでは、AWS と Lambda ドメインに関する中級レベルの知識が必要です。AWS Command Line Interface (AWS CLI) を使用してリソースを作成し、

関数およびその依存関係に .zip ファイルアーカイブのデプロイパッケージを作成する方法を示します。

チュートリアル: Amazon S3 トリガーを使用してサムネイル画像を作成する

このチュートリアルでは、Amazon Simple Storage Service (Amazon S3) バケットに追加された画像のサイズを変更する Lambda 関数を作成および構成します。バケットに画像ファイルを追加すると、Amazon S3 は Lambda 関数を呼び出します。その後、この関数が画像のサムネイルバージョンを作成し、別の Amazon S3 バケットに出力します。



このチュートリアルを完了するには、次のステップを実行します。

1. ソース元と保存先の Amazon S3 バケットを作成し、サンプル画像をアップロードします。
2. 画像のサイズを変更し、Amazon S3 バケットにサムネイルを出力する Lambda 関数 を作成します。
3. オブジェクトがソースバケットにアップロードされたときに、関数を呼び出す Lambda トリガーを設定します。
4. 最初にダミーイベントを使用して関数をテストし、その後画像をソースバケットにアップロードしてテストします。

これらのステップを完了することで、Lambda を使用して Amazon S3 バケットに追加されたオブジェクトに対してファイル処理タスクを実行する方法が分かるようになります。AWS Command Line Interface または AWS CLI(AWS Management Console) を使って、このチュートリアルを完了できます。

Lambda 用に Amazon S3 トリガーを設定する方法を知るための、より簡単な例が必要であれば、「チュートリアル: Amazon S3 トリガーを使用して Lambda 関数を呼び出す」をお試しください。

トピック

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [マイアカウント] を選んで、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理できます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウントのメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

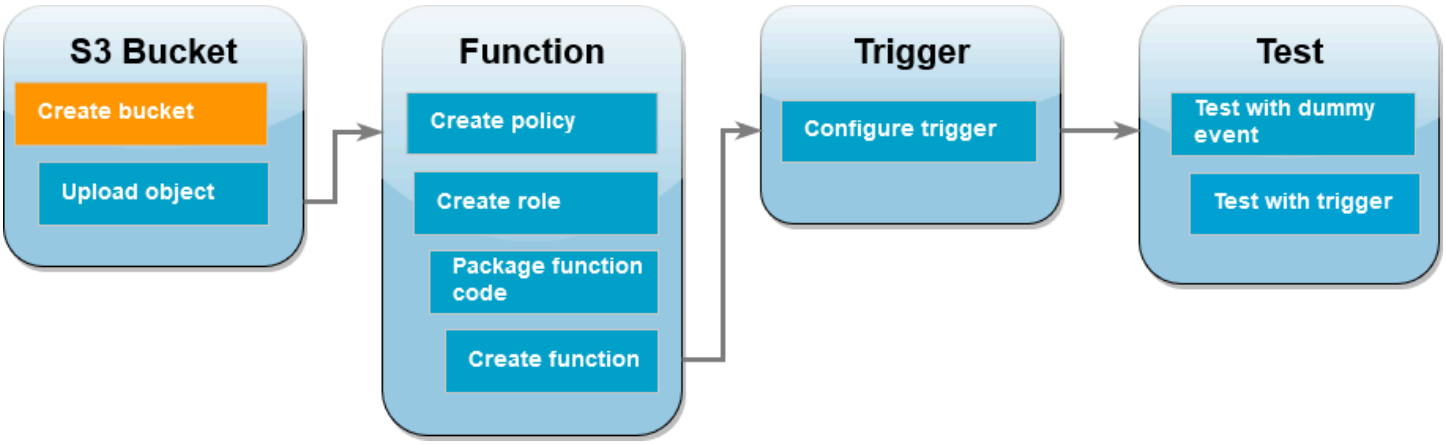
   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

AWS CLI を使用してチュートリアルを完了する場合は、AWS Command Line Interface の最新バージョンをインストールしてください。

Lambda 関数コードには、Python または Node.js を使用できます。使用する言語の言語サポートツールとパッケージマネージャーをインストールします。

2 つの Amazon S3 バケットを作成する



まず、2 つの Amazon S3 バケットを作成します。1 つ目のバケットは、画像をアップロードするソースバケットです。2 つ目のバケットは、関数を呼び出したときにサイズ変更されたサムネイルを保存するために Lambda が使用するバケットです。

AWS Management Console

Amazon S3 バケットを作成する方法 (コンソール)

1. Amazon S3 コンソールのバケットページを開きます。
2. [バケットを作成する] を選択します。
3. [全般設定] で、次の操作を行います。
   a. [バケット名] には、Amazon S3 バケットの命名規則を満たすグローバルに一意な名前を入力します。バケット名は、小文字、数字、ドット (.)、およびハイフン (-) のみで構成できます。
   b. AWS リージョン については、お住まいの地域に最も近い AWS リージョン を選択してください。チュートリアルの後半では、同じ AWS リージョン で Lambda 関数を作成する必要があるため、選択したリージョンを書き留めておいてください。
4. 他のすべてのオプションはデフォルト設定値のままにしておき、[バケットの作成] を選択します。
5. ステップ 1 ~ 4 を繰り返して、送信先のバケットを作成します。[バケット名] には **DOC-EXAMPLE-SOURCE-BUCKET-resized** と入力します。**DOC-EXAMPLE-SOURCE-BUCKET** は先ほど作成したソース元バケットの名前です。

AWS CLI

Amazon S3 バケットを作成する方法 (AWS CLI)

1. 次の CLI コマンドを実行して、ソース元のバケットを作成します。バケットに付ける名前は、グローバルに一意で、Amazon S3 バケットの命名規則に従ったものである必要があります。名前には、小文字、数字、ドット (.)、およびハイフン (-) のみを使用できます。region および LocationConstraint については、お住まいの地域に最も近い AWS リージョン を選択してください。

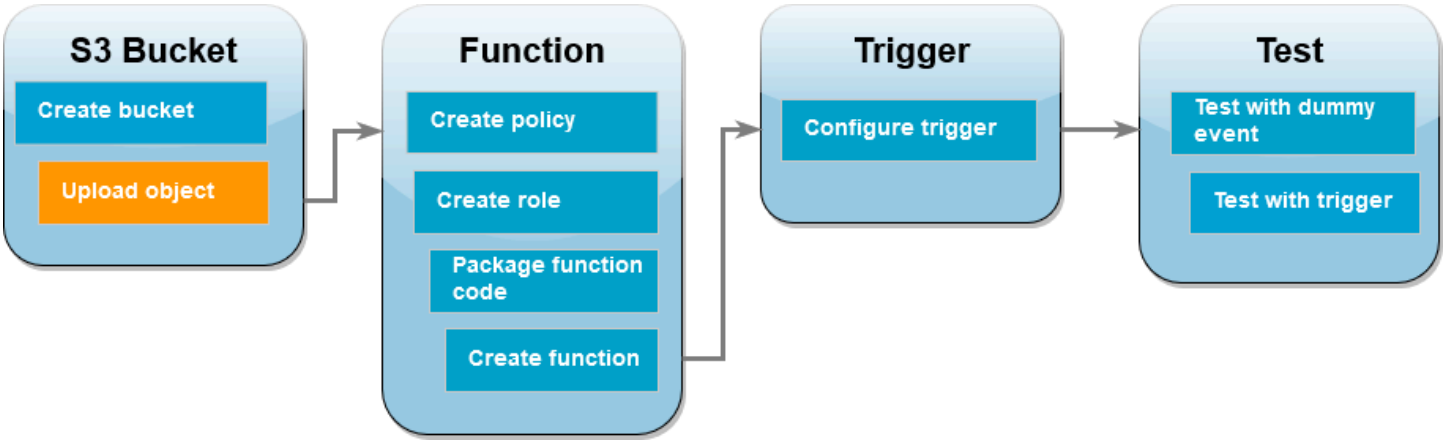
```
aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET --region us-west-2 \
--create-bucket-configuration LocationConstraint=us-west-2
```

   チュートリアルの後半では、ソース元バケットと同じ AWS リージョン で Lambda 関数を作成する必要があるため、選択したリージョンを書き留めておいてください。
2. 次のコマンドを実行して、送信先のバケットを作成します。バケット名には **DOC-EXAMPLE-SOURCE-BUCKET-resized** を使用する必要があります。**DOC-EXAMPLE-**

SOURCE-BUCKET はステップ 1 で作成したソース元バケットの名前です。region および LocationConstraint については、ソース元バケットを作成するときに使用したものと同じ AWS リージョン を選択してください。

```
aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized --region us-west-2 \
--create-bucket-configuration LocationConstraint=us-west-2
```

テスト画像をソース元バケットにアップロードする



チュートリアルの後半では、AWS CLI または Lambda コンソールを使用して、Lambda 関数を呼び出してテストします。関数が正しく動作していることを確認するために、ソース元バケットにはテスト画像が含まれている必要があります。この画像には、任意の JPG または PNG ファイルを使用できます。

AWS Management Console

テスト画像をソース元バケットにアップロードする方法 (コンソール)

1. Amazon S3 コンソールのバケットページを開きます。
2. 前のステップで作成したソースバケットを選択します。
3. [アップロード] を選択します。
4. [ファイルを追加] を選択し、ファイルセレクターを使用してアップロードするオブジェクトを選択します。
5. [開く]、[アップロード] の順に選択します。

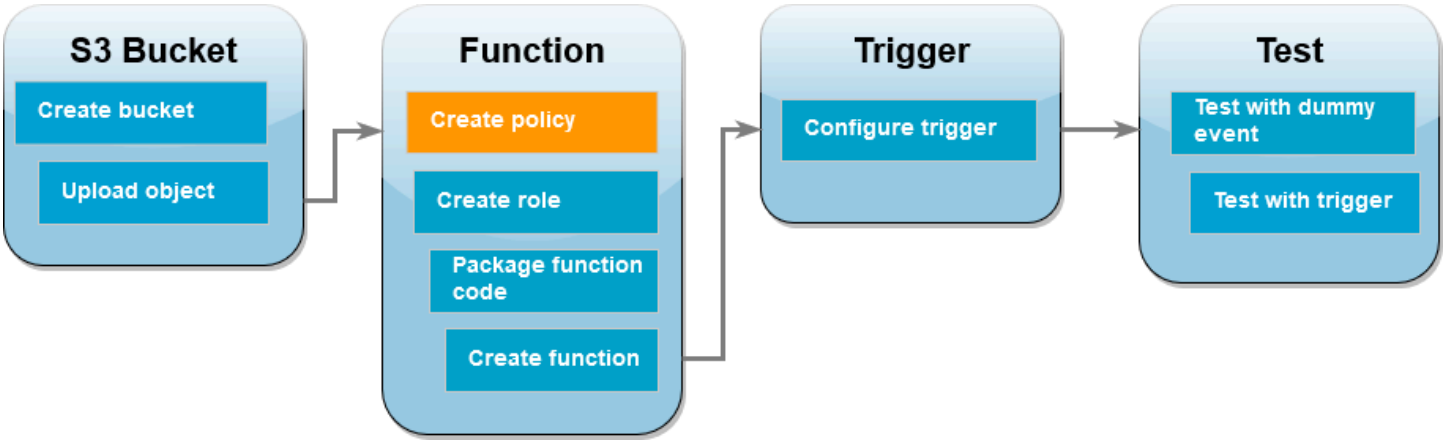
AWS CLI

テスト画像をソース元バケットにアップロードする方法 (AWS CLI)

- アップロードする画像が含まれるディレクトリから、次の CLI コマンドを実行します。--bucket パラメータをソース元バケットの名前に置き換えます。--key および --body パラメータには、テスト画像のファイル名を使用します。

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key HappyFace.jpg --
body ./HappyFace.jpg
```

許可ポリシーを作成する



Lambda 関数を作成するための最初のステップは、許可ポリシーを作成することです。このポリシーは、他の AWS リソースにアクセスするために必要となるアクセス許可を関数に付与します。このチュートリアルでは、ポリシーにより Lambda に Amazon S3 バケットの読み取り権限と書き込み権限が付与され、Amazon CloudWatch Logs に書き込めるようになります。

AWS Management Console

ポリシーを作成する方法 (コンソール)

1. AWS Identity and Access Management (IAM) コンソールの [Policies (ポリシー)] ページを開きます。
2. [Create policy] を選択します。
3. [JSON] タブを選択して、次のカスタムポリシーを JSON エディタに貼り付けます。

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. [Next] を選択します。
5. [ポリシーの詳細] で [ポリシー名] に「**LambdaS3Policy**」と入力します。
6. [Create policy] を選択します。

AWS CLI

ポリシーを作成する方法 (AWS CLI)

1. 次の JSON を `policy.json` という名のファイルに保存します。

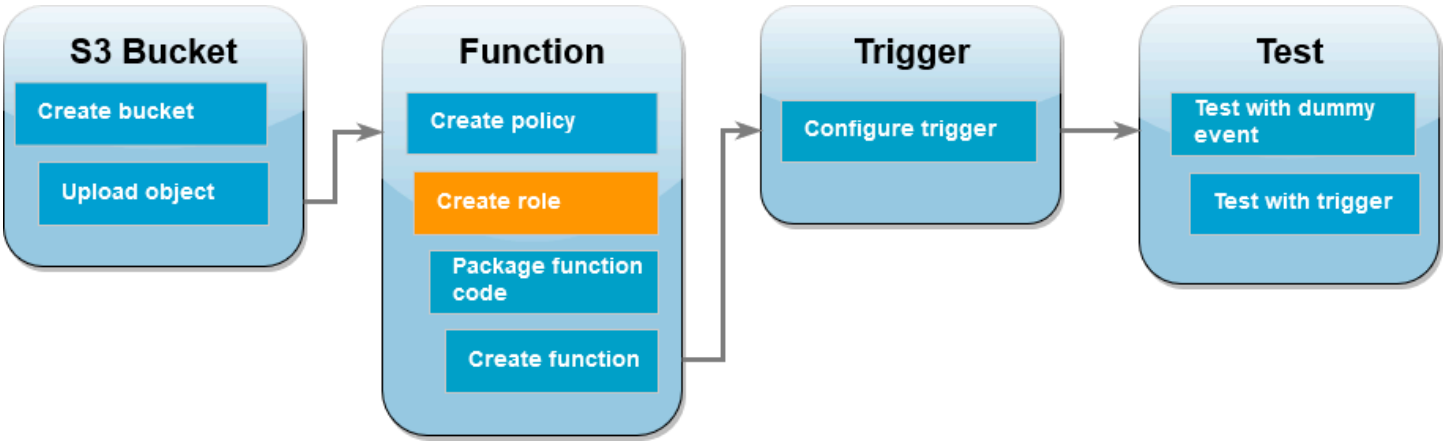
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. JSON ポリシードキュメントを保存したディレクトリから、次の CLI コマンドを実行します。

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

実行ロールを作成する



実行ロールとは、AWS のサービス とリソースにアクセスする許可を Lambda 関数に付与する IAM ロールです。関数に Amazon S3 バケットへの読み取りおよび書き込みアクセス許可を付与するには、前のステップで作成した許可ポリシーをアタッチします。

AWS Management Console

実行ロールを作成して、許可ポリシーをアタッチする方法 (コンソール)

1. (IAM) コンソールの [ロール] ページを開きます。
2. [ロールの作成] を選択します。
3. [信頼できるエンティティタイプ] で AWS のサービス を選択し、[ユースケース] では [Lambda] を選択します。
4. [Next] を選択します。
5. 次の手順を実行して、前のステップで作成した許可ポリシーを追加します。
   a. ポリシー検索ボックスに、「**LambdaS3Policy**」と入力します。
   b. 検索結果内にある LambdaS3Policy のチェックボックスを選択します。
   c. [Next] を選択します。
6. [ロールの詳細] にある [ロール名] には **LambdaS3Role** を入力します。
7. [ロールの作成] を選択します。

AWS CLI

実行ロールを作成して、許可ポリシーをアタッチする方法 (AWS CLI)

1. 次の JSON を `trust-policy.json` という名のファイルに保存します。この信頼ポリシーは、AWS Security Token Service (AWS STS) `AssumeRole` アクションを呼び出すサービスプリンシパルの `lambda.amazonaws.com` アクセス許可を付与することで、Lambda がロールのアクセス許可を使用できるようにします。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

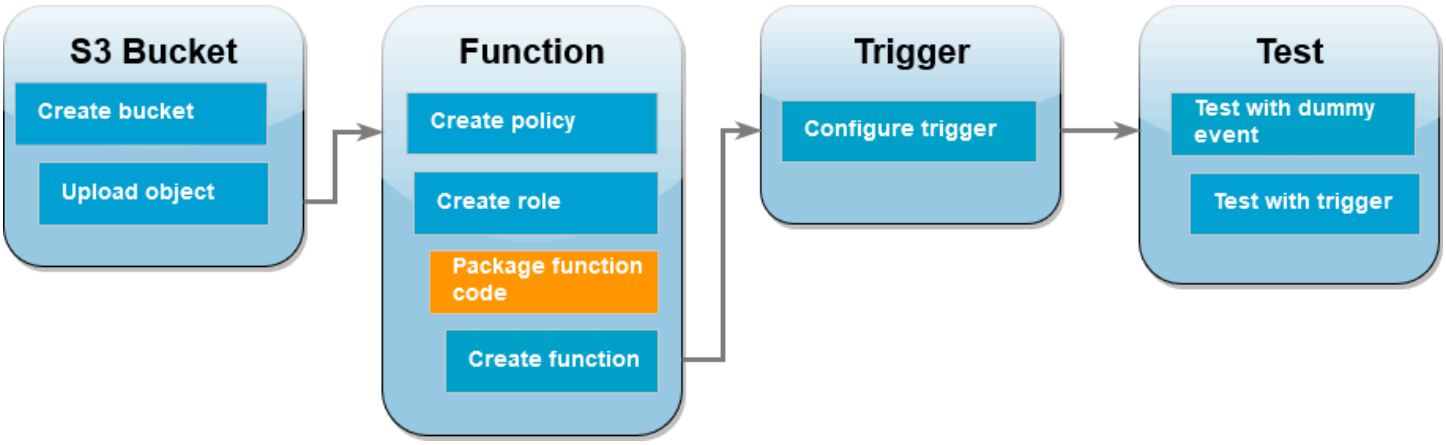
2. JSON 信頼ポリシードキュメントを保存したディレクトリから、次の CLI コマンドを実行して実行ロールを作成します。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 次の CLI コマンドを実行して、前のステップで作成した許可ポリシーをアタッチします。ポリシーの ARN にある AWS アカウント 番号を、自分のアカウント番号へと置き換えます。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

関数デプロイパッケージを作成する



関数を作成するには、関数コードとその依存関係を含むデプロイパッケージを作成します。この `CreateThumbnail` 関数では、関数コードで画像のサイズ変更に別のライブラリを使用します。選択した言語の指示に従って、必要なライブラリを含むデプロイパッケージを作成します。

Node.js

デプロイパッケージを作成する方法 (Node.js)

1. 関数コードと依存関係用に `lambda-s3` という名前のディレクトリを作成し、そこに移動します。

```
mkdir lambda-s3
cd lambda-s3
```

2. 以下の関数コードを `index.mjs` という名のファイルに保存します。必ず `'us-west-2'` を AWS リージョン (独自のソースバケットおよび宛先バケットを作成したもの) に置き換えてください。

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
```

```
const s3 = new S3Client({region: 'us-west-2'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
```

```
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

3. npm を使用して、lambda-s3 ディレクトリに sharp ライブラリをインストールします。sharp の最新バージョン (0.33) は Lambda と互換性がないことに注意してください。このチュートリアルを完了するには、バージョン 0.32.6 をインストールしてください。

```
npm install sharp@0.32.6
```

npm install コマンドによって、モジュール用の node_modules ディレクトリが作成されます。このステップ完了後のディレクトリ構造は、次のようになります。

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

4. 関数コードと依存関係が含まれる.zip ファイル形式のデプロイパッケージを作成します。MacOS および Linux では、次のコマンドを実行します。

```
zip -r function.zip .
```

Windows では、任意の zip ツールを使用して .zip ファイルを作成します。index.mjs、package.json、package-lock.json ファイルと node_modules ディレクトリがすべて、.zip ファイルのルートにあることを確認します。

Python

デプロイパッケージを作成する方法 (Python)

1. サンプルコードをファイル名 lambda_function.py で保存します。

```
import boto3
import os
import sys
import uuid
```

```
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. lambda_function.py ファイルを作成したのと同じディレクトリに、package という名前の新しいディレクトリを作成し、Pillow (PIL) ライブラリと AWS SDK for Python (Boto3) をインストールします。Lambda Python ランタイムには Boto3 SDK のあるバージョンが含まれていますが、ランタイムに含まれている場合でも、関数の依存関係はすべてデプロイパッケージに追加することをお勧めします。詳細については、「Python のランタイム依存関係」を参照してください。

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.9 \
--only-binary=:all: --upgrade \
pillow boto3
```

Pillow ライブラリには C/C++ コードが含まれています。--platform manylinux_2014_x86_64 および --only-binary=:all: のオプションを使用すると、pip は Amazon Linux 2 オペレーティングシステムと互換性のあるプリコンパイル済み

バイナリを含むバージョンの Pillow をダウンロードしてインストールします。これにより、ローカルビルドマシンのオペレーティングシステムやアーキテクチャに関係なく、デプロイパッケージが Lambda 実行環境で動作することが保証されます。

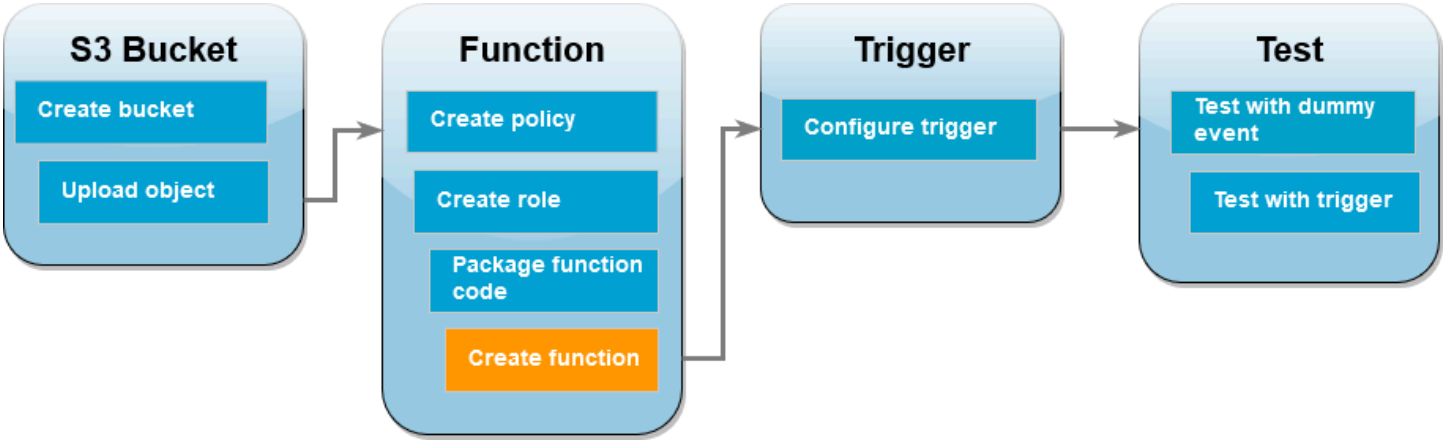
3. アプリケーションコードと Pillow および Boto3 ライブラリを含む .zip ファイルを作成します。Linux または MacOS では、コマンドラインインターフェイスから次のコマンドを実行します。

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

Windows では、任意の zip ツールを使用して、`lambda_function.zip` ファイルを作成します。`lambda_function.py` ファイルと依存関係が含まれるフォルダは、.zip ファイルのルートにインストールする必要があります。

また、Python 仮想環境を使用してデプロイパッケージを作成することもできます。「Python Lambda 関数で .zip ファイルアーカイブを使用する」を参照してください。

Lambda 関数を作成する



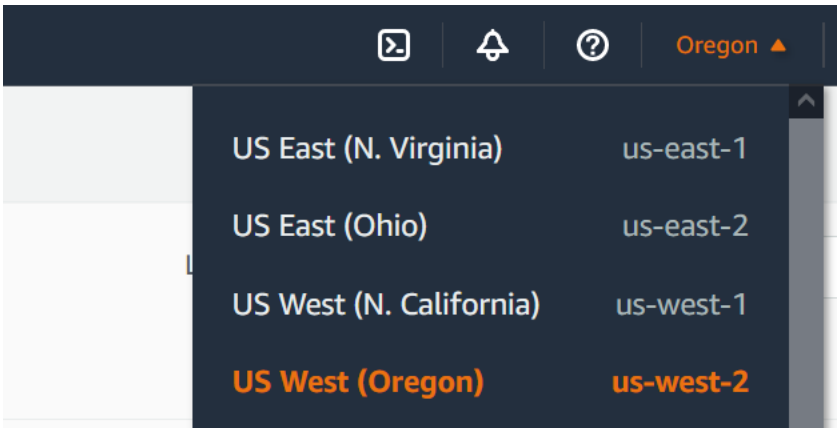
AWS CLI または Lambda コンソールを使用して、Lambda 関数を作成することができます。選択した言語の指示に従って関数を作成します。

AWS Management Console

関数を作成するには (コンソール)

コンソールを使用して Lambda 関数を作成するには、まず「Hello world」コードが含まれるベーシックな関数を作成します。次に、前のステップで作成した .zip または JAR ファイルをアップロードして、このコードを自身で作成した関数コードへと置き換えます。

1. Lambda コンソールの関数ページを開きます。
2. Amazon S3 バケットを作成したときと同じ AWS リージョン で操作していることを確認してください。画面上部にあるドロップダウンリストを使用して、リージョンを変更できます。



3. [関数の作成] を選択します。
4. Author from scratch を選択します。
5. 基本的な情報 で、以下の作業を行います。
   a. [関数名] に「**`CreateThumbnail`**」と入力します。
   b. [ランタイム] には、関数で選択した言語に応じて [Node.js 18.x] または [Python 3.9] を選択します。
   c. [アーキテクチャ] で [x86_64] を選択します。
6. [デフォルトの実行ロールの変更] タブで、次の操作を行います。
   a. タブを展開し、[既存のロールを使用する] を選択します。
   b. 先ほど作成した `LambdaS3Role` を選択します。
7. [Create function (関数の作成)] を選択します。

関数コードをアップロードする方法 (コンソール)

1. [コードソース] ペインで、[アップロード元] をクリックします。
2. [.zip ファイル] をクリックします。
3. [アップロード] を選択します。
4. ファイルセレクターで .zip ファイルを選択し、[開く] を選択します。
5. [Save] を選択します。

AWS CLI

関数を作成する方法 (AWS CLI)

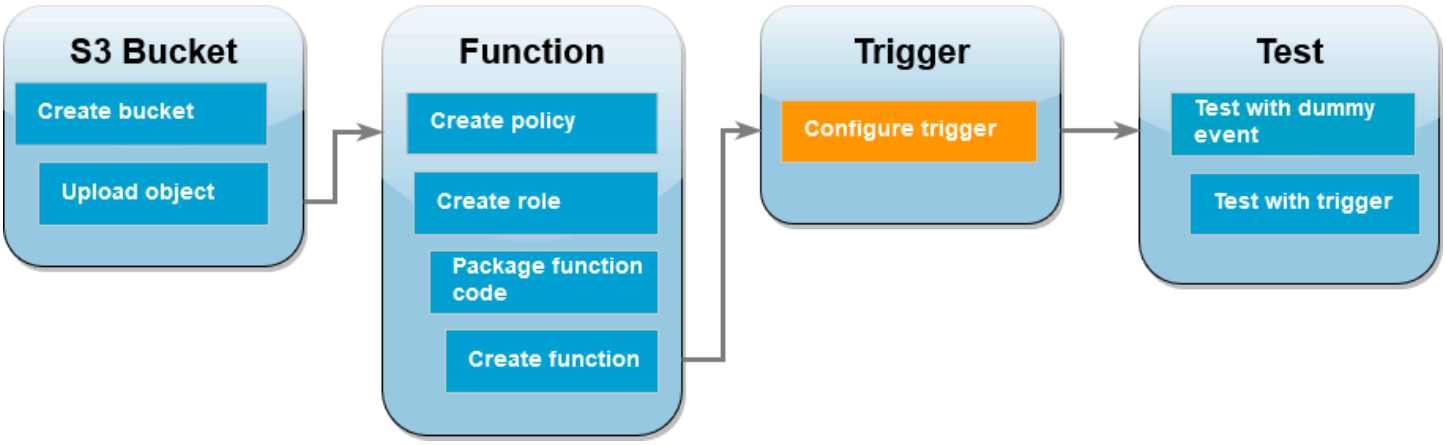
- 選択した言語の CLI コマンドを実行します。`role` パラメータでは、`123456789012` を自分自身の AWS アカウント ID へと置き換えます。`region` パラメータでは、`us-west-2` を Amazon S3 バケットを作成したリージョンへと置き換えます。
  - Node.js の場合は、`function.zip` ファイルが含まれるディレクトリから次のコマンドを実行します。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

  - Python の場合は、`lambda_function.zip` ファイルが含まれるディレクトリから次のコマンドを実行します。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.9 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-west-2
```

関数を呼び出すように Amazon S3 を設定する



ソース元のバケットに画像をアップロードしたときに Lambda 関数を実行するには、関数のトリガーを設定する必要があります。Amazon S3 トリガーは、コンソールまたは AWS CLI を使用して設定できます。

> **Important**
>
> この手順では、オブジェクトがバケット内に作成されるたびに関数を呼び出すように、Amazon S3 バケットを設定します。この設定は、ソース元バケットのみで行うようにしてください。Lambda 関数が自身を呼び出した同じバケットにオブジェクトを作成する場合、関数が連続的にループして呼び出される可能性があります。その結果、予期しない請求がお客様の AWS アカウント に請求される可能性があります。

AWS Management Console

Amazon S3 トリガーを設定する方法 (コンソール)

1. Lambda コンソールの [関数] ページを開き、関数を選択します (`CreateThumbnail`)。
2. [Add trigger] を選択します。
3. [S3] を選択します。
4. [バケット] で、ソース元のバケットを選択します。
5. [イベントタイプ] で、[すべてのオブジェクト作成イベント] を選択します。
6. [再帰呼び出し] でチェックボックスを選択して、入力と出力に同じ Amazon S3 バケットを使用することは推奨されないことを確認します。Lambda の再帰呼び出しパターンについて

詳しくは、Serverless Land の「Lambda 関数が暴走する原因となる再帰パターン」を参照してください。

7. 追加] を選択します。

   Lambda コンソールを使用してトリガーを作成すると、Lambda はリソースベースのポリシーを自動的に作成し、選択したサービスに関数を呼び出すアクセス許可を付与します。

AWS CLI

Amazon S3 トリガーを設定する方法 (AWS CLI)

1. 画像ファイルを追加したときに Amazon S3 ソース元バケットが関数を呼び出すようにするには、まずはリソースベースのポリシーを使用して関数への権限を設定する必要があります。リソースベースのポリシーステートメントが、他の AWS のサービス に関数を呼び出す権限を付与します。Amazon S3 に関数を呼び出す権限を付与するには、次の CLI コマンドを実行します。`source-account` パラメータは必ず自分自身の AWS アカウント ID に置き換えて、自分自身のソース元バケット名を使用するようにしてください。

```
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::DOC-EXAMPLE-SOURCE-BUCKET \
--source-account 123456789012
```

   このコマンドで定義するポリシーにより、Amazon S3 はソース元バケットでアクションが発生した場合にのみ、関数を呼び出すことができるようになります。

   > **Note**
   >
   > Amazon S3 のバケット名は世界的に一意ですが、リソースベースのポリシーを使用する場合には、バケットがアカウントに属していなければならないことを指定するのがベストプラクティスです。これは、バケットを削除したときに、別の AWS アカウント が同じ Amazon リソースネーム (ARN) でバケットを作成する可能性があるからです。

2. 次の JSON を `notification.json` という名のファイルに保存します。この JSON をソースバケットに適用すると、新しいオブジェクトが追加されるたびに Lambda 関数に通知を送

信するようにバケットが設定されます。Lambda 関数 ARN の AWS アカウント 番号と AWS リージョン を、自分自身のアカウント番号とリージョンへと置き換えます。

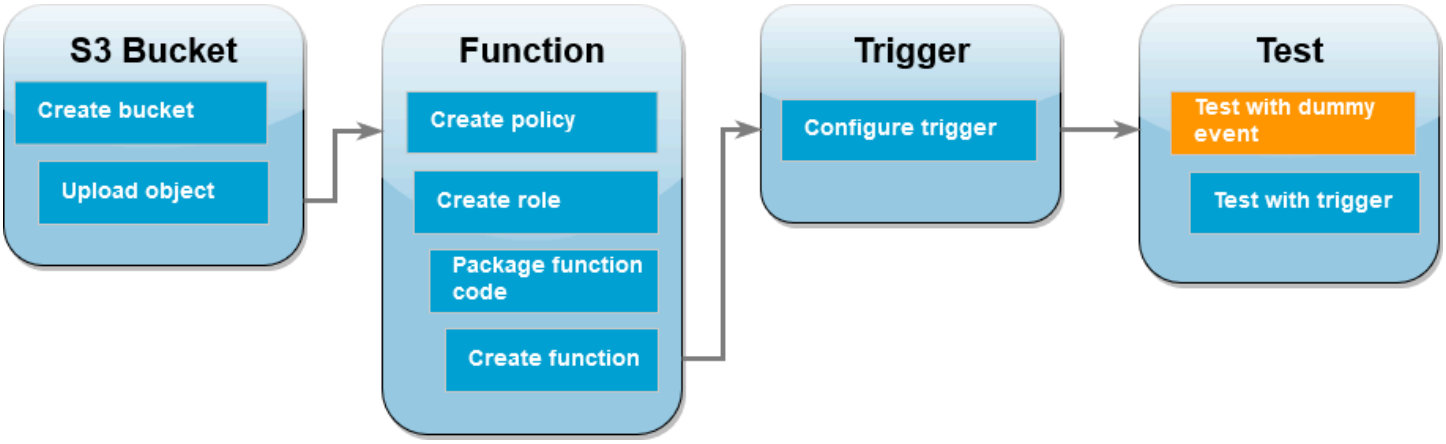
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 次の CLI コマンドを実行して、JSON ファイル内に作成した通知設定をソース元のバケットに適用します。DOC-EXAMPLE-SOURCE-BUCKET を自分自身のソース元バケットの名前へと置き換えます。

```
aws s3api put-bucket-notification-configuration --bucket DOC-EXAMPLE-SOURCE-
BUCKET \
--notification-configuration file://notification.json
```

put-bucket-notification-configuration コマンドと notification-configuration オプションの詳細については、「AWS CLI コマンドリファレンス」の「put-bucket-notification-configuration」を参照してください。

Lambda 関数をダミーイベントでテストする



Amazon S3 ソース元バケットに画像ファイルを追加してセットアップ全体をテストする前に、ダミーイベントで Lambda 関数を呼び出して正しく動作するかどうかをテストします。Lambda 内のイベントは、関数が処理するデータが含まれる JSON 形式のドキュメントです。Amazon S3 によって関数が呼び出されたとき、関数に送信されるイベントには、バケット名、バケット ARN、オブジェクトキーなどの情報が含まれます。

AWS Management Console

Lambda 関数をダミーイベントでテストする方法 (コンソール)

1. Lambda コンソールの [関数] ページを開き、関数を選択します (`CreateThumbnail`)。
2. [テスト] タブを選択します。
3. テストイベントを作成するには、[テストイベント] ペインで次の操作を行います。
   a. [テストイベントアクション] で、[新しいイベントを作成] を選択します。
   b. イベント名()で、**`myTestEvent`** と入力します。
   c. [テンプレート] で [S3 Put] を選択します。
   d. 次のパラメータの値を自分自身の値へと置き換えます。
      - `awsRegion` では、`us-east-1` を Amazon S3 バケットを作成した AWS リージョン へと置き換えます。
      - `name` では、`DOC-EXAMPLE-BUCKET` を自分自身の Amazon S3 ソース元バケットの名前へと置き換えます。
      - `key` では、`test%2Fkey` を テスト画像をソース元バケットにアップロードする ステップでソース元バケットにアップロードしたテストオブジェクトのファイル名へと置き換えます。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
```

```
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "DOC-EXAMPLE-BUCKET",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. [Save] を選択します。

4. [テストイベント] ペインで [テスト] を選択します。

5. 関数によってサイズ変更した画像が作成され、送信先の Amazon S3 バケットに保存されたかどうかを確認するには、以下を実行してください。

   a. Amazon S3 コンソールの [バケットページ] を開きます。

   b. 送信先のバケットを選択し、サイズ変更したファイルが [オブジェクト] ペインに表示されていることを確認します。

AWS CLI

Lambda 関数をダミーイベントでテストする方法 (AWS CLI)

1. 次の JSON を dummyS3Event.json という名のファイルに保存します。次のパラメータの値を自分自身の値へと置き換えます。

   1. awsRegion では、us-west-2 を Amazon S3 バケットを作成した AWS リージョン へと置き換えます。
   2. name では、DOC-EXAMPLE-SOURCE-BUCKET を自分自身の Amazon S3 ソース元バケットの名前へと置き換えます。
   3. key では、HappyFace.jpg を テスト画像をソース元バケットにアップロードする ステップでソース元バケットにアップロードしたテストオブジェクトのファイル名へと置き換えます。

```
{
  "Records":[
    {
      "eventVersion":"2.0",
      "eventSource":"aws:s3",
      "awsRegion":"us-west-2",
      "eventTime":"1970-01-01T00:00:00.000Z",
      "eventName":"ObjectCreated:Put",
      "userIdentity":{
        "principalId":"AIDAJDPLRKLG7UEXAMPLE"
      },
      "requestParameters":{
        "sourceIPAddress":"127.0.0.1"
      },
      "responseElements":{
        "x-amz-request-id":"C3D13FE58DE4C810",
        "x-amz-id-2":"FMyUVURIY8/IgAtTv8xRjskZQpcIZ9KG4V5Wp6S7S/
JRWeUWerMUE5JgHvANOjpD"
      },
      "s3":{
        "s3SchemaVersion":"1.0",
        "configurationId":"testConfigRule",
        "bucket":{
          "name":"DOC-EXAMPLE-SOURCE-BUCKET",
          "ownerIdentity":{
            "principalId":"A3NL1KOZZKExample"
```

```
          },
          "arn":"arn:aws:s3:::DOC-EXAMPLE-SOURCE-BUCKET"
        },
        "object":{
          "key":"HappyFace.jpg",
          "size":1024,
          "eTag":"d41d8cd98f00b204e9800998ecf8427e",
          "versionId":"096fKKXTRTtl3on89fVO.nfljtsv6qko"
        }
      }
    }
  ]
}
```

2. dummyS3Event.json ファイルを保存したディレクトリから、次の CLI コマンドを実行して関数を呼び出します。このコマンドは、RequestResponse を呼び出しタイプパラメータの値として指定することにより、Lambda 関数を同期的に呼び出します。同期呼び出しと非同期呼び出しの詳細については、「Lambda 関数の呼び出し」を参照してください。

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

AWS CLI のバージョン 2 を使用している場合は、cli-binary-format オプションが必要です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、AWS CLI でサポートされているグローバルコマンドラインオプションを参照してください。

3. 関数が画像のサムネイルバージョンを作成し、それを送信先の Amazon S3 バケットに保存していることを確認します。DOC-EXAMPLE-SOURCE-BUCKET-resized を自分自身の送信先のバケットの名前へと置き換えて、次の CLI コマンドを実行します。

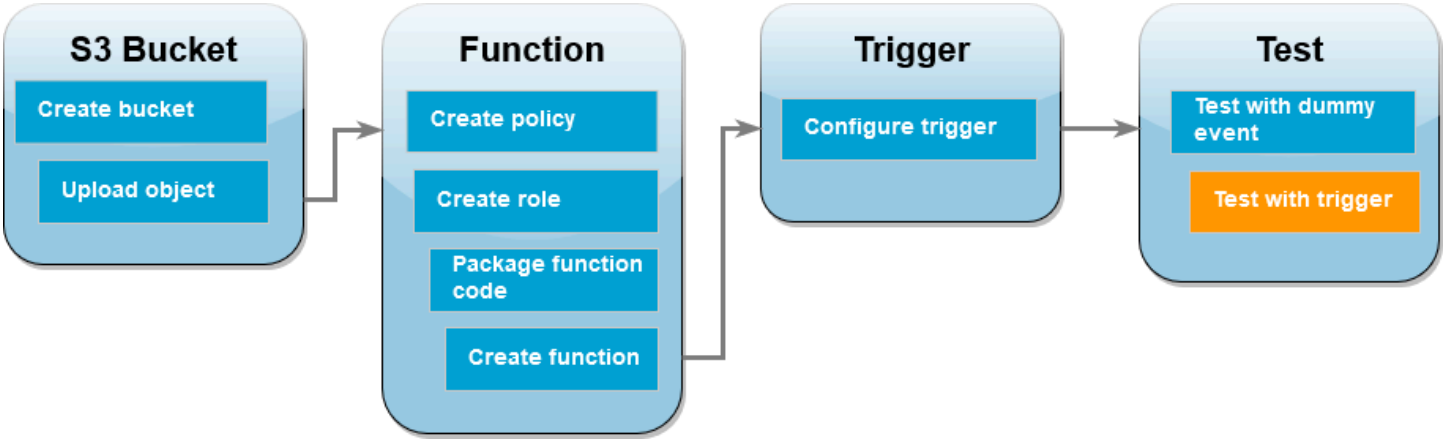
```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

次のような出力が表示されます。Key パラメータには、サイズ変更された画像ファイルのファイル名が表示されます。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
```

```
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Amazon S3 トリガーを使用して関数をテストする



Lambda 関数が正しく動作していることを確認した後は、Amazon S3 ソース元バケットに画像ファイルを追加して、セットアップ全体をテストします。ソース元バケットに画像を追加すると、Lambda 関数が自動的に呼び出されるはずです。関数がファイルのサイズを変更したバージョンを作成し、送信先のバケットに保存します。

AWS Management Console

Amazon S3 トリガーを使用して Lambda 関数をテストする方法 (コンソール)

1. 画像を Amazon S3 バケットにアップロードするには、以下を実行します。

   a. Amazon S3 コンソールの[バケット] ページを開き、ソース元のバケットを選択します。

   b. [アップロード] を選択します。

   c. [ファイルを追加] を選択し、ファイルセレクターを使用してアップロードする画像ファイルを選択します。画像オブジェクトには、任意の .jpg ファイルまたは.png ファイルを使用できます。

   d. [開く]、[アップロード] の順に選択します。

2. 以下を実行して、Lambda が画像ファイルのサイズを変更したバージョンを送信先のバケットに保存したことを確認します。

   a. Amazon S3 コンソールの [バケット] ページに戻り、送信先バケットを選択します。

   b. [オブジェクト] ペインに、Lambda 関数に対する各テストで 1 つずつ作成された、合計 2 つのサイズ変更された画像ファイルが表示されるはずです。サイズを変更した画像をダウンロードするには、ファイルを選択してから [Download] を選択します。

AWS CLI

Amazon S3 トリガーを使用して Lambda 関数をテストする方法 (AWS CLI)

1. アップロードする画像が含まれるディレクトリから、次の CLI コマンドを実行します。--bucket パラメータをソース元バケットの名前に置き換えます。--key および --body パラメータには、テスト画像のファイル名を使用します。テスト画像には、任意の .jpg ファイルまたは .png ファイルを使用できます。

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. 関数が画像のサムネイルバージョンを作成し、それを送信先の Amazon S3 バケットに保存していることを確認します。DOC-EXAMPLE-SOURCE-BUCKET-resized を自分自身の送信先のバケットの名前へと置き換えて、次の CLI コマンドを実行します。

```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

関数が正常に実行されると、以下に類似した出力が表示されます。これで、送信先のバケットに、サイズ変更されたファイルが 2 つ含まれるはずです。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
```

```
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
}
```

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **delete** と入力し、[削除] を選択します。

作成したポリシーを削除する

1. IAM コンソールの [Policies (ポリシー)] ページを開きます。
2. 作成したポリシー (AWSLambdaS3Policy) を選択します。
3. [ポリシーアクション]、[削除] の順に選択します。
4. [削除] を選択します。

実行ロールを削除するには

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

S3 バケットを削除するには

1. Amazon S3 コンソールを開きます。
2. 作成したバケットを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにバケットの名前を入力します。
5. [バケットの削除] を選択します。

Amazon S3 バッチ操作での AWS Lambda の使用

Amazon S3 のバッチ操作を使用することで、大量の Amazon S3 オブジェクトに対して Lambda 関数を呼び出すことができます。Amazon S3 は、バッチオペレーションの進行状況を追跡して、通知を送信し、各アクションのステータスを示す完了レポートを保存します。

バッチ操作を実行するには、Amazon S3 バッチ操作ジョブを作成します。ジョブを作成するときは、マニフェスト (オブジェクトのリスト) を用意し、それらのオブジェクトに対して実行するアクションを設定します。

バッチジョブが開始されると、Amazon S3 はマニフェスト内の各オブジェクトに対して同期的に Lambda 関数を呼び出します。イベントパラメータには、バケットとオブジェクトの名前が含まれます。

以下の例では、Amazon S3 が DOC-EXAMPLE-BUCKET バケット内の customerImage1.jpg という名前のオブジェクトの Lambda 関数に送信するイベントを示しています。

Example Amazon S3 バッチリクエストイベント

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
        }
    ]
}
```

Lambda 関数は、次の例に示すとおり、フィールドを含む JSON オブジェクトを返す必要があります。イベントパラメータから `invocationId` と `taskId` をコピーできます。`resultString` で文字列を返します。Amazon S3 は、`resultString` 値を完了レポートに保存します。

Example Amazon S3 バッチリクエストのレスポンス

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Amazon S3 バッチ操作からの Lambda 関数の呼び出し

Lambda 関数は、非修飾または修飾関数 ARN を使用して呼び出すことができます。バッチジョブ全体に同じ関数バージョンを使用する場合は、ジョブの作成時に FunctionARN パラメータで特定の関数バージョンを構成します。エイリアスまたは $LATEST 修飾子を設定した場合、ジョブ実行中にエイリアスまたは $LATEST が更新されると、バッチジョブは新しいバージョンの関数の呼び出しをただちに開始します。

既存の Amazon S3 イベントベースの関数をバッチ操作に再利用することはできません。これは、Amazon S3 バッチ操作によって異なるイベントパラメータが Lambda 関数に渡され、特定の JSON 構造を持つリターンメッセージが期待されるためです。

Amazon S3 Batch Job 用に作成するリソースベースのポリシーで、Lambda 関数を呼び出すためのジョブに、アクセス許可が設定されていることを確認します。

関数の実行ロールで、Amazon S3 が関数の実行時にロールを引き受けるための信頼ポリシーを設定します。

関数で AWS SDK を使用して Amazon S3 リソースを管理する場合は、実行ロールに Amazon S3 のアクセス許可を追加する必要があります。

ジョブが実行されると、Amazon S3 は複数の関数インスタンスを起動して、関数の同時実行数の上限に達するまで Amazon S3 オブジェクトを並列処理します。Amazon S3 は、小規模なジョブでコストが高騰することを避けるため、インスタンスの初期ランプアップを制限します。

Lambda 関数が `TemporaryFailure` レスポンスコードを返した場合、Amazon S3 はオペレーションを再試行します。

Amazon S3 バッチ操作の詳細については、Amazon S3 デベロッパーガイドの バッチ操作の実行を参照してください。

Amazon S3 バッチ操作で Lambda 関数を使用する方法の例については、Amazon S3デベロッパーガイドの Invoking a Lambda function from Amazon S3 batch operations を参照してください。

S3 Object Lambda を使用したオブジェクトの変換

S3 Object Lambda を使用すると、Amazon S3 GET、HEAD、または LIST リクエストに独自のコードを追加して、データがアプリケーションに返される前にそのデータを変更および処理できます。カスタムコードを使用して、標準 S3 GET、HEAD、または LIST リクエストによって返されるデータを変更し、行のフィルタリング、画像の動的なサイズ変更、機密データの編集などを行うことができるようになりました。AWS Lambda 関数により、コードは AWS によって完全に管理されているインフラストラクチャで実行されるため、データの派生コピーを作成して保存したり、プロキシを実行したりする必要はなく、アプリケーションに変更を加える必要もありません。

詳細については、「S3 Object Lambda を使用したオブジェクトの変換」を参照してください。

チュートリアル

- Amazon S3 Object Lambda を使用したアプリケーションのデータ変換
- Amazon S3 Object Lambda と Amazon Comprehend を使用した PII データの検出と編集
- Amazon S3 Object Lambda を使用して、取得時に画像に動的に透かしを入れる

Secrets Manager での AWS Lambda の使用

AWS Lambda 関数は、Secrets Manager API または任意の AWS Software Development Kit (SDK) を使用して、AWS Secrets Manager とやり取りすることができます。SDK を使用せずに Lambda 関数にある AWS Secrets Manager シークレットを取得してキャッシュするには、AWS Parameters and Secrets Lambda Extension を使用することもできます。詳細については、「AWS Lambda 関数で AWS Secrets Manager シークレットを使用する」を参照してください。

Amazon SES で AWS Lambda を使用する

Amazon SES を使用してメッセージを受信する場合、メッセージが到着したとき Lambda 関数を呼び出すように Amazon SES を設定することができます。次に、受信 E メールイベント (実際には Amazon SNS イベントの Amazon SES メッセージ) をパラメータとして渡すことで、サービスによって Lambda 関数を呼び出すことができます。

Example Amazon SES メッセージイベント

```
{
  "Records": [
    {
      "eventVersion": "1.0",
      "ses": {
        "mail": {
          "commonHeaders": {
            "from": [
              "Jane Doe <janedoe@example.com>"
            ],
            "to": [
              "johndoe@example.com"
            ],
            "returnPath": "janedoe@example.com",
            "messageId": "<0123456789example.com>",
            "date": "Wed, 7 Oct 2015 12:34:56 -0700",
            "subject": "Test Subject"
          },
          "source": "janedoe@example.com",
          "timestamp": "1970-01-01T00:00:00.000Z",
          "destination": [
            "johndoe@example.com"
          ],
          "headers": [
            {
              "name": "Return-Path",
              "value": "<janedoe@example.com>"
            },
            {
              "name": "Received",
              "value": "from mailer.example.com (mailer.example.com [203.0.113.1])
 by inbound-smtp.us-west-2.amazonaws.com with SMTP id o3vrnil0e2ic for
 johndoe@example.com; Wed, 07 Oct 2015 12:34:56 +0000 (UTC)"
            },
```

```
            {
              "name": "DKIM-Signature",
              "value": "v=1; a=rsa-sha256; c=relaxed/relaxed; d=example.com;
 s=example; h=mime-version:from:date:message-id:subject:to:content-type;
 bh=jX3F0bCAI7sIbkHyy3mLYO28ieDQz2R0P8HwQkklFj4=; b=sQwJ+LMe9RjkesGu
+vqU56asvMhrLRRYrWCbV"
            },
            {
              "name": "MIME-Version",
              "value": "1.0"
            },
            {
              "name": "From",
              "value": "Jane Doe <janedoe@example.com>"
            },
            {
              "name": "Date",
              "value": "Wed, 7 Oct 2015 12:34:56 -0700"
            },
            {
              "name": "Message-ID",
              "value": "<0123456789example.com>"
            },
            {
              "name": "Subject",
              "value": "Test Subject"
            },
            {
              "name": "To",
              "value": "johndoe@example.com"
            },
            {
              "name": "Content-Type",
              "value": "text/plain; charset=UTF-8"
            }
          ],
          "headersTruncated": false,
          "messageId": "o3vrnil0e2ic28tr"
        },
        "receipt": {
          "recipients": [
            "johndoe@example.com"
          ],
          "timestamp": "1970-01-01T00:00:00.000Z",
```

```
          "spamVerdict": {
            "status": "PASS"
          },
          "dkimVerdict": {
            "status": "PASS"
          },
          "processingTimeMillis": 574,
          "action": {
            "type": "Lambda",
            "invocationType": "Event",
            "functionArn": "arn:aws:lambda:us-west-2:111122223333:function:Example"
          },
          "spfVerdict": {
            "status": "PASS"
          },
          "virusVerdict": {
            "status": "PASS"
          }
        }
      },
      "eventSource": "aws:ses"
    }
  ]
}
```

詳細については、Amazon SES デベロッパーガイドの Lambda action を参照してください。

Amazon SNS 通知を使用した Lambda 関数の呼び出し

Lambda 関数を使用して、Amazon Simple Notification Service (Amazon SNS) 通知を処理することができます。Amazon SNS では、トピックに送信されるメッセージのターゲットとして Lambda 関数がサポートされます。関数は、同じアカウントまたは他の AWS アカウントのトピックにサブスクライブできます。詳細なチュートリアルについては、「the section called “チュートリアル”」を参照してください。

Lambda は、標準 SNS トピックの SNS トリガーのみをサポートします。FIFO トピックはサポートされていません。

非同期呼び出しの場合、Lambda はそのメッセージをキューに入れ、再試行を処理します。Amazon SNS が Lambda に到達できない場合、またはメッセージが拒否される場合、Amazon SNS は、数時間にわたって間隔を増やして再試行します。詳細については、Amazon SNS のよくある質問の中の信頼性を参照してください。

> **Warning**
>
> Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

トピック

- コンソールを使用した Lambda 関数の Amazon SNS トピックトリガーの追加
- Lambda 関数の Amazon SNS トピックトリガーの手動追加
- SNS イベントシェイプのサンプル
- チュートリアル: Amazon Simple Notification Service での AWS Lambda の使用

コンソールを使用した Lambda 関数の Amazon SNS トピックトリガーの追加

SNS トピックを Lambda 関数のトリガーとして追加する最も簡単な方法は、Lambda コンソールを使用することです。コンソールからトリガーを追加すると、Lambda は SNS トピックからのイベントの受信を開始するために必要なアクセス許可とサブスクリプションを自動的に設定します。

SNS トピックを Lambda 関数のトリガーとして追加するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. トリガーを追加する対象の関数の名前を選択します。
3. [設定] を選択し、[トリガー] を選択します。
4. [Add trigger] を選択します。
5. [トリガーの設定] の下のドロップダウンリストから [SNS] を選択します。
6. [SNS トピック] で、サブスクライブする SNS トピックを選択します。

Lambda 関数の Amazon SNS トピックトリガーの手動追加

Lambda 関数の SNS トリガーを手動で設定するには、次のステップを完了する必要があります。

- 関数に対するリソースベースのポリシーを定義して、SNS がその関数を呼び出すことを許可します。
- Lambda 関数を Amazon SNS トピックにサブスクライブします。

> **Note**
>
> SNS トピックと Lambda 関数が異なる AWS アカウントにある場合は、SNS トピックへのクロスアカウントサブスクリプションを許可するための追加のアクセス許可も付与する必要があります。詳細については、「Amazon SNS サブスクリプションのクロスアカウントのアクセス許可を付与する」を参照してください。

AWS Command Line Interface (AWS CLI) を使用して、これらの両方のステップを完了できます。まず、SNS 呼び出しを許可するためのリソースベースのポリシーを Lambda 関数に対して定義するには、次の AWS CLI コマンドを使用します。`--function-name` の値は Lambda 関数名に置き換え、`--source-arn` の値は SNS トピック ARN に置き換えてください。

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

関数を SNS トピックにサブスクライブするには、次の AWS CLI コマンドを使用します。`--topic-arn` の値は SNS トピック ARN に置き換え、`--notification-endpoint` の値は Lambda 関数 ARN に置き換えてください。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-function
```

SNS イベントシェイプのサンプル

Amazon SNS は、メッセージやメタデータが含まれたイベントを使用して、関数を非同期に呼び出します。

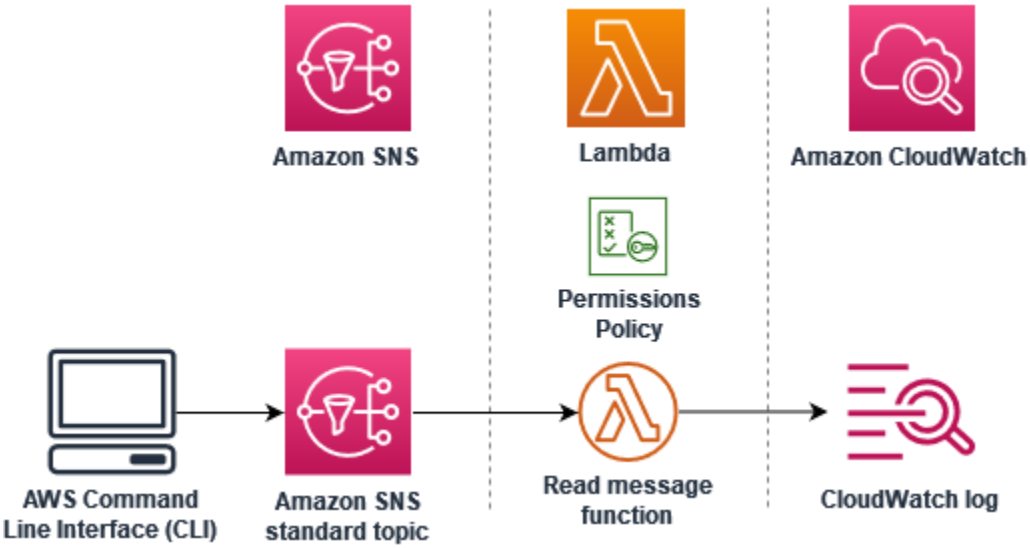
Example Amazon SNS メッセージイベント

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
```

```
        "Type": "Notification",
        "UnsubscribeURL": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

チュートリアル: Amazon Simple Notification Service での AWS Lambda の使用

このチュートリアルでは、Lambda 関数を 1 つの AWS アカウント で使用して、別の AWS アカウント で Amazon Simple Notification Service (Amazon SNS) トピックをサブスクライブします。Amazon SNS トピックにメッセージを発行すると、Lambda 関数がメッセージの内容を読み取り、Amazon CloudWatch Logs に出力します。このチュートリアルを完了するには、AWS Command Line Interface (AWS CLI) を使用します。



このチュートリアルを完了するには、次のステップを実行します。

- アカウント A で、Amazon SNS トピックを作成します。
- アカウント B で、トピックからメッセージを読み取る Lambda 関数を作成します。
- アカウント B で、トピックへのサブスクリプションを作成します。

- アカウント A の Amazon SNS トピックにメッセージを発行し、[アカウント B] の Lambda 関数がメッセージを CloudWatch Logs に出力することを確認します。

これらのステップを完了することで、Lambda 関数を呼び出すように Amazon SNS トピックを設定する方法を学べます。また、別の AWS アカウント のリソースに Lambda を呼び出す許可を与える AWS Identity and Access Management (IAM) ポリシーを作成する方法も学習します。

このチュートリアルでは、2 つの別々の AWS アカウント を使用します。この AWS CLI コマンドは、それぞれが別の AWS アカウント で使用されるように設定された 2 つの名前付きプロファイル `accountA` および `accountB` を使用して実行します。異なるプロファイルを使用するように AWS CLI を設定する方法については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「設定と認証情報ファイルの設定」を参照してください。両方のプロファイルに同じデフォルトの AWS リージョン を設定してください。

2 つの AWS アカウント に対して作成した AWS CLI プロファイルが異なる名前を使用している場合、またはデフォルトのプロファイルと 1 つの名前付きプロファイルを使用している場合は、必要に応じて次の手順の AWS CLI コマンドを変更します。

前提条件

AWS アカウント にサインアップする

AWS アカウントがない場合は、以下のステップを実行して作成します。

AWS アカウントにサインアップするには

1. https://portal.aws.amazon.com/billing/signup を開きます。
2. オンラインの手順に従います。

   サインアップ手順の一環として、通話呼び出しを受け取り、電話キーパッドで検証コードを入力するように求められます。

   AWS アカウントにサインアップすると、AWS アカウントのルートユーザーが作成されます。ルートユーザーには、アカウントのすべてのAWS のサービスとリソースへのアクセス権があります。セキュリティのベストプラクティスとして、ユーザーに管理アクセスを割り当て、ルートユーザーのみを使用してルートユーザーアクセスが必要なタスクを実行してください。

サインアップ処理が完了すると、AWS からユーザーに確認メールが送信されます。https://aws.amazon.com/ の [アカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

AWS アカウント にサインアップしたら、AWS アカウントのルートユーザー をセキュリティで保護し、AWS IAM Identity Center を有効にして、管理ユーザーを作成します。これにより、日常的なタスクにルートユーザーを使用しないようにします。

AWS アカウントのルートユーザーをセキュリティで保護する

1. [ルートユーザー] を選択し、AWS アカウント のメールアドレスを入力して、アカウント所有者として AWS Management Console にサインインします。次のページでパスワードを入力します。

   ルートユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「ルートユーザーとしてサインインする」を参照してください。
2. ルートユーザーの多要素認証 (MFA) を有効にします。

   手順については、IAM ユーザーガイドの「AWS アカウント のルートユーザーの仮想 MFA デバイスを有効にする (コンソール)」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「AWS IAM Identity Center の有効化」を参照してください。
2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

   IAM アイデンティティセンターディレクトリ をアイデンティティソースとして使用するチュートリアルについては、「AWS IAM Identity Center ユーザーガイド」の「デフォルト IAM アイデンティティセンターディレクトリを使用したユーザーアクセスの設定」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン ユーザーガイドの「AWS アクセスポータルにサインインする」を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「権限設定を作成する」を参照してください。
2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

   手順については、「AWS IAM Identity Center ユーザーガイド」の「グループの参加」を参照してください。

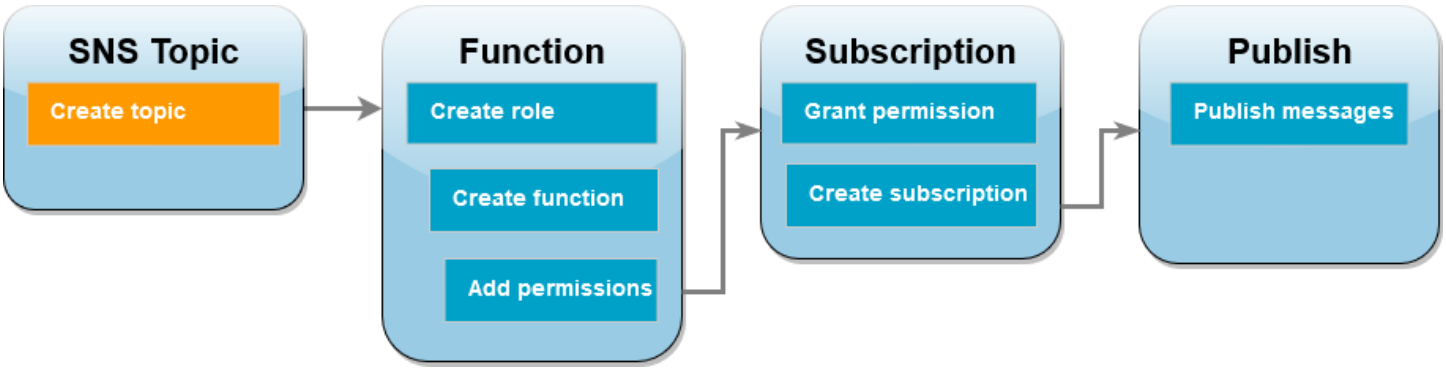
AWS Command Line Interface のインストール

AWS Command Line Interface をまだインストールしていない場合は、「最新バージョンの AWS CLI のインストールまたは更新」にある手順に従ってインストールしてください。

このチュートリアルでは、コマンドを実行するためのコマンドラインターミナルまたはシェルが必要です。Linux および macOS では、任意のシェルとパッケージマネージャーを使用してください。

> **Note**
>
> Windows では、Lambda でよく使用される一部の Bash CLI コマンド (`zip` など) が、オペレーティングシステムの組み込みターミナルでサポートされていません。Ubuntu および Bash の Windows 統合バージョンを取得するには、Windows Subsystem for Linux をインストールします。

Amazon SNS トピックを作成する (アカウント A)



トピックを作成するには

- アカウント A で、次の AWS CLI コマンドを使用して Amazon SNS 標準トピックを作成します。

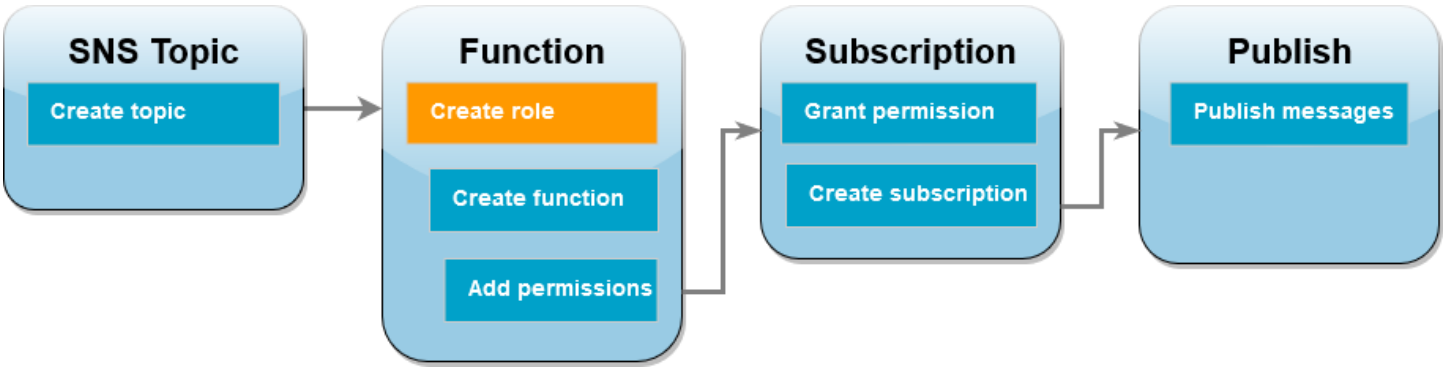
  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

  次のような出力が表示されます。

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  }
  ```

  トピックの Amazon リソースネーム (ARN) をメモしておきます。これは、このチュートリアルで後ほどトピックをサブスクライブするための許可を Lambda 関数に追加するときに必要になります。

関数実行ロールを作成する (アカウント B)

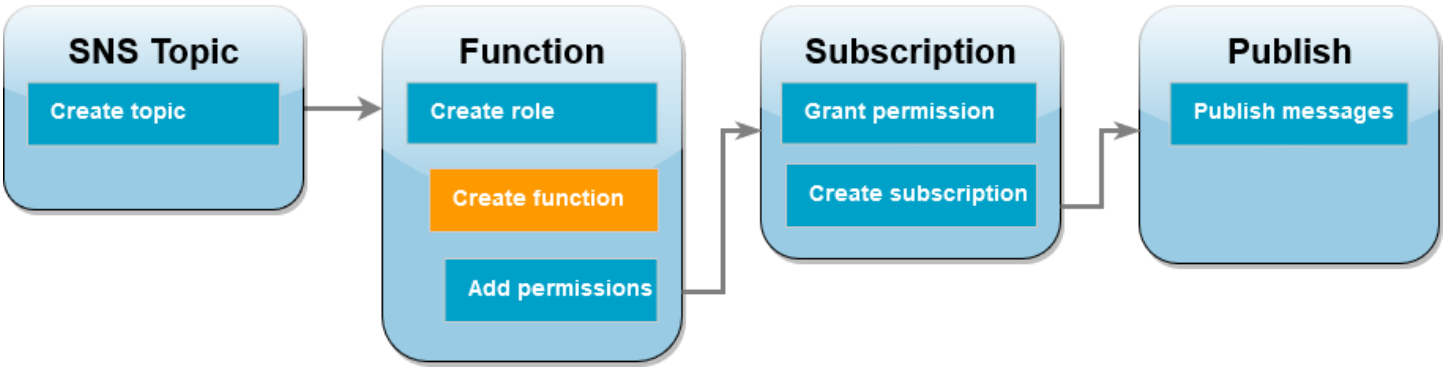


実行ロールとは、AWS サービスとリソースにアクセスする許可を Lambda 関数に付与する IAM ロールです。アカウント B で関数を作成する前に、CloudWatch Logs にログを書き込むための基本的なアクセス許可を関数に与えるロールを作成します。Amazon SNS トピックから読み取るアクセス許可は、後のステップで追加します。

実行ロールを作成するには

1. アカウント B で、IAM コンソールのロールのページを開きます。
2. [ロールの作成] を選択します。
3. [信頼できるエンティティタイプ] で、[AWS サービス] を選択します。
4. [ユースケース] で、[Lambda] を選択します。
5. [Next] を選択します。
6. 次の手順を実行して、基本的なアクセス許可ポリシーをロールに追加します。
   a. [許可ポリシー] 検索ボックスに **AWSLambdaBasicExecutionRole** と入力します。
   b. [Next] を選択します。
7. 次の手順を実行して、ロールの作成を完了します。
   a. [ロールの詳細] にある [ロール名] には **lambda-sns-role** を入力します。
   b. [ロールの作成] を選択します。

Lambda 関数を作成する (アカウント B)



Amazon SNS メッセージを処理する Lambda 関数を作成します。この関数コードは、各レコードのメッセージコンテンツを Amazon CloudWatch Logs に記録します。

このチュートリアルでは Node.js 18.x ランタイムを使用しますが、他のランタイム言語のサンプルコードも提供しています。次のボックスでタブを選択すると、関心のあるランタイムのコードが表示

されます。このステップで使用する JavaScript コードは、[JavaScript] タブに表示されている最初のサンプルにあります。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行する方法を確認してください。

.NET を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
```

```
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)
```

```
func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
```

```
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行する方法を確認してください。

JavaScript を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

TypeScript を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;
```

```
class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で SNS イベントを消費します。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
```

```
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SNS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

関数を作成するには

1. プロジェクト用のディレクトリを作成し、そのディレクトリに切り替えます。

```
mkdir sns-tutorial
cd sns-tutorial
```

2. サンプル JavaScript コードを index.js という名前の新しいファイルにコピーします。
3. 以下の zip コマンドを使用して、デプロイパッケージを作成します。

```
zip function.zip index.js
```

4. 次の AWS CLI コマンドを実行して、アカウント B に Lambda 関数を作成します。

```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

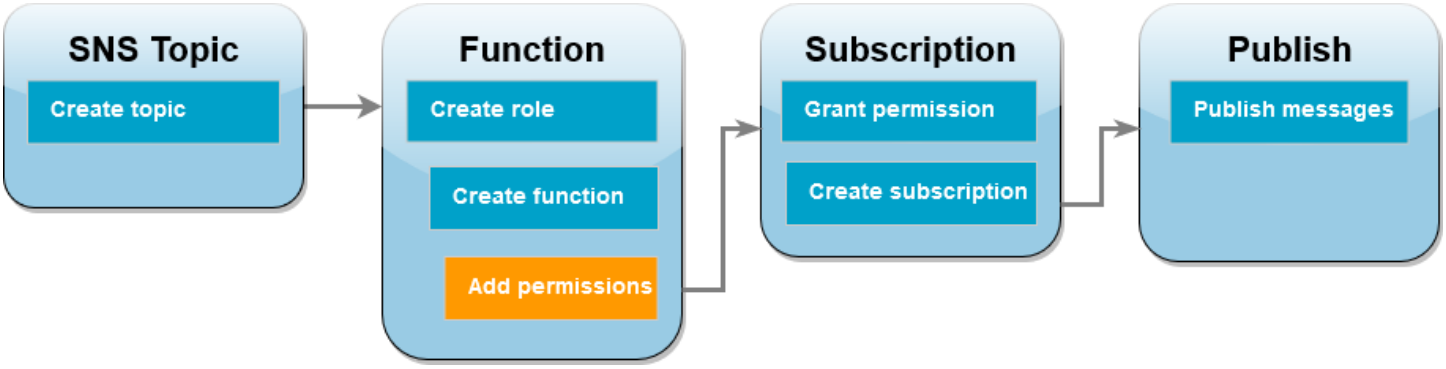
次のような出力が表示されます。

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs18.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
```

```
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. 関数の Amazon リソースネーム (ARN) を記録します。これは、このチュートリアルで後ほど Amazon SNS が関数を呼び出せるようにする許可を追加するときに必要になります。

関数にアクセス許可を追加する (アカウント B)



Amazon SNS が関数を呼び出すには、リソースベースのポリシーのステートメントでその関数にアクセス許可を付与する必要があります。AWS CLI `add-permission` コマンドを使用してこのステートメントを追加します。

Amazon SNS アクセス許可を付与して関数を呼び出すには

- アカウント B で、前に記録した Amazon SNS トピックの ARN を使用して次の AWS CLI コマンドを実行します。

```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

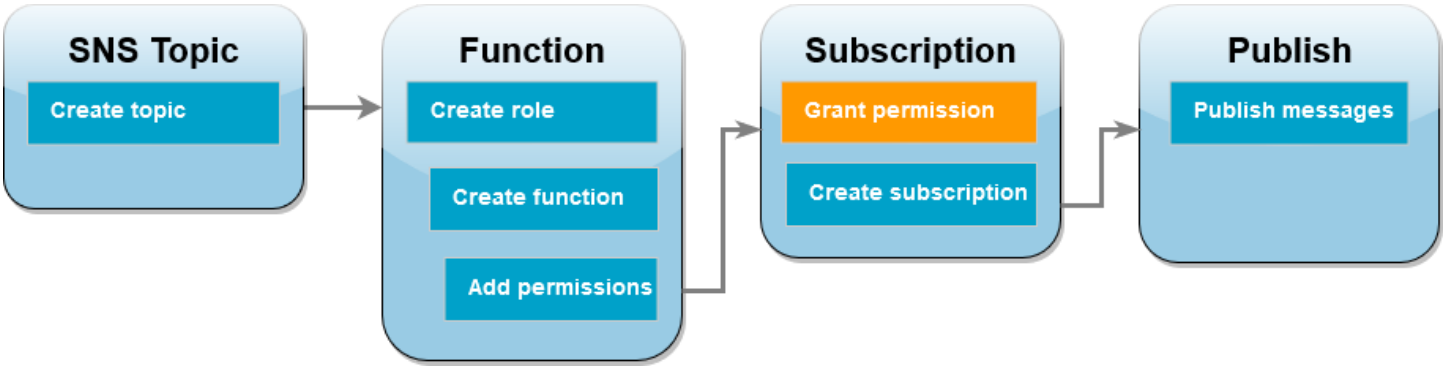
次のような出力が表示されます。

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
```

```
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> Amazon SNS トピックを持つアカウントがオプトイン AWS リージョン でホストされている場合、プリンシパルでリージョンを指定する必要があります。例えば、アジアパシフィック (香港) リージョンの Amazon SNS トピックを使用している場合、プリンシパルに「`sns.amazonaws.com`」ではなく「`sns.ap-east-1.amazonaws.com`」を指定する必要があります。

Amazon SNS サブスクリプションのクロスアカウントのアクセス許可を付与する (アカウント A)



アカウント B の Lambda 関数が アカウント A で作成した Amazon SNS トピックをサブスクライブするには、アカウント B にトピックをサブスクライブするアクセス許可を付与する必要があります。AWS CLI `add-permission` コマンドを使用してこのアクセス許可を付与します。

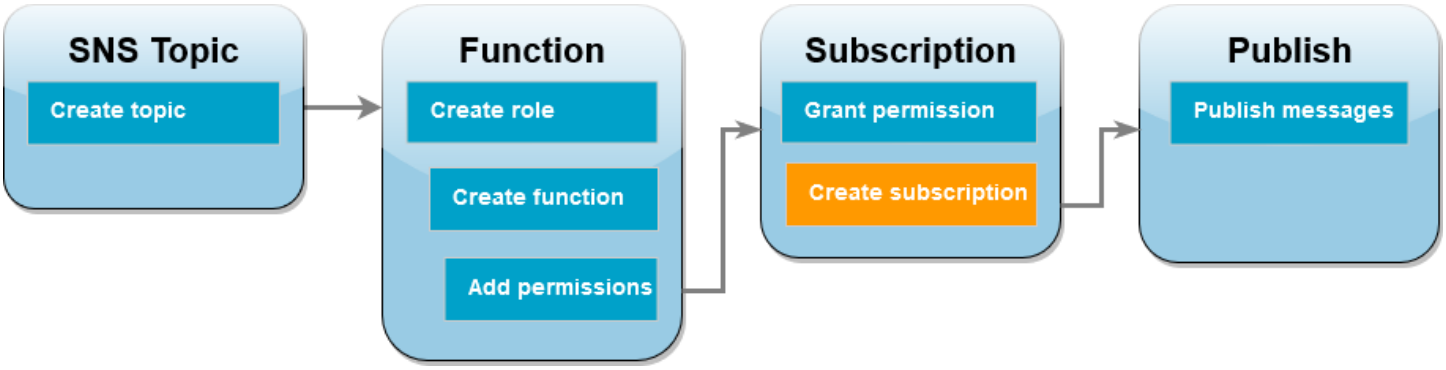
トピックをサブスクライブする許可をアカウント B に付与するには

- アカウント A で、次の AWS CLI コマンドを実行します。以前に記録した Amazon SNS トピックの ARN を使用します。

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
```

```
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

サブスクリプションを作成する (アカウント B)



アカウント B で、Lambda 関数によってチュートリアルの最初にアカウント A で作成した Amazon SNS トピックをサブスクライブします。メッセージがこのトピック (`sns-topic-for-lambda`) に送信されると、Amazon SNS はアカウント B の Lambda 関数 `Function-With-SNS` を呼び出します。

サブスクリプションを作成するには

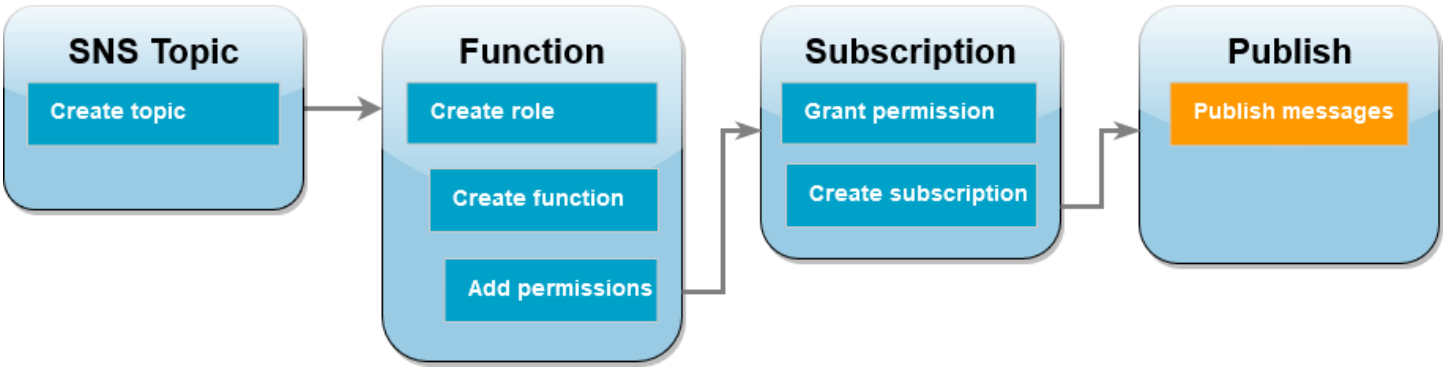
- アカウント B で、次の AWS CLI コマンドを実行します。トピックを作成したデフォルトのリージョンと、トピックおよび Lambda 関数の ARN を使用します。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

次のような出力が表示されます。

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

メッセージをトピックに発行する (アカウント A とアカウント B)



アカウント B の Lambda 関数が アカウント A の Amazon SNS トピックにサブスクライブされたので、トピックにメッセージを発行してセットアップをテストします。Amazon SNS が Lambda 関数を呼び出したことを確認するには、CloudWatch Logs を使用して関数の出力を表示します。

トピックにメッセージを発行して関数の出力を表示するには

1. テキストファイルに「Hello World」と入力して、message.txt として保存します。
2. テキスト ファイルを保存したのと同じディレクトリから、アカウント A で次の AWS CLI コマンドを実行します。自分のトピックの ARN を使用します。

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

これにより、メッセージが Amazon SNS によって承諾されたことを示す、一意の識別子を持つメッセージ ID が返されます。次に、Amazon SNS は、メッセージをトピックのサブスクライバーに配信しようと試みます。Amazon SNS が Lambda 関数を呼び出したことを確認するには、CloudWatch Logs を使用して次の手順で関数の出力を確認します。

3. アカウント B で、Amazon CloudWatch コンソールの [ロググループ] ページを開きます。
4. 関数のロググループの名前を選択します (/aws/lambda/Function-With-SNS) 。
5. 最新のログストリームを選択します。
6. 関数が正しく呼び出されていれば、トピックに発行したメッセージの内容を示す次のような出力が表示されます。

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

リソースのクリーンアップ

このチュートリアル用に作成したリソースは、保持しない場合は削除できます。使用しなくなった AWS リソースを削除することで、AWS アカウント アカウントに請求される料金の発生を防ぎます。

[Account A] (アカウント A) で、Amazon SNS トピックをクリーンアップします。

Amazon SNS トピックを作成するには

1. Amazon SNS コンソールで [Topics (トピック)] ページを開きます。
2. 先ほど作成したトピックを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドに **`delete me`** を入力します。
5. [削除] を選択します。

[Account B] (アカウント A) で、実行ロール、Lambda 関数、および Amazon SNS サブスクリプションをクリーンアップします。

実行ロールを削除する

1. IAM コンソールのロールページを開きます。
2. 作成した実行ロールを選択します。
3. [削除] を選択します。
4. テキスト入力フィールドにロールの名前を入力し、[削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **`delete`** と入力し、[Delete] (削除) を選択します。

Amazon SNS サブスクリプションを削除するには

1. Amazon SNS コンソールのSubscriptions page] (サブスクリプションページ) をを開きます。

2. 作成したサブスクリプションを選択します。
3. [Delete] (削除) を選択し、削除します。

AWS Lambda 関数を使用するためのベストプラクティス

AWS Lambda の使用時に推奨されるベストプラクティスを以下に示します。

トピック

Lambda アプリケーションのベストプラクティスの詳細については、Serverless Land の「アプリケーション設計」を参照してください。AWS アカウントチームに連絡し、アーキテクチャのレビューをリクエストすることもできます。

関数コード

- Lambda ハンドラーをコアロジックから分離します。これにより、関数の単体テストが実行しやすくなります。Node.js では、次のようになります。

```
exports.myHandler = function(event, context, callback) {
 var foo = event.foo;
 var bar = event.bar;
 var result = MyLambdaFunction (foo, bar);

 callback(null, result);
}

function MyLambdaFunction (foo, bar) {
 // MyLambdaFunction logic here
}
```

- 実行環境の再利用を活用して関数のパフォーマンスを向上させます。関数ハンドラー外で SDK クライアントとデータベース接続を初期化し、静的なアセットを /tmp ディレクトリにローカルにキャッシュします。関数の同じインスタンスで処理された後続の呼び出しは、これらのリソースを再利用できます。これにより、関数の実行時間が短縮され、コストが節約されます。

呼び出し間でデータが漏れるのを防ぐため、実行環境を使用してセキュリティ上の懸念があるユーザーデータ、イベント、またはその他の情報を保存しないでください。関数がハンドラー内のメモリに保存できない変更可能な状態に依存している場合は、ユーザーごとに個別の関数または個別のバージョンの関数を作成することを検討してください。

- keep-alive ディレクティブを使用して永続的な接続を維持します。Lambda は、時間の経過とともにアイドル状態の接続を消去します。関数を呼び出すときにアイドル状態の接続を再利用しようとすると、接続エラーが発生します。永続的な接続を維持するには、ランタイムに関連付けられている keep-alive ディレクティブを使用します。例については、「Node.js で Keep-alive を使用して接続を再利用する」を参照してください。
- 環境変数を使用して、オペレーショナルパラメータを関数に渡します。たとえば、Amazon S3 バケットに書き込む場合、書き込み先のバケット名はハードコーディングせずに、環境変数として設定します。
- 関数のデプロイパッケージ内で依存関係を制御します。AWS Lambda 実行環境には、AWS SDK for the Node.js および Python ランタイムなどのライブラリが多数含まれています (詳細なリストについては、Lambda ランタイム を参照してください)。最新の機能やセキュリティ更新プログラムを有効にするために、Lambda はこれらのライブラリを定期的に更新します。この更新により、Lambda 関数の動作が微妙に変化する場合があります。関数で使用する依存関係を完全に制御するには、すべての依存関係をデプロイパッケージでパッケージングします。
- デプロイパッケージをランタイムに必要な最小限のサイズにします。これにより、呼び出しに先立ってデプロイパッケージをダウンロードして解凍する所要時間が短縮されます。Java または .NET Core で作成した関数の場合は、デプロイパッケージの一環として AWS SDK ライブラリ全体をアップロードしないようにします。代わりに、SDK のコンポーネントを必要に応じて選別するモジュール (DynamoDB、Amazon S3 SDK モジュール、Lambda コアライブラリなど) を使用します。
- Java で記述されたデプロイパッケージを Lambda で解凍する所要時間を短縮します。そのために、依存する `.jar` ファイルを別個の /lib ディレクトリにファイルします。これで関数のすべてのコードを多数の `.class` ファイルと一緒に単一の Jar に収納するよりも高速化されます。手順については、「.zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする」を参照してください。
- 依存関係の複雑さを最小限に抑えます。フレームワークを単純化して、実行環境起動時のロードを高速化します。たとえば、Spring Framework などの複雑なフレームワークよりも、Dagger や Guice などの単純な Java 依存関係インジェクション (IoC) フレームワークを使用します。
- Lambda 関数内で任意の条件が満たされるまで、その関数自身を自動的に呼び出すような再帰的なコードを使用しないでください。これを行うと意図しないボリュームで関数が呼び出され、料金が

急増する可能性があります。誤ってこのようなコードを使用した場合は、すぐに関数の予約済同時実行数を 0 に設定して、コードを更新している間のすべての関数の呼び出しをスロットリングします。

- Lambda 関数コードで文書化されていない非公開の API を使用しないでください。AWS Lambda マネージドランタイムでは、Lambda が Lambda の内部 API にセキュリティと機能面の更新を定期的に適用します。これらの内部 API 更新には後方互換性がないことがあり、関数にこれらの非公開 API に対する依存関係がある場合、呼び出しの失敗などの意図しない結果につながります。公開されている API のリストについては、「API リファレンス」を参照してください。
- 冪等性コードを記述します。関数の記述に冪等性コードを使用すると、重複するイベントが同じ方法で処理されるようになります。コードでは、イベントを適切に検証し、重複するイベントを適切に処理する必要があります。詳細については、「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。
- Java DNS キャッシュを使用しない Lambda 関数は、既に DNS レスポンスをキャッシュしています。別の DNS キャッシュを使用すると、接続タイムアウトが発生する可能性があります。

  `java.util.logging.Logger` クラスは JVM DNS キャッシュを間接的に有効にできます。デフォルト設定を上書きするには、 を初期化する前に networkaddress.cache.ttl を 0 に設定しますlogger。例：

  ```
  public class MyHandler {
    // first set TTL property
    static{
     java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
    }
   // then instantiate logger
    var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
  }
  ```

  UnknownHostException 障害を防ぐために、 networkaddress.cache.negative.ttlを 0 に設定することをお勧めします。このプロパティは、AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0環境変数を使用して Lambda 関数に設定できます。

  JVM DNS キャッシュを無効にしても、Lambda のマネージド DNS キャッシュは無効になりません。

Function Configuration

- Lambda 関数のパフォーマンステストは、最適なメモリサイズ設定を選択する上で欠かせない部分です。メモリサイズが増えると、関数で利用できる CPU も同様に増加します。関数のメモリ使用量は呼び出しごとに決定され、Amazon CloudWatch で表示できます。次に示すように、呼び出しごとに REPORT: エントリが作成されます。

  ```
  REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
   Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
  ```

  Max Memory Used: フィールドを分析することで、関数のメモリが不足しているか、関数のメモリサイズをオーバープロビジョニングしているかを判断できます。

  関数のために適切なメモリの構成を見つけるには、オープンソースの AWS Lambda Power Tuning プロジェクトを使用することを推奨しています。詳細については、GitHub の AWS Lambda Power Tuning を参照してください。

  関数のパフォーマンスを最適化するためには、Advanced Vector Extensions 2 (AVX2) が利用可能なライブラリのデプロイも推奨されます。これにより、機械学習による推定、メディア処理、ハイパフォーマンスコンピューティング (HPC)、科学シミュレーション、財務モデリングなど、負荷が大きくなりがちなワークロードを処理できるようになります。詳細については、AVX2 を使用した高速な AWS Lambda 関数の作成を参照してください。
- Lambda 関数のロードテストにより、最適なタイムアウト値を決定します。関数の実行時間を分析し、依存関係サービスの問題に伴って関数の同時実行が必要以上に増えるような状況をより的確に判定します。これは、Lambda のスケーリングを処理しない可能性があるリソースに対して Lambda 関数からネットワークの呼び出しを行うときに特に重要です。
- IAM ポリシーの設定時に最も制限的なアクセス許可を使用します。Lambda 関数に必要なリソースとオペレーションを把握し、実行ロールをこれらのアクセス許可に制限します。詳細については、「AWS Lambda アクセス許可の管理」を参照してください。
- Lambda クォータ について理解を深めます。ランタイムのリソース制限を決定する際に、ペイロードサイズ、ファイル記述子、および /tmp スペースが見過ごされがちです。
- 使用しなくなった Lambda 関数を削除します。削除することで、未使用の関数がデプロイパッケージサイズの制限対象として不必要にカウントされなくなります。
- イベントソースとして Amazon Simple Queue Service を使用している場合、関数の予想呼び出し時間の値が、キューの可視性タイムアウトの値を超過していないことを確認してください。これは、CreateFunctionおよびUpdateFunctionConfigurationの両方に適用されます。

- CreateFunction の場合、AWS Lambda における関数の作成プロセスは失敗します。
- UpdateFunctionConfiguration の場合、関数の呼び出しが重複する可能性があります。

関数のスケーラビリティ

- アップストリームおよびダウンストリームでのスループットの制約について理解を深めます。Lambda 関数は負荷に応じてシームレスにスケールしますが、アップストリームとダウンストリームの依存関係においてスループット能力は同じではない場合があります。関数がスケーリングできる高さを制限する必要がある場合、関数に 予約された同時実行数を設定 できます。
- スロットリング耐性を組み込みます。Lambda のスケーリングレートを超えるトラフィックが原因で同期関数にスロットリングが発生した場合、次の戦略を使用してスロットリング耐性を改善できます。
  - 「ジッターを伴うタイムアウト、再試行、バックオフ」を使用します。これらの戦略を実装すると、呼び出しの再試行がスムーズになり、Lambda が数秒以内にスケールアップしてエンドユーザーのスロットリングを最小限に抑えることができます。
  - プロビジョニングされた同時実行数 を使用します。プロビジョニングされた同時実行は、Lambda が関数に配分する事前初期化済みの実行環境の数です。Lambda は、利用可能なときにプロビジョニングされた同時実行を使用し、受信リクエストを処理します。Lambda は、必要に応じてプロビジョニングされた同時実行設定を超えて、関数をスケーリングすることもできます。プロビジョニングされた同時実行を設定すると、AWS アカウントに追加料金が請求されます。

メトリクスおよびアラーム

- Lambda 関数のメトリクスの使用 および CloudWatch アラームを使用することで、Lambda 関数コード内からメトリクスを作成または更新しないようにします。Lambda 関数のヘルスを追跡する方法としてより効率的であり、デプロイプロセスの早期で問題を把握できます。たとえば、Lambda 関数の推定される呼び出し所要時間に基づいてアラームを設定し、関数コードに起因するボトルネックやレイテンシーに対処できます。
- ログ記録のライブラリやAWS Lambda メトリクスとディメンションを活用して、アプリケーションエラー (ERR、ERROR、WARNING など) を補足します。
- AWS コスト異常の検出を使用して、アカウントの異常なアクティビティを検出します。コスト異常の検出は、機械学習を使用し、コストと使用量を継続的にモニタリングしながら誤検出アラートを最小限に抑えます。コスト異常の検出は、AWS Cost Explorer のデータを使用しますが、デー

タには最大 24 時間の遅延があります。その結果、使用を開始してから異常を検出するまでに最大 24 時間かかる場合があります。コスト異常の検出を開始するには、まず Cost Explorer にサインアップする必要があります。次に、コスト異常の検出をアクセスします。

ストリームの使用

- バッチおよびレコードの各種サイズのテストにより、関数がタスクを完了できるスピードに合わせて各イベントソースのポーリング間隔を調整します。CreateEventSourceMapping BatchSize パラメータは、各呼び出しで関数に送信できるレコードの最大数を制御します。通常、バッチサイズが大きいほど、大きなレコードセット全体での呼び出しのオーバーヘッドをより効率的に吸収し、スループットを増大できます。

  デフォルトで、Lambda はレコードが使用可能になると同時に関数を呼び出します。Lambda がイベントソースから読み取るバッチにレコードが 1 つしかない場合、Lambda は関数に 1 つのレコードしか送信しません。少数のレコードで関数を呼び出さないようにするには、バッチ処理ウィンドウを設定することで、最大 5 分間レコードをバッファリングするようにイベントソースに指示できます。関数を呼び出す前に、Lambda は、完全なバッチを収集する、バッチ処理ウィンドウの期限が切れる、またはバッチが 6 MB のペイロード制限に到達するまでイベントソースからのレコードの読み取りを継続します。詳細については、「バッチ処理動作」を参照してください。

  > **Warning**
  >
  > Lambda イベントソースマッピングは各イベントを少なくとも 1 回処理し、レコードの重複処理が発生する可能性があります。重複するイベントに関連する潜在的な問題を避けるため、関数コードを冪等にすることを強くお勧めします。詳細については、 AWS ナレッジセンターの「Lambda 関数を冪等にするにはどうすればよいですか?」を参照してください。

- シャードを追加して Kinesis ストリーム処理のスループットを増加させます。Kinesis ストリームは 1 つ以上のシャードで構成されます。Lambda は、最大で 1 つの同時呼び出しを使用して、各シャードをポーリングします。たとえば、ストリーミングに 100 個のアクティブなシャードがある場合は、最大で 100 個の Lambda 関数呼び出しが同時に実行されます。シャードの数を増やすと、直接的な結果として、Lambda 関数の同時呼び出しの最大数が増えます。また、Kinesis ストリーム処理のスループットが増える場合があります。Kinesis ストリームのシャード数を増やす場合は、データの適切なパーティションキー (「パーティションキー」を参照) を選択していることを確認し、関連レコードが同じシャードに割り当てられ、データが適切に配分されるようにします。

- Amazon CloudWatchを IteratorAge で使用し、Kinesis ストリームが処理されているかどうかを判断します。たとえば、CloudWatch アラームを最大値の 30,000 (30 秒) に設定します。

セキュリティに関するベストプラクティス

- AWS Security Hub を使用して、セキュリティのベストプラクティスに関連する AWS Lambda の使用状況をモニタリングします。Security Hub は、セキュリティコントロールを使用してリソース設定とセキュリティ標準を評価し、お客様がさまざまなコンプライアンスフレームワークに準拠できるようサポートします。Security Hub を使用して Lambda リソースを評価する方法の詳細については、「AWS Security Hub ユーザーガイド」の「AWS Lambda コントロール」を参照してください。
- Amazon GuardDuty Lambda Protection を使用し、Lambda ネットワークのアクティビティログを監視します。GuardDuty Lambda Protection は、AWS アカウント で Lambda 関数が呼び出されたときの潜在的なセキュリティ脅威を特定するために役立ちます。例えば、1 つの関数が暗号通貨関連のアクティビティに関連付けられている IP アドレスをクエリしたとします。GuardDuty は、Lambda 関数が呼び出されたときに生成されるネットワークアクティビティのログを監視します。詳細については、「Amazon GuardDuty ユーザーガイド」の「Lambda Protection」を参照してください。

AWS Lambda アクセス許可の管理

AWS Identity and Access Management (IAM) を使用して AWS Lambda でアクセス許可を管理できます。Lambda 関数を使用する際に考慮する必要があるアクセス許可には、主に次の 2 つのカテゴリがあります。

- Lambda 関数が API アクションを実行して他の AWS リソースにアクセスするために必要なアクセス許可
- 他の AWS ユーザーやエンティティが Lambda 関数にアクセスするために必要なアクセス許可

Lambda 関数は、多くの場合、他の AWS リソースにアクセスし、それらのリソースに対してさまざまな API オペレーションを実行する必要があります。例えば、Amazon DynamoDB データベースのエントリを更新してイベントに応答する Lambda 関数があるとします。この場合、その関数にはデータベースにアクセスするためのアクセス許可と、そのデータベースに項目を配置または更新するためのアクセス許可が必要です。

Lambda 関数に必要なアクセス許可は、実行ロールと呼ばれる特別な IAM ロールで定義します。このロールでは、関数が他の AWS リソースにアクセスして、イベントソースから読み取るために必要なすべてのアクセス許可を定義するポリシーをアタッチできます。すべての Lambda 関数には実行ロールが必要です。Lambda 関数はデフォルトで Amazon CloudWatch に口グ記録するため、実行ロールには、少なくとも Amazon CloudWatch へのアクセス権が必要です。`AWSLambdaBasicExecutionRole` マネージドポリシーを実行ロールにアタッチして、この要件を満たすことができます。

他の AWS アカウント、組織、サービスに、Lambda リソースにアクセスするためのアクセス許可を付与するには、いくつかのオプションがあります。

- ID ベースのポリシーを使用して、Lambda リソースへのアクセス権を他のユーザーに付与することができます。アイデンティティベースのポリシーは、ユーザーに直接適用するか、ユーザーに関連付けられているグループおよびロールに適用することができます。
- リソースベースのポリシーを使用して、Lambda リソースにアクセスするためのアクセス許可を他のアカウントや AWS サービスに付与することができます。ユーザーが Lambda リソースへのアクセスを試みた際、Lambda は、ユーザーのアイデンティティベースのポリシーと、リソースのリソースベースのポリシーの両方を認識しようとします。Amazon Simple Storage Service (Amazon S3) などの AWS のサービスが Lambda 関数を呼び出す際。Lambda はリソースベースのポリシーのみを認識しようとします。

- 属性ベースのアクセス制御 (ABAC) モデルを使用して、Lambda 関数へのアクセスを制御できます。ABAC を使用すると、Lambda 関数にタグをアタッチしたり、特定の API リクエストでタグを渡したり、リクエストを実行する IAM プリンシパルにタグをアタッチしたりすることができます。IAM ポリシーの条件要素で同じタグを指定して、関数アクセスを制御します。

AWS のベストプラクティスとして、タスクを実行するために必要なアクセス許可のみ (最小特権のアクセス許可) を付与するようにしてください。Lambda でこれを実装するには、AWS マネージドポリシーから始めることをお勧めします。これらの管理ポリシーは、そのまま使用することができますが、より制限的なポリシーを記述する際の開始点として使用することもできます。

最小特権アクセスを実現するためにアクセス許可を微調整できるように、Lambda ではポリシーに含めることができるいくつかの追加条件が用意されています。詳細については、「the section called “リソースと条件”」を参照してください。

IAM の詳細については、『IAM ユーザーガイド』を参照してください。

実行ロールを使用した Lambda 関数のアクセス許可の定義

Lambda 関数の実行ロールは、AWS サービスおよびリソースにアクセスする許可を関数に付与する AWS Identity and Access Management (IAM) ロールです。Amazon CloudWatch にログを送信するアクセス許可を持つ実行ロールを作成し、トレースデータを AWS X-Ray にアップロードすることができます。このページでは、Lambda 関数の実行ロールを作成、表示、および管理する方法について説明します。

関数を呼び出すと、Lambda が自動的に実行ロールを引き受けます。関数コード内で、実行ロールを引き受けるために `sts:AssumeRole` を手動で呼び出すことは避けてください。ユースケースでロール自体を引き受ける必要がある場合は、ロール自体を信頼できるプリンシパルとしてロールの信頼ポリシーに含める必要があります。詳細については、「IAM ユーザーガイド」の「ロールの信頼ポリシーの変更 (コンソール)」を参照してください。

Lambda が実行ロールを適切に引き受けるには、ロールの信頼ポリシーで、Lambda サービスプリンシパル (`lambda.amazonaws.com`) が信頼できるサービスとして指定されている必要があります。

トピック

- IAM コンソールでの実行ロールの作成
- AWS CLI を使用したロールの作成と管理
- Lambda 実行ロールへの最小権限アクセスを付与する
- 実行ロールのアクセス許可の表示と更新
- 実行ロールでの AWS マネージドポリシーの使用
- ソース関数 ARN を使用した関数のアクセス動作の制御

IAM コンソールでの実行ロールの作成

デフォルトでは、Lambda コンソールで関数を作成するときに、Lambda により最小限のアクセス許可で実行ロールが作成されます。具体的には、この実行ロールには `AWSLambdaBasicExecutionRole` マネージドポリシーが含まれており、Amazon CloudWatch Logs にイベントをログ記録するための基本的なアクセス許可を関数に付与します。

通常、関数には、より意味のあるタスクを実行するための追加のアクセス許可が必要です。例えば、Amazon DynamoDB データベースのエントリを更新してイベントに応答する Lambda 関数があるとします。IAM コンソールを使用して、必要なアクセス許可を持つ実行ロールを作成できます。

IAM コンソールで実行ロールを作成するには

1. IAM コンソールの [Roles (ロール)] ページを開きます。
2. [ロールの作成] を選択します。
3. [信頼されたエンティティタイプ] から、[AWS サービス] を選択します。
4. [ユースケース] で、Lambda を選択します。
5. [Next] を選択します。
6. ロールにアタッチする AWS マネージドポリシーを選択します。例えば、関数が DynamoDB にアクセスする必要がある場合は、AWSLambdaDynamoDBExecutionRole マネージドポリシーを選択します。
7. [Next] を選択します。
8. [Role name] ボックスに入力し、[Create role] を選択します。

詳しい手順については、IAM ユーザーガイドの AWS サービスのロールの作成 (コンソール) を参照してください。

実行ロールを作成したら、それを関数にアタッチします。Lambda コンソールで関数を作成するときに、以前に作成した任意の実行ロールを関数にアタッチできます。既存の関数に新しい実行ロールをアタッチする場合は、「」の手順に従います。

AWS CLI を使用したロールの作成と管理

AWS Command Line Interface (AWS CLI) を使用して実行ロールを作成するには、create-role コマンドを使用します。このコマンドを使用するときに、信頼ポリシーインラインを指定することもできます。ロールの信頼ポリシーでは、指定したプリンシパルに、ロールを引き受けるための許可を付与します。次の例では、Lambda サービスプリンシパルに自分の役割を引き受けるアクセス権限を付与します。JSON 文字列で引用符をエスケープするための要件は、シェルに応じて異なることに注意してください。

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document '{"Version":
 "2012-10-17","Statement": [{ "Effect": "Allow", "Principal": {"Service":
 "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]}'
```

また、個別の JSON ファイルを使用してロールの信頼ポリシーを定義することもできます。次の例では、`trust-policy.json` は現在のディレクトリにあるファイルです。

Example trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role --role-name lambda-ex --assume-role-policy-document file://trust-policy.json
```

以下の出力が表示されます。

```
{
    "Role": {
        "Path": "/",
        "RoleName": "lambda-ex",
        "RoleId": "AROAQFOXMPL6TZ6ITKWND",
        "Arn": "arn:aws:iam::123456789012:role/lambda-ex",
        "CreateDate": "2020-01-17T23:19:12Z",
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    },
                    "Action": "sts:AssumeRole"
                }
            ]
        }
    }
}
```

ロールにアクセス許可を追加するには、attach-policy-to-role コマンドを使用します。次のコマンドは、AWSLambdaBasicExecutionRole マネージドポリシーを lambda-ex 実行ロールにアタッチします。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

実行ロールを作成したら、それを関数にアタッチします。Lambda コンソールで関数を作成するときに、以前に作成した任意の実行ロールを関数にアタッチできます。既存の関数に新しい実行ロールをアタッチする場合は、「」の手順に従います。

Lambda 実行ロールへの最小権限アクセスを付与する

デプロイのフェーズで Lambda 関数の IAM ロールを初めて作成するときに、必要な範囲を超えたアクセス許可を付与することがあります。ベストプラクティスとしては、本番環境に関数を公開する前に、ポリシーを調整して必要なアクセス許可のみを含めるようにします。詳細については、「IAM ユーザーガイド」の「最小特権アクセス許可を適用する」を参照してください。

IAM 実行ロールポリシーに必要なアクセス許可を確認するときは、IAM Access Analyzer を使用します。IAM Access Analyzer は、指定した日付範囲で AWS CloudTrail ログを確認し、その期間中に関数が使用したアクセス許可のみを持つポリシーテンプレートを生成します。このテンプレートを使用することで、きめ細かなアクセス許可で管理ポリシーを作成し、それを IAM ロールにアタッチすることができます。これにより、特定のユースケースでロールが AWS リソースとインタラクションするために必要なアクセス許可のみを付与します。

詳細については、「IAM ユーザーガイド」の「アクセスアクティビティに基づいてポリシーを生成する」を参照してください。

実行ロールのアクセス許可の表示と更新

このトピックでは、関数の実行ロールを表示および更新する方法について説明します。

トピック

- 関数の実行ロールの表示
- 関数の実行ロールの更新

関数の実行ロールの表示

関数の実行ロールを表示するには、Lambda コンソールを使用します。

関数の実行ロールを表示するには (コンソール)

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [設定]、[アクセス権限] の順に選択します。
4. [実行ロール] で、関数の実行ロールとして現在使用されているロールを確認できます。便宜上、関数がアクセスできるすべてのリソースとアクションは、[リソースの概要] セクションで確認できます。また、ドロップダウンリストからサービスを選択して、そのサービスに関連するすべてのアクセス許可を確認することもできます。

関数の実行ロールの更新

アクセス許可は、関数の実行ロールからいつでも追加または削除できます。または、別のロールを使用するように関数を設定することもできます。関数が他のサービスまたはリソースにアクセスする必要がある場合は、必要なアクセス許可を実行ロールに追加する必要があります。

関数にアクセス許可を追加する場合は、そのコードや設定にも些細な更新を行います。これにより、(古い認証情報により実行中の) 関数のインスタンスが、強制的に停止され置き換えられます。

関数の実行ロールを更新するには、Lambda コンソールを使用できます。

関数の実行ロールを更新するには (コンソール)

1. Lambda コンソールの関数ページを開きます。
2. 関数の名前を選択します。
3. [設定]、[アクセス権限] の順に選択します。
4. [実行ロール] で、[編集] を選択します。
5. 実行ロールとして別のロールを使用するように関数を更新する場合は、[既存のロール] の下のドロップダウンメニューから新しいロールを選択します。

> **Note**
>
> 既存の実行ロール内でアクセス許可を更新する場合は、AWS Identity and Access Management (IAM) コンソールで行う必要があります。

実行ロールとして使用する新しいロールを作成する場合は、[実行ロール] で [AWS ポリシーテンプレートから新しいロールを作成する] を選択します。次に、[ロール名] で新しいロールの名前を入力し、[ポリシーテンプレート] で新しいロールにアタッチするポリシーを指定します。

6. [Save] を選択します。

実行ロールでの AWS マネージドポリシーの使用

次の AWS マネージドポリシーは、Lambda の機能を使うために必要なアクセス許可を付与します。

変更	説明	日付
AWSLambdaMSKExecutionRole - Lambda で kafka:DescribeClusterV2 がこのポリシーに追加されました。	AWSLambdaMSKExecutionRole は、Amazon Managed Streaming for Apache Kafka (Amazon MSK) クラスターからレコードを読み取り、アクセスし、Elastic Network Interface (ENI) を管理し、CloudWatch Logs に書き込むための許可を付与します。	2022 年 6 月 17 日
AWSLambdaBasicExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaBasicExecutionRole は、ログを CloudWatch にアップロードするための許可を付与します。	2022 年 2 月 14 日
AWSLambdaDynamoDBExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaDynamoDBExecutionRole は、Amazon DynamoDB ストリームからレコードを読み取り、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日

変更	説明	日付
AWSLambdaKinesisExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaKinesisExecutionRole は、Amazon Kinesis データストリームからイベントを読み取り、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日
AWSLambdaMSKExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaMSKExecutionRole は、Amazon Managed Streaming for Apache Kafka (Amazon MSK) クラスターからレコードを読み取り、アクセスし、Elastic Network Interface (ENI) を管理し、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日
AWSLambdaSQSQueueExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaSQSQueueExecutionRole は、Amazon Simple Queue Service (Amazon SQS) キューからメッセージを読み取り、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日
AWSLambdaVPCAccessExecutionRole — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSLambdaVPCAccessExecutionRole は、Amazon VPC 内の ENI を管理し、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日

変更	説明	日付
AWSXRayDaemonWriteAccess — Lambda は、このポリシーに対する変更の追跡を開始しました。	AWSXRayDaemonWriteAccess は、トレースデータを X-Ray にアップロードするための許可を付与します。	2022 年 2 月 14 日
CloudWatchLambdaInsightsExecutionRolePolicy — Lambda は、このポリシーに対する変更の追跡を開始しました。	CloudWatchLambdaInsightsExecutionRolePolicy は、CloudWatch Lambda Insights にランタイムメトリクスを書き込むための許可を付与します。	2022 年 2 月 14 日
AmazonS3ObjectLambdaExecutionRolePolicy — Lambda は、このポリシーに対する変更の追跡を開始しました。	AmazonS3ObjectLambdaExecutionRolePolicy は、Amazon Simple Storage Service (Amazon S3) オブジェクトの Lambda とやり取りし、CloudWatch Logs に書き込むための許可を付与します。	2022 年 2 月 14 日

一部の機能では、Lambda コンソールは、カスタマーマネージドポリシーの実行ロールに対して、不足しているアクセス許可を追加しようとします。これらのポリシーは数が増える可能性があります。余分なポリシーを作成しないように、機能を有効にする前に関連する AWS 管理ポリシーを実行ロールに追加します。

イベントソースマッピングを使用して関数を呼び出すと、Lambda は実行ロールを使用してイベントデータを読み出します。例えば、Kinesis のイベントソースマッピングでは、データストリームからイベントを読み出し、バッチで関数に送信します。

アカウント内でサービスがロールを引き受ける場合は、ロール信頼ポリシーに aws:SourceAccount または aws:SourceArn のグローバル条件コンテキストキーを含めることで、期待するリソースによって生成されたリクエストのみに、ロールのアクセスを制限できます。詳細については、「AWS Security Token Service のサービス間の混乱した代理の防止」を参照してください。

Lambda コンソールには、AWS マネージドポリシーに加えて、追加のユースケース用のアクセス許可を含むカスタムポリシーを作成するためのテンプレートが用意されています。Lambda コンソールで関数を作成する際、1 つ以上のテンプレートのアクセス許可を使用して新しい実行ロールを作成することを選択できます。これらのテンプレートは、設計図から関数を作成する場合、または他のサービスへのアクセスを必要とするオプションを設定する場合にも自動的に適用されます。サンプルテンプレートは、本ガイドの GitHub リポジトリから入手できます。

ソース関数 ARN を使用した関数のアクセス動作の制御

一般的に、Lambda 関数のコードは、他の AWS のサービスに対し API リクエストを送信します。これらのリクエストを行うために、Lambda は関数の実行ロールを引き受けることによって、認証情報のセットを一時的に生成します。これらの認証情報は、関数の呼び出し中に環境変数として利用できます。AWS SDK を使用している場合に、コード内で SDK の認証情報を直接提供する必要はありません。デフォルトで、認証情報プロバイダーチェーンは認証情報を設定できる各場所を順番にチェックし、最初に利用できるものを選択します。これは通常、環境変数 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、および `AWS_SESSION_TOKEN`) です。

Lambda は、リクエストが実行環境内から実行される AWS API リクエストである場合、ソース関数 ARN を認証情報コンテキストに挿入します。Lambda は、Lambda がユーザーに代わって実行環境外で実行する以下の AWS API リクエストにも、ソース関数 ARN を挿入します。

サービス	アクション	理由
CloudWatch ログ	`CreateLogGroup` ，`CreateLogStream` ，`PutLogEvents`	CloudWatch Logs ロググループにログを保存する
X-Ray	`PutTraceSegments`	X-Ray にトレースデータを送信する
Amazon EFS	`ClientMount`	関数を Amazon Elastic File System (Amazon EFS) ファイルシステムに接続する

ソース関数 ARN は、Lambda が同じ実行ロールを使用して、実行環境外でユーザーに代わって実行するその他の AWS API 呼び出しには含まれていません。このように、実行環境外で実行される API 呼び出しの例としては、以下が挙げられます。

- 環境変数を自動的に暗号化および復号化するための AWS Key Management Service (AWS KMS) の呼び出し。
- VPC 対応関数用の Elastic Network Interfaces (ENI) を作成するための Amazon Elastic Compute Cloud (Amazon EC2) の呼び出し。
- イベントソースマッピングとしてセットアップされたイベントソースから読み込むための、Amazon Simple Queue Service (Amazon SQS) などの AWS サービスの呼び出し。

認証情報コンテキストに挿入されたソース関数 ARN を使用すると、特定の Lambda 関数のコードからリソースへの呼び出しが行われたのかどうかを確認できます。これを確認するには、IAM ID ベースのポリシーまたはサービスコントロールポリシー (SCP) で `lambda:SourceFunctionArn` 条件キーを使用します。

> **Note**
>
> リソースベースのポリシーにある `lambda:SourceFunctionArn` は使用できませんでした。

ID ベースのポリシーまたは SCP でこの条件キーを使用することで、関数コードが他の AWS のサービスに対し実行する、API アクションのためのセキュリティ制御を実装できます。こういったセキュリティアプリケーションには、認証情報の漏洩の原因を特定する場合など、重要なものがいくつか含まれています。

> **Note**
>
> `lambda:SourceFunctionArn` 条件キーは、`lambda:FunctionArn` および `aws:SourceArn` 条件キーとは異なります。`lambda:FunctionArn` 条件キーは、イベントソースマッピングにのみ適用され、イベントソースから呼び出しが可能な関数を定義するのに使用されます。`aws:SourceArn` 条件キーは、Lambda 関数がターゲットリソースであるポリシーにのみ適用され、その機能を呼び出すことができる他の AWS サービスとリソースを定義するのに役立ちます。`lambda:SourceFunctionArn` 条件キーは任意の ID ベースのポリシーまたは SCP に適用して、他のリソースに対して特定の AWS API 呼び出しを行う許可を持つ特定の Lambda 関数を定義します。

ポリシーで `lambda:SourceFunctionArn` を使用するには、それを、任意の ARN 条件演算子に条件として含めます。キーの値は有効な ARN にする必要があります。

例えば、Lambda 関数のコードが特定の Amazon S3 バケットをターゲットとして、s3:PutObject 呼び出しを実行したとします。これには、Lambda 関数の 1 つだけに、対象のバケットに対する s3:PutObject アクセスを許可する必要があります。この場合、関数の実行ロールには、次のようなポリシーがアタッチされている必要があります。

Example 特定の Lambda 関数に Amazon S3 リソースへのアクセスを許可するポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

このポリシーでは、ARN が arn:aws:lambda:us-east-1:123456789012:function:source_lambda である Lambda 関数がソースの場合にのみ、s3:PutObject アクセスを許可します。このポリシーは、他の呼び出し ID に対して s3:PutObject アクセスを許可することはありません。別の関数またはエンティティが、同じ実行ロールを使用し s3:PutObject 呼び出しを行った場合も同様です。

> **Note**
>
> lambda:SourceFunctionARN 条件キーは、Lambda 関数のバージョンや関数エイリアスをサポートしていません。特定の関数バージョンまたはエイリアスの ARN を使用しても、関数には指定したアクションを実行するアクセス許可は付与されません。関数には、必ずバージョンやエイリアスのサフィックスが付いていない、修飾されていない ARN を使用してください。

SCP で lambda:SourceFunctionArn を使用することもできます。例えば、バケットへのアクセスを、単一の Lambda 関数のコードまたは特定の Amazon Virtual Private Cloud (VPC) からの呼び出しに制限したいとします。以下の SCP は、これを示したものです。

Example 特定の条件下で Amazon S3 へのアクセスを拒否するポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

このポリシーは、ARN arn:aws:lambda:*:123456789012:function:source_lambda を持つ特定の Lambda 関数からのものでない限り、または指定された VPC からのものでない限り、すべての S3 アクションを拒否します。StringNotEqualsIfExists 演算子は、aws:SourceVpc

キー がリクエストに含まれている場合にのみ、IAM に対し、この条件を処理することを指示します。これと同様に、IAM は `lambda:SourceFunctionArn` が存在する場合にのみ `ArnNotEqualsIfExists` を認識します。

他の AWS エンティティに Lambda 関数へのアクセス権を付与する

他の AWS アカウント、組織、サービスに、Lambda リソースにアクセスするためのアクセス許可を付与するには、いくつかのオプションがあります。

- ID ベースのポリシーを使用して、Lambda リソースへのアクセス権を他のユーザーに付与することができます。アイデンティティベースのポリシーは、ユーザーに直接適用するか、ユーザーに関連付けられているグループおよびロールに適用することができます。
- リソースベースのポリシーを使用して、Lambda リソースにアクセスするためのアクセス許可を他のアカウントや AWS サービスに付与することができます。ユーザーが Lambda リソースへのアクセスを試みた際、Lambda は、ユーザーのアイデンティティベースのポリシーと、リソースのリソースベースのポリシーの両方を認識しようとします。Amazon Simple Storage Service (Amazon S3) などの AWS のサービスが Lambda 関数を呼び出す際。Lambda はリソースベースのポリシーのみを認識しようとします。
- 属性ベースのアクセス制御 (ABAC) モデルを使用して、Lambda 関数へのアクセスを制御できます。ABAC を使用すると、Lambda 関数にタグをアタッチしたり、特定の API リクエストでタグを渡したり、リクエストを実行する IAM プリンシパルにタグをアタッチしたりすることができます。IAM ポリシーの条件要素で同じタグを指定して、関数アクセスを制御します。

最小特権アクセスを実現するためにアクセス許可を微調整できるように、Lambda ではポリシーに含めることができるいくつかの追加条件が用意されています。詳細については、「the section called “リソースと条件”」を参照してください。

Lambda での ID ベースの IAM ポリシーの使用

Lambda へのアクセス権をアカウントのユーザーに付与するには、AWS Identity and Access Management (IAM) のアイデンティティベースのポリシーを使用します。アイデンティティベースのポリシーは、ユーザーに直接適用するか、ユーザーに関連付けられているグループおよびロールに適用することができます。または、アカウントのロールを引き受け、Lambda リソースにアクセスするためのアクセス許可を、別のアカウントのユーザーに付与することもできます。このページでは、アイデンティティベースのポリシーを関数開発に使用する方法の例を紹介します。

Lambda は、Lambda API アクションへのアクセスを許可し、場合によっては、Lambda リソースの開発と管理に使用される他の AWS サービスへのアクセスを許可する、AWS マネージドポリシーを提供します。Lambda は、新機能がリリースされたときにユーザーがそれらにアクセスできるよう、これらのマネージドポリシーを必要に応じて更新します。

- AWSLambda_FullAccess - Lambda リソースの開発および維持に使用する Lambda アクションおよびその他の AWS サービスへのフルアクセス権を付与します。このポリシーは、以前のポリシー AWSLambdaFullAccess をスコープダウンすることによって作成されました。
- AWSLambda_ReadOnlyAccess - Lambda リソースへの読み取り専用のアクセス権を付与します。このポリシーは、以前のポリシー AWSLambdaReadOnlyAccess をスコープダウンすることによって作成されました。
- AWSLambdaRole - Lambda 関数を呼び出すアクセス許可を付与します。

AWS マネージドポリシーでは、ユーザーが変更できる Lambda 関数やレイヤーを制限することなく、API アクションへのアクセス許可を付与します。きめ細かな制御では、ユーザーのアクセス許可の範囲を制限する独自のポリシーを作成することができます。

セクション

関数にユーザーアクセス許可を付与するサンプルポリシーの作成

ID ベースのポリシーを使用して、ユーザーが Lambda 関数でオペレーションを実行することを許可します。

> **Note**
>
> コンテナーイメージとして定義された関数の場合、イメージにアクセスするためのユーザー権限は、Amazon Elastic Container Registry で設定する必要があります。例については、Amazon ECR のアクセス許可を参照してください。

範囲を制限したアクセス許可ポリシーの例を以下に示します。このポリシーにより、ユーザーは、指定されたプレフィックス (`intern-`) が名前に付き、指定された実行ロールで設定されている Lambda 関数を作成および管理することができます。

Example 関数の開発ポリシー

```
{
    "Version": "2012-10-17",
```

```
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
```

```
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
}
```

ポリシーのアクセス許可は、アクセス許可でサポートされているリソースおよび条件に基づき、ステートメントに整理されます。

- ReadOnlyPermissions - Lambda コンソールでは、関数を参照および表示する場合に次のアクセス許可を使用します。リソースパターンや条件はサポートされていません。

```
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
```

- DevelopFunctions - プレフィックス intern- が付いた関数で動作する、Lambda アクションを使用します (AddPermission および PutFunctionConcurrency は除く)。AddPermission では、関数のリソースベースのポリシーが変更されるため、セキュリティへの影響が生じる可能性があります。PutFunctionConcurrency では、関数のスケーリングキャパシティーを予約するため、他の関数にキャパシティーが奪われる可能性があります。

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- DevelopEventSourceMappings プレフィックス が付いた関数のイベントソースマッピングを管理します。intern-これらのアクションは、イベントソースマッピング上で動作しますが、条件を指定した関数を使用して制限することができます。

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- PassExecutionRole - intern-lambda-execution-role という名前のロールのみ表示して渡します。このロールは、ユーザーが IAM アクセス許可を使用して作成および管理する必要があります。PassRole は、実行ロールを関数に割り当てる際に使用します。

```
        "Action": [
            "iam:ListRolePolicies",
            "iam:ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:PassRole",
```

```
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs` - CloudWatch Logs を使用して、プレフィックス `intern-` が付いた関数のログを表示します。

```
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

このポリシーでは、他のユーザーのリソースをリスクにさらすことなく、Lambda の使用を開始することができます。これは、ユーザーが関数を他の AWS のサービスによってトリガーされる、またはそれらのサービスを呼び出すように設定できないようにします。これには、より広範は IAM 許可が必要です。また、範囲が制限されたポリシーをサポートしていないサービス (CloudWatch や X-Ray など) のアクセス許可は含まれません。メトリクスおよびトレースデータへのアクセス権をユーザーに付与するには、このようなサービスの読み取り専用ポリシーを使用します。

関数のトリガーを設定する場合は、関数を呼び出す AWS のサービスを使用するためのアクセス権が必要です。例えば、Amazon S3 トリガーを設定するには、バケット通知を管理する Amazon S3 アクションを使用するアクセス許可が必要です。このようなアクセス許可の多くは、AWSLambdaFullAccess 管理ポリシーに含まれています。サンプルポリシーは、本ガイドの GitHub リポジトリで入手できます。

レイヤーを使用するためのアクセス許可を付与するサンプルポリシーの作成

次のポリシーでは、レイヤーを作成し、それらを関数と共に使用するためのユーザーアクセス許可を付与します。リソースパターンでは、レイヤーの名前が AWS で開始している限り、ユーザーは、すべての `test-` リージョンですべてのレイヤーバージョンを使用できます。

Example レイヤーの開発ポリシー

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
```

```
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

また、関数作成時のレイヤーの使用や、lambda:Layer 条件を指定した設定を強制することもできます。たとえば、ユーザーが、他のアカウントによって発行されたレイヤーを使用できないようにすることもできます。次のポリシーでは、指定されたすべてのレイヤーをアカウント CreateFunction で作成することを求める条件を UpdateFunctionConfiguration および 123456789012 に追加します。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
```

```
}
```

条件が適用されるように、他のステートメントによって、これらのアクションに対するアクセス許可がユーザーに付与されていないことを確認します。

ID ベースのポリシーを使用したクロスアカウントアクセスの実装

以前のポリシーおよびステートメントのいずれかをロールに適用することができます。これで、別のアカウントと共有して、Lambda リソースへのアクセス権を付与することができます。ユーザーとは異なり、認証用の認証情報はロールに含まれません。その代わりに、ロールを引き受け、そのアクセス許可を使用することができるユーザーを指定する信頼ポリシーがあります。

クロスアカウントロールを使用して、Lambda アクションおよびリソースへのアクセス権を、信頼するアカウントに付与することができます。関数を呼び出すアクセス許可、またはレイヤーを使用するアクセス許可を付与する場合は、代わりにリソースベースのポリシーを使用します。

詳細については、IAM ユーザーガイドの IAM ロールを参照してください。

Lambda でのリソースベースのポリシーの使用

Lambda では、Lambda 関数およびレイヤーのための、リソースベースのアクセス許可ポリシーをサポートしています。リソースベースのポリシーを使用すれば、リソースごとに他の AWS アカウントまたは組織に使用許可を付与できます。また、リソースベースのポリシーでは、お客様に代わって関数を呼び出すことを AWS のサービスに許可することもできます。

Lambda 関数では、関数の呼び出しまたは管理を行うアクセス許可をアカウントに付与することができます。1 つのリソースベースのポリシーを使用して、アクセス許可を AWS Organizations の組織全体に付与することもできます。リソースベースのポリシーを使用して、アカウントのアクティビティに応じて関数を呼び出す AWS のサービスに対する呼び出しアクセス許可を付与することもできます。

関数のリソースベースのポリシーを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択して、[アクセス許可] を選択します。
4. [リソースベースのポリシー] まで下にスクロールし、[View policy document (ポリシードキュメントの表示)] を選択します。リソースベースのポリシーには、別のアカウントまたは AWS の

サービスが関数にアクセスしようとしたときに適用されるアクセス許可が表示されます。次の例は、アカウント 123456789012 の DOC-EXAMPLE-BUCKET という名前のバケットに対して my-function という名前の関数を呼び出すことを Amazon S3 に許可するステートメントを示しています。

Example リソースベースのポリシー

```
{
    "Version": "2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
              }
            }
        }
     ]
}
```

Lambda レイヤーでは、レイヤー全体ではなく特定のレイヤーバージョンにのみ、リソースベースのポリシーを使用できます。1 つのアカウントまたは複数のアカウントにアクセス許可を付与するポリシーに加えて、レイヤーでは 1 つの組織のすべてのアカウントにアクセス許可を付与することもできます。

> **Note**
>
> 更新できるのは、AddPermission および AddLayerVersionPermission API アクションの範囲内の、Lambda リソースのリソースベースポリシーのみです。現在、Lambda リソースのポ

リシーを JSON で作成したり、それらのアクションのパラメータにマッピングされていない条件を使用したりすることはできません。

リソースベースのポリシーは、1 つの関数、バージョン、エイリアス、レイヤーバージョンに適用されます。また、1 つ以上のサービスやアカウントにアクセス許可を付与します。複数のリソースにアクセスする、またはリソースベースのポリシーでサポートされていない API アクションを使用する、信頼されたアカウントの場合は、クロスアカウントロールを使用できます。

トピック

サポートされている API アクション

次の Lambda API アクションは、リソースベースのポリシーをサポートしています。

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig

- Invoke
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

AWS のサービスへのアクセス権を関数に付与する

AWS のサービスを使用して関数を呼び出す場合は、リソースベースのポリシーのステートメントでアクセス許可を付与します。このステートメントを、呼び出しまたは管理する関数全体に適用するか、1 つのバージョンまたはエイリアスに制限することができます。

> **Note**
>
> Lambda コンソールで関数にトリガーを追加すると、サービスでその関数を呼び出せるように、関数のリソースベースのポリシーがコンソールで更新されます。Lambda コンソールで使用できない他のアカウントやサービスにアクセス許可を付与するには、AWS CLI を使用します。

`add-permission` コマンドを使用してステートメントを追加します。最もシンプルなリソースベースのポリシーのステートメントでは、サービスで関数を呼び出すことができます。次のコマンドでは、`my-function` という名前の関数を呼び出すためのアクセス許可を Amazon SNS に付与します。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id sns \
    --principal sns.amazonaws.com --output text
```

次のような出力が表示されます。

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

これにより、Amazon SNS は関数の `lambda:Invoke` API を呼び出すことができますが、呼び出しをトリガーする Amazon SNS トピックは制限されません。関数が特定のリソースからのみ呼び出せるようにするには、`source-arn` オプションでリソースの Amazon リソースネーム (ARN) を指定します。次のコマンドは、Amazon SNS に、`my-topic` という名前のトピックをサブスクリプションするための、関数を呼び出すことのみ許可します。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id sns-my-topic \
    --principal sns.amazonaws.com --source-arn arn:aws:sns:us-east-2:123456789012:my-
topic
```

一部のサービスでは、他のアカウントで関数を呼び出すことができます。アカウント ID を含むソース ARN を指定した場合は問題ではありません。ただし、Amazon S3 では、ソースは、ARN にアカウント ID が含まれないバケットになります。バケットを削除すると、別のアカウントで同じ名前のバケットが作成される可能性があります。アカウントのリソースでのみ関数を呼び出せるように、アカウント ID を指定して `source-account` オプションを使用します。

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id s3-account \
    --principal s3.amazonaws.com --source-arn arn:aws:s3:::DOC-EXAMPLE-BUCKET --source-
account 123456789012
```

関数へのアクセス権を組織に付与する

AWS Organizations の組織にアクセス許可を付与するには、組織 ID を `principal-org-id` として指定します。次の AWS CLI コマンド AddPermission によって、`o-a1b2c3d4e5f` の組織内のすべてのユーザーに呼び出しアクセス許可が付与されます。

```
aws lambda add-permission --function-name example \
```

```
    --statement-id PrincipalOrgIDExample --action lambda:InvokeFunction \
    --principal * --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> このコマンドでは、Principal は * です。これにより、組織 o-a1b2c3d4e5f 内のすべてのユーザーが関数呼び出しアクセス許可を取得します。Principal として AWS アカウントまたはロールを指定した場合、そのプリンシパルだけが関数呼び出しアクセス許可を取得します。ただし、それらが o-a1b2c3d4e5f 組織の一部である場合に限ります。

このコマンドで、次のようなリソースベースのポリシーが作成されます。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

詳細については、「AWS Identity and Access Management ユーザーガイド」の「aws:PrincipalOrgID」を参照してください。

他のアカウントへのアクセス権を関数に付与する

別の AWS アカウントへのアクセス許可を付与するには、アカウント ID を principal として指定します。次の例では、111122223333 エイリアスを使用して my-function を呼び出すアクセス許可をアカウント prod に付与します。

```
aws lambda add-permission --function-name my-function:prod --statement-id xaccount --
action lambda:InvokeFunction \
    --principal 111122223333 --output text
```

次のような出力が表示されます。

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-2:123456789012:function:my-function"}
```

リソースベースのポリシーは、他のアカウントに対して関数へのアクセス許可を付与しますが、そのアカウントのユーザーに対してはユーザーに付与済みのアクセス許可を超える許可を付与しません。他のアカウントのユーザーが Lambda API を使用するには、対応するユーザー権限が必要です。

別のアカウントのユーザーまたはロールへのアクセスを制限するには、ID の完全な ARN をプリンシパルとして指定します。例えば、`arn:aws:iam::123456789012:user/developer` と指定します。

エイリアスでは、他のアカウントが呼び出すことができるバージョンを制限します。この方法では、他のアカウントは、エイリアスを関数 ARN を含める必要があります。

```
aws lambda invoke --function-name arn:aws:lambda:us-west-2:123456789012:function:my-
function:prod out
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

その後、関数の所有者は、新しいバージョンを参照するようにエイリアスを更新できます。これにより、呼び出し元で関数を呼び出す方法を変更する必要がなくなります。また、他のアカウントは、新しいバージョンを使用するようにコードを変更する必要もなくなります。エイリアスに関連付けられた関数のバージョンを呼び出すアクセス許可が付与されるだけです。

既存の関数で動作するほとんどの API アクションにクロスアカウントアクセスを付与できます。例えば、アカウントによるエイリアスのリストの取得を許可する `lambda:ListAliases`、または関数コードのダウンロードを許可する `lambda:GetFunction` へのアクセス権を付与することができ

ます。各アクセス許可を個別に追加するか、lambda:* を使用して、指定された関数のすべてのアクションに対するアクセス権を付与します。

他のアカウントに複数の関数に対するアクセス許可、または関数で実行しないアクションに対するアクセス許可を付与するには、IAM ロールを使用することをお勧めします。

他のアカウントへのアクセス権をレイヤーに付与する

レイヤーの使用に関するアクセス許可を別のアカウントに付与するには、add-layer-version-permission コマンドを使用して、ステートメントをレイヤーバージョンのアクセス許可ポリシーに追加します。アクセス許可は、各ステートメントで、1 つのアカウント、すべてのアカウント、または組織に付与することができます。

以下の例では、アカウント 111122223333 に bash-runtime レイヤーのバージョン 2 へのアクセス許可を付与します。

```
aws lambda add-layer-version-permission --layer-name bash-runtime --statement-id
 xaccount \
--action lambda:GetLayerVersion --principal 111122223333 --version-number 2 --output
 text
```

次のような出力が表示されます。

```
e210ffdc-e901-43b0-824b-5fcd0dd26d16    {"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

アクセス許可は、単一レイヤーバージョンにのみ適用されます。新しいレイヤーバージョンを作成するたびに、このプロセスを繰り返します。

組織内のすべてのアカウントにアクセス許可を付与するには、organization-id オプションを使用します。以下の例では、バージョン 3 のレイヤーを使用するアクセス許可を、組織のすべてのアカウントに付与します。

```
aws lambda add-layer-version-permission --layer-name my-layer \
    --statement-id engineering-org --version-number 3 --principal '*' \
    --action lambda:GetLayerVersion --organization-id o-t194hfs8cz --output text
```

次のような出力が表示されます。

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

アクセス許可をすべての AWS アカウントに付与するには、プリンシパルに * を使用して、組織 ID を除外します。複数のアカウントまたは組織が対象の場合は、複数のステートメントを追加する必要があります。

リソースベースのポリシーのクリーンアップ

関数のリソースベースのポリシーを表示するには、get-policy コマンドを使用します。

```
aws lambda get-policy --function-name my-function --output text
```

次のような出力が表示されます。

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}      7c681fc9-
b791-4e91-acdf-eb847fdaa0f0
```

バージョンおよびエイリアスでは、バージョン番号またはエイリアスを関数名に追加します。

```
aws lambda get-policy --function-name my-function:PROD
```

関数からアクセス許可を削除するには、remove-permission を使用します。

```
aws lambda remove-permission --function-name example --statement-id sns
```

レイヤーでのアクセス許可を表示するには、get-layer-version-policy コマンドを使用します。

```
aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output
 text
```

次のような出力が表示されます。

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

remove-layer-version-permission を使用して、ポリシーからステートメントを削除します。

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 --
statement-id engineering-org
```

Lambda での属性ベースのアクセスコントロールの使用

属性ベースのアクセスコントロール (ABAC) では、タグを使用して Lambda 関数へのアクセスを制御できます。これらのタグは、Lambda 関数にをアタッチすることや、特定の API リクエストに渡すことができます。あるいは、リクエストを実行する AWS Identity and Access Management (IAM) プリンシパルにアタッチすることも可能です。AWS による属性ベースアクセスの付与の詳細については、「IAM ユーザーガイド」の「タグを使用した AWS リソースへのアクセスの制御」を参照してください。

ABAC を使用すると、IAM ポリシーで Amazon リソースネーム (ARN) または ARN パターンを指定しなくても、最小特権を付与することができます。代わりに、IAM ポリシーの条件要素で、アクセスを制御するためのタグを指定します。新しい関数を作成する際の、IAM ポリシーの更新が必要なくなるため、ABAC を使用するとスケーリングが容易になります。代わりに、新しい関数には、アクセスを制御するためのタグを追加します。

Lambda では、これらのタグは関数レベルで機能します。これらのタグは、レイヤー、コードの署名設定、またはイベントソースマッピングではサポートされていません。関数にタグを付けると、それらのタグは対象の関数に関連付けられているすべてのバージョンとエイリアスに適用されます。関数のタグ付けの方法については、「Lambda 関数でのタグの使用」を参照してください。

関数のアクションは、以下の条件キーを使用して制御できます。

- aws:ResourceTag/tag-key: Lambda 関数にアタッチされているタグに基づいてアクセスを制御します。
- aws:RequestTag/tag-key: 新しい関数の作成時などに、リクエスト内のタグを要求します。
- aws:PrincipalTag/tag-key: IAM プリンシパル (リクエストを行っているユーザー) が実行できる操作内容を、IAM ユーザーまたはロールにアタッチされたタグに基づき制御します。
- aws:TagKeys: リクエストで特定のタグキーを使用できるかどうかを制御します。

ABAC をサポートする Lambda アクションの完全なリストについては、「サポートされている関数アクション」にアクセスして、表にある [Condition] (条件) 列を参照してください。

次の手順は、ABAC を使用してアクセス許可を設定する方法の一例です。この例のシナリオでは、IAM アクセス許可ポリシーを 4 つ作成しています。その後、これらのポリシーを新しい IAM ロールにアタッチします。最後に、IAM ユーザーを作成し、そのユーザーに、新しいロールを引き受けるためのアクセス許可を付与します。

トピック

前提条件

Lambda の実行ロールが必要です。このロールは、IAM アクセス許可の付与、および Lambda 関数の作成を行う際に使用します。

ステップ 1: 新しい関数のタグを要求する

Lambda で ABAC を使用する場合、すべての関数にタグを付けるようにするのがベストプラクティスです。これにより、ABAC での許可ポリシーが期待どおりに機能することが保証されます。

次の例のような IAM ポリシーを作成します。このポリシーでは、aws:RequestTag/tag-key、aws:ResourceTag/tag-key、および aws:TagKeys 条件キーにより、新しい関数と、その関数を作成する IAM プリンシパルの両方に、`project` タグが付けられていることを要求しています。`ForAllValues` 修飾子により、`project` を唯一許可されているタグとして指定しています。`ForAllValues` 修飾子含めない場合、ユーザーは `project` を渡すことで他のタグを関数に追加できるようになります。

Example – 新しい関数のタグを要求する

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "lambda:CreateFunction",
      "lambda:TagResource"
    ],
    "Resource": "arn:aws:lambda:*:*:function:*",
    "Condition": {
      "StringEquals": {
        "aws:RequestTag/project": "${aws:PrincipalTag/project}",
        "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
      },
      "ForAllValues:StringEquals": {
        "aws:TagKeys": "project"
      }
    }
  }
}
```

ステップ 2: Lambda 関数と IAM プリンシパルにアタッチされたタグに基づいてアクションを許可する

aws:ResourceTag/tag-key 条件キーを使用して 2 番目の IAM ポリシーを作成し、プリンシパルのタグが関数にアタッチされているタグと一致することを要求します。次のポリシー例は、project タグが付けられたプリンシパルに対し、project タグが付けられた関数を呼び出すことを許可します。他のタグが関数に付けられている場合、このアクションは拒否されます。

Example – 関数と IAM プリンシパル間でタグの一致を要求する

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction",
        "lambda:GetFunction"
      ],
```

```
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        }
      }
    }
  ]
}
```

ステップ 3: リスト作成のためのアクセス許可を付与する

プリンシパルに対し、Lambda 関数と IAM ロールのリスト作成を許可するポリシーを作成します。これによりプリンシパルは、すべての Lambda 関数と IAM ロールをコンソールに表示でき、API アクション呼び出時に認識できるようになります。

Example – Lambda と IAM に関するリスト作成を許可する

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllResourcesLambdaNoTags",
      "Effect": "Allow",
      "Action": [
        "lambda:GetAccountSettings",
        "lambda:ListFunctions",
        "iam:ListRoles"
      ],
      "Resource": "*"
    }
  ]
}
```

ステップ 4: IAM のアクセス許可を付与する

iam:PassRole を許可するポリシーを作成します。このアクセス許可は、関数に実行ロールを割り当てる際に必要となります。次のポリシー例にあるサンプルの ARN は、実際のLambda 実行ロールの ARN に置き換えます。

> **Note**
>
> iam:PassRole アクションでポリシーの ResourceTag 条件キーを使用しないでください。IAM ロールのタグを使用して、そのロールを渡すことができるユーザーへのアクセスを制御することはできません。サービスにロールを渡すために必要となるアクセス許可については、「AWS のサービスにロールを渡すアクセス許可をユーザーに付与する」を参照してください。

Example – 実行ロールを渡すためのアクセス許可を付与する

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
    }
  ]
}
```

ステップ 5: IAM ロールを作成する

アクセス許可を委任するためには、ロールを使用することがベストプラクティスです。abac-project-role という IAM ロールを作成します。

- [ステップ 1: 信頼されたエンティティを選択] で、[AWS アカウント]、[このアカウント] の順に選択します。
- [Step 2: Add permissions] (ステップ 2: アクセス許可を追加する) で、前のステップで作成した 4 つの IAM ポリシーをアタッチします。
- [Step 3: Name, review, and create] (ステップ 3: 名前、確認、および作成) で、[Add tag] (タグを追加) を選択します。[Key] (キー) に「project」と入力します。ここでは、値は入力しません。

ステップ 6: IAM ユーザーを作成する

abac-test-user という IAM ユーザーを作成します。[Set permissions] (アクセス許可の設定) セクションで、[Attach existing policies directly] (既存のポリシーを直接アタッチ) を選択し、次に [Create policy] (ポリシーを作成) を選択します。ポリシーの定義を以下のように入力します。*111122223333* の部分は、自分の AWS アカウントID に置き換えます。このポリシーでは、abac-project-role を引き受けることを abac-test-user に対し許可します。

Example – ABAC ロールを引き受けることを、IAM ユーザーに対し許可する

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
  }
}
```

ステップ 7: アクセス許可をテストする

1. AWS コンソールに、abac-test-user としてサインインします。詳細については、「IAM ユーザーとしてサインインする」を参照してください。
2. abac-project-role ロールに切り替えます。詳細については、「ロールの切り替え (コンソール)」を参照してください。
3. Lambda 関数を作成します。
   - [Permissions] (許可) で [Change default execution role] (デフォルトの実行ロールの変更) を選択した後、[Execution role] (実行ロール) で [Use an existing role] (既存のロールを使用する) を選択します。ステップ 4: IAM のアクセス許可を付与する で使用したものと同じ実行ロールを選択します。
   - [Advanced settings] (詳細設定) で [Enable tags] (タグを有効化) を選択した上で、[Add new tag] (新しいタグを追加) を選択します。[Key] (キー) に「project」と入力します。ここでは、値は入力しません。
4. 関数をテストします。
5. 2 つ目の Lambda 関数を作成し、異なるタグ (例: environment) を追加します。通常、この操作は失敗します。ステップ 1: 新しい関数のタグを要求する で作成した ABAC ポリシーでは、project タグが付いた関数を作成することのみをプリンシパルに許可しているためです。

6. タグを付けずに 3 つ目の関数を作成します。通常、この操作も失敗します。ステップ 1: 新しい関数のタグを要求する で作成した ABAC ポリシーでは、タグなしの関数を作成することをプリンシパルに許可していないためです。

この認証戦略により、それぞれの新しいユーザーに新しいポリシーを作成することなく、アクセスの制御が可能になります。新しいユーザーにアクセス権を付与する際は、割り当てられたプロジェクトに対応するロールを引き受けるための、アクセス許可を付与するだけですみます。

ステップ 8: リソースをクリーンアップする

IAM ロールを削除するには

1. IAM コンソールのロールページを開きます。
2. ステップ 5 で作成したロールを選択します。
3. [削除] を選択します。
4. 削除を確認するには、テキスト入力フィールドにロール名を入力します。
5. [削除] を選択します。

IAM ユーザーを削除するには

1. IAM コンソールでユーザーページを開きます。
2. ステップ 6 で作成した IAM ユーザーを選択します。
3. [削除] を選択します。
4. 削除を確認するには、テキスト入力フィールドにユーザー名を入力します。
5. [ユーザーの削除] を選択します。

Lambda 関数を削除するには

1. Lambda コンソールの関数ページを開きます。
2. 作成した関数を選択します。
3. [アクション] で、[削除] を選択します。
4. テキスト入力フィールドに **`delete`** と入力し、[Delete] (削除) を選択します。

ポリシーのリソースセクションと条件セクションの微調整

AWS Identity and Access Management (IAM) ポリシーでリソースと条件を指定することで、ユーザーのアクセス許可の範囲を制限できます。ポリシーの各アクションが、それぞれの動作によって異なるリソースタイプと条件タイプの組み合わせをサポートします。

各 IAM ポリシーステートメントによって、リソースで実行されるアクションに対するアクセス許可が付与されます。アクションが名前の付いたリソースで動作しない場合、またはすべてのリソースに対してアクションを実行するアクセス許可を付与した場合、ポリシー内のリソースの値はワイルドカード (*) になります。多くのアクションでは、リソースの Amazon リソースネーム (ARN)、または複数のリソースに一致する ARN パターンを指定することによって、ユーザーによる変更が可能なリソースを制限できます。

リソース別にアクセス許可を制限するには、ARN 別にリソースを指定します。

Lambda リソース ARN 形式

- 関数 - `arn:aws:lambda:us-west-2:123456789012:function:my-function`
- 関数のバージョン - `arn:aws:lambda:us-west-2:123456789012:function:my-function:1`
- 関数のエイリアス - `arn:aws:lambda:us-west-2:123456789012:function:my-function:TEST`
- イベントソースマッピング - `arn:aws:lambda:us-west-2:123456789012:event-source-mapping:fa123456-14a1-4fd2-9fec-83de64ad683de6d47`
- レイヤー - `arn:aws:lambda:us-west-2:123456789012:layer:my-layer`
- レイヤーバージョン - `arn:aws:lambda:us-west-2:123456789012:layer:my-layer:1`

例えば、以下のポリシーでは、`my-function` という名前の関数を米国西部 (オレゴン) AWS リージョンで呼び出すことを、AWS アカウント 123456789012 のユーザーに対し許可します。

Example 関数ポリシーを呼び出す

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

アクションの識別子 (lambda:InvokeFunction) が、API オペレーション (Invoke) と異なる特別なケースです。その他のアクションのアクションの識別子は、lambda: プレフィックスがついたオペレーション名です。

セクション

- ポリシーの条件セクションについて
- ポリシーのリソースセクションで関数を参照する
- サポートされている関数アクション
- サポートされているイベントソースマッピングアクション
- サポートされているレイヤーアクション

ポリシーの条件セクションについて

条件は、アクションが許可されているかどうかを判断するために追加の口ジックを適用するオプションのポリシー要素です。すべてのアクションでサポートされている共通の条件に加えて、Lambda は、一部のアクションが使用する追加パラメータの値を制限するための条件タイプも定義します。

例えば、lambda:Principal 条件では、関数のリソースベースのポリシーへの呼び出しアクセス権をユーザーが付与できる、サービスまたはアカウントを制限できます。次のポリシーを使用すると、ユーザーは、test という名前の関数を呼び出すアクセス許可を Amazon Simple Notification Service (Amazon SNS) トピックに付与できます。

Example 関数ポリシーのアクセス許可を管理する

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
```

```
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

この条件では、プリンシパルが Amazon SNS で、別のサービスやアカウントでないことが必要です。リソースパターンでは、関数名が `test` で、バージョン番号またはエイリアスが含まれている必要があります。例えば、`test:v1` と指定します。

Lambda および他の AWS のサービスでのリソースと条件の詳細については、「サービス認証リファレンス」の「AWS のサービスのアクション、リソース、および条件キー」を参照してください。

ポリシーのリソースセクションで関数を参照する

Amazon リソースネーム (ARN) を使用して、ポリシーステートメント内で Lambda 関数を参照します。関数 ARN の形式は、関数全体を参照する (修飾)か、関数のバージョンやエイリアスを参照する (非修飾) かに応じて異なります。

Lambda API コールを行うとき、ユーザーは、GetFunction `FunctionName` パラメータにあるバージョン ARN またはエイリアス ARN を渡すか、 GetFunction `Qualifier` パラメータにある値を設定することで、バージョンまたはエイリアスを指定することができます。Lambda は、IAM ポリシー内のリソース要素を、API コールで渡された `FunctionName` と `Qualifier` の両方と比較することによって、認可の決定を行います。一致しないものがある場合、Lambda はそのリクエストを拒否します。

関数に対するアクションを許可するか拒否するかにかかわらず、期待どおりの結果を得るには、ポリシーステートメントで正しい関数の ARN タイプを使用する必要があります。例えば、ポリシーが非修飾 ARN を参照する場合、Lambda は非修飾 ARN を参照するリクエストを受け入れますが、修飾 ARN を参照するリクエストは拒否します。

> **Note**
>
> アカウント ID を照合するためにワイルドカード文字 (*) を使用することはできません。認められる構文の詳細については、「IAM ユーザーガイド」の「IAM JSON ポリシーリファレンス」を参照してください。

Example 非修飾 ARN の呼び出しの許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction"
        }
    ]
}
```

ポリシーが特定の修飾 ARN を参照する場合、Lambda はその ARN を参照するリクエストを受け入れますが、非修飾 ARN や別の修飾 ARN (`myFunction:2` など) を参照するリクエストは拒否します。

Example 特定の修飾 ARN の呼び出しの許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

ポリシーが `:*` を使用して任意の修飾 ARN を参照する場合、Lambda は修飾 ARN ならどれでも受け入れますが、非修飾 ARN を参照するリクエストは拒否します。

Example 任意の修飾 ARN の呼び出しの許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

ポリシーが * を使用して任意の ARN を参照する場合、Lambda はすべての修飾 ARN と非修飾 ARN を受け入れます。

Example 任意の修飾または非修飾 ARN の呼び出しの許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction*"
        }
    ]
}
```

サポートされている関数アクション

以下の表で説明されているように、関数を操作するアクションは、関数、バージョン、またはエイリアス ARN によって特定の関数に制限することができます。リソース制限をサポートしていないアクションは、すべてのリソース (*) に付与されます。

関数のアクション

アクション	リソース	条件
AddPermission	関数	lambda:Principal
RemovePermission		aws:ResourceTag/${TagKey}

アクション	リソース	条件
	関数のバージョン 関数のエイリアス	lambda:FunctionUrlAuthType
Invoke アクセス許可: lambda:InvokeFunction	関数 関数のバージョン 関数のエイリアス	aws:ResourceTag/${TagKey} lambda:EventSourceToken
CreateFunction	機能	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys

アクション	リソース	条件
UpdateFunctionConfiguration	関数	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/${TagKey}

アクション	リソース	条件
CreateAlias	機能	aws:ResourceTag/${TagKey}
DeleteAlias		
DeleteFunction		
DeleteFunctionCodeSigningConfig		
DeleteFunctionConcurrency		
GetAlias		
GetFunction		
GetFunctionCodeSigningConfig		
GetFunctionConcurrency		
GetFunctionConfiguration		
GetPolicy		
ListProvisionedConcurrencyConfigs		
ListAliases		
ListTags		
ListVersionsByFunction		
PublishVersion		
PutFunctionCodeSigningConfig		
PutFunctionConcurrency		
UpdateAlias		
UpdateFunctionCode		

アクション	リソース	条件
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	機能 関数のエイリアス	lambda:FunctionUrlAuthType lambda:FunctionArn aws:ResourceTag/${TagKey}
ListFunctionUrlConfigs	機能	lambda:FunctionUrlAuthType
DeleteFunctionEventInvokeConfig GetFunctionEventInvokeConfig ListFunctionEventInvokeConfigs PutFunctionEventInvokeConfig UpdateFunctionEventInvokeConfig	機能	aws:ResourceTag/${TagKey}
DeleteProvisionedConcurrencyConfig GetProvisionedConcurrencyConfig PutProvisionedConcurrencyConfig	関数のエイリアス 関数のバージョン	aws:ResourceTag/${TagKey}
GetAccountSettings ListFunctions	*	なし
TagResource	機能	aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys
UntagResource	機能	aws:ResourceTag/${TagKey} aws:TagKeys

サポートされているイベントソースマッピングアクション

イベントソースマッピングでは、アクセス許可に関する削除および更新を、特定のイベントソースに制限することができます。lambda:FunctionArn 条件では、ユーザーが呼び出すイベントソースを設定できる関数を制限することができます。

これらのアクションでは、リソースはイベントソースマッピングであるため、Lambda では、イベントソースマッピングで呼び出される関数に基づき、アクセス許可を制限することができる条件を提供します。

イベントソースのマッピングアクション

アクション	リソース	条件
DeleteEventSourceMapping UpdateEventSourceMapping	イベントソースマッピング	lambda:FunctionArn
CreateEventSourceMapping GetEventSourceMapping	*	lambda:FunctionArn
ListEventSourceMappings	*	なし

サポートされているレイヤーアクション

レイヤーアクションでは、関数を使用してユーザーが管理または使用できるレイヤーを制限できます。レイヤーの使用とアクセス許可に関連するアクションはレイヤーのバージョン、PublishLayerVersion はレイヤー名に影響します。ワイルドカードとあわせていずれかを使用して、ユーザーが操作できるレイヤーを名前で制限することができます。

> **Note**
>
> GetLayerVersion アクションは GetLayerVersionByArn もカバーします。Lambda は、GetLayerVersionByArn を IAM アクションとしてサポートしていません。

レイヤーのアクション

アクション	リソース	条件
AddLayerVersionPermission RemoveLayerVersionPermission GetLayerVersion GetLayerVersionPolicy DeleteLayerVersion	レイヤーバージョン	なし
ListLayerVersions PublishLayerVersion	Layer	なし
ListLayers	*	なし

AWS Lambda のセキュリティ

AWS ではクラウドセキュリティが最優先事項です。セキュリティを最も重視する組織の要件を満たすために構築された AWS のデータセンターとネットワークアーキテクチャは、お客様に大きく貢献します。

セキュリティは、AWS と顧客の間の責任共有です。責任共有モデルでは、この責任がクラウドのセキュリティおよびクラウド内のセキュリティとして説明されています。

- クラウドのセキュリティ - AWS は、AWS クラウドで AWS のサービスを実行するインフラストラクチャを保護する責任を負います。また、AWS は、使用するサービスを安全に提供します。AWS コンプライアンスプログラムの一環として、サードパーティーの監査が定期的にセキュリティの有効性をテストおよび検証しています。AWS Lambda に適用されるコンプライアンスプログラムの詳細については、「コンプライアンスプログラムによる対象範囲内の AWS のサービス」を参照してください。
- クラウド内のセキュリティ - ユーザーの責任は、使用する AWS のサービスに応じて異なります。また、お客様は、お客様のデータの機密性、企業の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントは、Lambda を使用する際に責任共有モデルを適用する方法を理解するのに役立ちます。以下のトピックでは、セキュリティとコンプライアンスの目的を達成するために Lambda を設定する方法を示します。また、Lambda リソースのモニタリングや保護に役立つ、他の AWS のサービスの使用方法についても説明します。

Lambda アプリケーションに対するセキュリティプリンシパルの適用の詳細については、Serverless Land の「セキュリティ」を参照してください。

トピック

AWS Lambda でのデータ保護

AWS責任共有モデルは、AWS Lambda でのデータ保護に適用されます。このモデルで説明されているように、AWS は、AWS クラウド のすべてを実行するグローバルインフラストラクチャを保護するがあります。このインフラストラクチャでホストされているコンテンツに対する管理を維持する責任はユーザーにあります。また、使用する AWS のサービスのセキュリティ設定と管理タスクもユーザーの責任となります。データプライバシーの詳細については、「データプライバシーのよくある質問」を参照してください。欧州でのデータ保護の詳細については、「AWS セキュリティブログ」に投稿された「AWS 責任共有モデルおよび GDPR」のブログ記事を参照してください。

データを保護するため、AWS アカウント認証情報を保護し、AWS IAM Identity Center または AWS Identity and Access Management (IAM) を使用して個々のユーザーをセットアップすることをお勧めします。こうすると、それぞれのジョブを遂行するために必要なアクセス許可のみを各ユーザーに付与できます。また、以下の方法でデータを保護することをお勧めします。

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。TLS 1.2 が必須です。TLS 1.3 が推奨されます。
- AWS CloudTrail で API とユーザーアクティビティロギングをセットアップします。
- AWS のサービス内でデフォルトである、すべてのセキュリティ管理に加え、AWS 暗号化ソリューションを使用します。
- Amazon Macie などの高度なマネージドセキュリティサービスを使用します。これらは、Amazon S3 に保存されている機密データの検出と保護を支援します。
- コマンドラインインターフェイスまたは API により AWS にアクセスするときに FIPS 140-2 検証済み暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「連邦情報処理規格 (FIPS) 140-2」を参照してください。

お客様の E メールアドレスなどの機密情報やセンシティブ情報は、タグや名前フィールドなどの自由形式のフィールドに配置しないことを強くお勧めします。これには、コンソール、API、AWS CLI、または AWS SDK を使用して Lambda またはその他の AWS のサービスを使用する状況が含まれます。名前に使用する自由記述のテキストフィールドやタグに入力したデータは、課金や診断ログに使用される場合があります。外部サーバーへの URL を提供する場合は、そのサーバーへのリクエストを検証するための認証情報を URL に含めないように強くお勧めします。

セクション

- 転送時の暗号化

- 保管中の暗号化

転送時の暗号化

Lambda API エンドポイントでは、HTTPS 経由の安全な接続のみがサポートされます。Lambda リソースを AWS Management Console、AWS SDK、または Lambda API を使用して管理する場合、すべての通信は Transport Layer Security (TLS) で暗号化されます。API エンドポイントの完全なリストについては、AWS 全般のリファレンス の「AWSリージョンとエンドポイント」を参照してください。

関数をファイルシステムに接続すると、Lambda では、すべての接続で転送時の暗号化が使用されます。詳細については、Amazon Elastic File System ユーザーガイド の「Amazon EFS でのデータの暗号化」を参照してください。

環境変数を使用する場合、コンソール暗号化ヘルパーを有効にして、クライアント側の暗号化で転送中の環境変数を保護できます。詳細については、「Lambda 環境変数の保護」を参照してください。

保管中の暗号化

Lambda は、保存時に環境変数を常に暗号化します。デフォルトでは、環境変数は、Lambda がアカウントに作成する AWS KMS key を使用して暗号化されます。この AWS マネージドキー は、`aws/lambda` という名前です。

関数ごとに Lambda を設定して、デフォルトの AWS マネージドキー ではなく、カスタマー管理のキーを使用して環境変数を暗号化することもできます。詳細については、「Lambda 環境変数の保護」を参照してください。

Lambda は、デプロイパッケージやレイヤーアーカイブを含め、Lambda にアップロードされるファイルを常に暗号化します。

Amazon CloudWatch Logs と AWS X-Ray は、デフォルトでデータを暗号化し、カスタマーマネージドキーを使用するように設定できます。詳細については、CloudWatch Logs での口グデータの暗号化およびAWS X-Ray でのデータ保護を参照してください。

AWS Lambda 向けの Identity and Access Management

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御するために役立つ AWS のサービス です。IAM 管理者は、誰を認証 (サインイン) し、誰に

Lambda リソースの使用を許可する (アクセス許可を持たせる) かを制御します。IAM は、追加費用なしで使用できる AWS のサービスです。

トピック

対象者

AWS Identity and Access Management (IAM) の用途は、Lambda で行う作業によって異なります。

サービスユーザー – ジョブを実行するために Lambda サービスを使用する場合は、管理者から必要なアクセス許可と認証情報が与えられます。作業を実行するためにさらに多くの Lambda 機能を使用するとき、追加のアクセス許可が必要になる場合があります。アクセスの管理方法を理解すると、管理者から適切なアクセス許可をリクエストするのに役に立ちます。Lambda の機能にアクセスできない場合は、AWS Lambda ID とアクセスのトラブルシューティング を参照してください。

サービス管理者 - 社内の Lambda リソースを担当している場合は、通常、Lambda に完全にアクセスすることができます。サービスのユーザーがどの Lambda 機能やリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を確認して、IAM の基本概念を理解してください。ご自分の会社で Lambda を使って IAM をいかに使用すべきかの詳細については、AWS Lambda と IAM の連携方法 を参照してください。

IAM 管理者 – IAM 管理者は、Lambda へのアクセスを管理するポリシーの、作成方法の詳細を確認する場合があります。IAM で使用できる Lambda アイデンティティベースのポリシーの例を表示するには、AWS Lambda のアイデンティティベースのポリシーの例 を参照してください。

アイデンティティを使用した認証

認証とは、アイデンティティ認証情報を使用して AWSにサインインする方法です。ユーザーは、AWS アカウントのルートユーザーもしくは IAM ユーザーとして、または IAM ロールを引き受けることによって、認証を受ける (AWS にサインインする) 必要があります。

ID ソースから提供された認証情報を使用して、フェデレーティッドアイデンティティとして AWS にサインインできます。 AWS IAM Identity Centerフェデレーティッドアイデンティティの例としては、(IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook の認証情報などがあります。フェデレーティッドアイデンティティとしてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーションを使用して AWSにアクセスする場合、間接的にロールを引き受けることになります。

ユーザーのタイプに応じて、AWS Management Console または AWS アクセスポータルにサインインできます。AWS へのサインインの詳細については、『AWS サインイン ユーザーガイド』の「AWS アカウント にサインインする方法」を参照してください。

プログラムで AWS にアクセスする場合、AWS は Software Development Kit (SDK) とコマンドラインインターフェイス (CLI) を提供し、認証情報でリクエストに暗号で署名します。AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。リクエストに署名する推奨方法の使用については、『IAM ユーザーガイド』の「AWS API リクエストの署名」を参照してください。

使用する認証方法を問わず、追加のセキュリティ情報の提供が求められる場合もあります。例えば、AWS では、アカウントのセキュリティ強化のために多要素認証 (MFA) の使用をお勧めしています。詳細については、「AWS IAM Identity Center ユーザーガイド」の「Multi-factor authentication (多要素認証)」 および「IAM ユーザーガイド」の「AWS での多要素認証 (MFA) の使用」を参照してください。

AWS アカウントのルートユーザー

AWS アカウントを作成する場合は、そのアカウントのすべての AWS のサービスとリソースに対して完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。このアイデンティティは AWS アカウントのルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることによってアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、「IAM ユーザーガイド」の「ルートユーザー認証情報が必要なタスク」を参照してください。

フェデレーティッド ID

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに対し、ID プロバイダーとのフェデレーションを使用して、一時的な認証情報の使用により、AWS のサービスにアクセスすることを要求します。

フェデレーテッドアイデンティティ は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、アイデンティティセンターディレクトリのユーザーか、または ID ソースから提供された認証情報を使用して AWS のサービス にアクセスするユーザーです。フェデレーテッドアイデンティティ が AWS アカウント にアクセスすると、ロールが継承され、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、AWS IAM Identity Center を使用することをお勧めします。IAM アイデンティティセンター でユーザーとグループを作成するか、すべての AWS アカウント とアプリケーションで使用するために、独自の ID ソースで一連のユーザーとグループに接続して同期することもできます。IAM アイデンティティセンター の詳細については、「AWS IAM Identity Center ユーザーガイド」の「What is IAM アイデンティティセンター?」(IAM アイデンティティセンター とは) を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、1 人のユーザーまたは 1 つのアプリケーションに対して特定の権限を持つ AWS アカウント 内のアイデンティティです。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時的な認証情報を使用することをお勧めします。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、「IAM ユーザーガイド」の「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する権限を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時的な認証情報が提供されます。詳

細については、「IAM ユーザーガイド」の「IAM ユーザー (ロールではなく) の作成が適している場合」を参照してください。

IAM ロール

IAM ロールは、特定の権限を持つ、AWS アカウント 内のアイデンティティです。これは IAM ユーザーに似ていますが、特定のユーザーには関連付けられていません。ロールを切り替えることによって、AWS Management Console で IAM ロールを一時的に引き受けることができます。ロールを引き受けるには、AWS CLI または AWS API オペレーションを呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「IAM ロールの使用」を参照してください。

一時的な認証情報を持った IAM ロールは、以下の状況で役立ちます。

- フェデレーションユーザーユーザーアクセス – フェデレーションアイデンティティに権限を割り当てるには、ロールを作成してそのロールの権限を定義します。フェデレーティッドアイデンティティが認証されると、そのアイデンティティはロールに関連付けられ、ロールで定義されている権限が付与されます。フェデレーションの詳細については、「IAM ユーザーガイド」の「サードパーティー ID プロバイダー向けロールの作成」を参照してください。IAM アイデンティティセンターを使用する場合、権限セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。権限セットの詳細については、「AWS IAM Identity Center ユーザーガイド」の「権限セット」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。
- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の AWS のサービスでは、(ロールをプロキシとして使用する代わりに) リソースにポリシーを直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、「IAM ユーザーガイド」の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。
- クロスサービスアクセス - 一部の AWS のサービスでは、他の AWS のサービスの機能を使用します。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの権限、サービスロール、またはサービスリンクロールを使用してこれを行う場合があります。

- 転送アクセスセッション (FAS) – IAM ユーザーまたはロールを使用して AWS でアクションを実行するユーザーは、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行することで、別のサービスの別のアクションがトリガーされることがあります。FAS は、AWS のサービスを呼び出すプリンシパルの権限を、AWS のサービスのリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストは、サービスが、完了するために他の AWS のサービス またはリソースとのやりとりを必要とするリクエストを受け取ったときにのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。
- サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成」を参照してください。
- サービスリンクロール - サービスリンクロールは、AWS のサービス にリンクされたサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスリンクロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスリンクロールの権限を表示できますが、編集することはできません。

- Amazon EC2 で実行されているアプリケーション - EC2 インスタンスで実行され、AWS CLI または AWS API 要求を行っているアプリケーションの一時的な認証情報を管理するには、IAM ロールを使用できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。AWS ロールを EC2 インスタンスに割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスに添付されたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、「IAM ユーザーガイド」の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用してアクセス許可を付与する」を参照してください。

IAM ロールと IAM ユーザーのどちらを使用するかについては、「IAM ユーザーガイド」の「(IAM ユーザーではなく) IAM ロールをいつ作成したら良いのか?」を参照してください。

ポリシーを使用したアクセス権の管理

AWS でアクセス権を管理するには、ポリシーを作成して AWS アイデンティティまたはリソースにアタッチします。ポリシーは AWS のオブジェクトであり、アイデンティティやリソースに関連付けて、これらの権限を定義します。AWS は、プリンシパル (ユーザー、ルートユーザー、またはロールセッション) がリクエストを行うと、これらのポリシーを評価します。ポリシーでの権限により、

リクエストが許可されるか拒否されるかが決まります。大半のポリシーは JSON ドキュメントとして AWSに保存されます。JSON ポリシードキュメントの構造と内容の詳細については、「IAM ユーザーガイド」の「JSON ポリシー概要」を参照してください。

管理者は AWSJSON ポリシーを使用して、だれが何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

デフォルトでは、ユーザーやロールに権限はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者はロールに IAM ポリシーを追加し、ユーザーはロールを引き継ぐことができます。

IAM ポリシーは、オペレーションの実行方法を問わず、アクションの権限を定義します。例えば、`iam:GetRole` アクションを許可するポリシーがあるとします。このポリシーがあるユーザーは、AWS Management Console、AWS CLI、または AWS API からロール情報を取得できます。

アイデンティティベースポリシー

アイデンティティベースポリシーは、IAM ユーザー、ユーザーのグループ、ロールなど、アイデンティティにアタッチできる JSON 権限ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件を制御します。アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーの作成」を参照してください。

アイデンティティベースポリシーは、さらにインラインポリシーまたはマネージドポリシーに分類できます。インラインポリシーは、単一のユーザー、グループ、またはロールに直接埋め込まれます。管理ポリシーは、AWS アカウント 内の複数のユーザー、グループ、およびロールにアタッチできるスタンドアロンポリシーです。マネージドポリシーには、AWS マネージドポリシーとカスタマー管理ポリシーがあります。マネージドポリシーまたはインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「マネージドポリシーとインラインポリシーの比較」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プ

リンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービス を含めることができます。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは IAM の AWSマネージドポリシーは使用できません。

アクセスコントロールリスト (ACL)

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための権限を持つかをコントロールします。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Simple Storage Service (Amazon S3)、AWS WAF、および Amazon VPC は、ACL をサポートするサービスの例です。ACL の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「アクセスコントロールリスト (ACL) の概要」を参照してください。

その他のポリシータイプ

AWS では、他の一般的ではないポリシータイプをサポートしています。これらのポリシータイプでは、より一般的なポリシータイプで付与された最大の権限を設定できます。

- 権限の境界 - 権限の境界は、アイデンティティベースのポリシーによって IAM エンティティ (IAM ユーザーまたはロール) に付与できる許可の上限を設定する高度な機能です。エンティティに権限の境界を設定できます。結果として得られる権限は、エンティティのアイデンティティベースポリシーとその権限の境界の共通部分になります。`Principal` フィールドでユーザーまたはロールを指定するリソースベースのポリシーでは、権限の境界は制限されません。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。権限の境界の詳細については、「IAM ユーザーガイド」の「IAM エンティティの権限の境界」を参照してください。
- サービスコントロールポリシー (SCP) - SCP は、AWS Organizations で組織や組織単位 (OU) の最大権限を指定する JSON ポリシーです。AWS Organizations は、顧客のビジネスが所有する複数の AWS アカウント をグループ化し、一元的に管理するサービスです。組織内のすべての機能を有効にすると、サービスコントロールポリシー (SCP) を一部またはすべてのアカウントに適用できます。SCP はメンバーアカウントのエンティティに対する権限を制限します (各 AWS アカウントのルートユーザー など)。Organizations と SCP の詳細については、『AWS Organizations ユーザーガイド』の「SCP の仕組み」を参照してください。
- セッションポリシー - セッションポリシーは、ロールまたはフェデレーションユーザーの一時的なセッションをプログラムで作成する際にパラメータとして渡す高度なポリシーです。結果としてセッションの権限の範囲は、ユーザーまたはロールのアイデンティティベースポリシーとセッションポリシーの共通部分になります。また、リソースベースのポリシーから権限が派生する場合もあ

ります。これらのポリシーのいずれかを明示的に拒否した場合、権限は無効になります。詳細については、「IAM ユーザーガイド」の「セッションポリシー」をご参照ください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成される権限を理解するのがさらに難しくなります。複数のポリシータイプが関連するとき、リクエストを許可するかどうかを AWS が決定する方法の詳細については、「IAM ユーザーガイド」の「ポリシーの評価論理」を参照してください。

AWS Lambda と IAM の連携方法

IAM を使用して Lambda へのアクセスを管理する前に、Lambda で使用できる IAM の機能について学びます。

AWS Lambda で使用できる IAM の機能

IAM の機能	Lambda サポート
アイデンティティベースのポリシー	あり
リソースベースのポリシー	はい
ポリシーアクション	あり
ポリシーリソース	はい
ポリシー条件キー (サービス固有)	はい
ACL	なし
ABAC (ポリシー内のタグ)	部分的
一時的な認証情報	はい
転送アクセスセッション (FAS)	いいえ
サービスロール	あり
サービスリンクロール	部分的

Lambda およびその他の AWSのサービスがほとんどの IAM 機能と連携する方法の概要を把握するには、「IAM ユーザーガイド」のAWS「IAM と連携する のサービス」を参照してください。

Lambda のアイデンティティベースのポリシー

アイデンティティベースポリシーをサポートする	あり

アイデンティティベースポリシーは、IAM ユーザー、ユーザーグループ、ロールなど、アイデンティティにアタッチできる JSON 権限ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件を制御します。アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「IAM ポリシーの作成」を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。プリンシパルは、それがアタッチされているユーザーまたはロールに適用されるため、アイデンティティベースのポリシーでは指定できません。JSON ポリシーで使用できるすべての要素については、「IAM ユーザーガイド」の「IAM JSON ポリシーの要素のリファレンス」を参照してください。

Lambda のアイデンティティベースのポリシーの例

Lambda アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS Lambda のアイデンティティベースのポリシーの例。

Lambda 内のリソースベースのポリシー

リソースベースのポリシーのサポート	はい

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーでは、プリンシパルを指定する必要があります。プ

リンシパルには、アカウント、ユーザー、ロール、フェデレーションユーザー、または AWS のサービス を含めることができます。

クロスアカウントアクセスを有効にするには、全体のアカウント、または別のアカウントの IAM エンティティを、リソースベースのポリシーのプリンシパルとして指定します。リソースベースのポリシーにクロスアカウントのプリンシパルを追加しても、信頼関係は半分しか確立されない点に注意してください。プリンシパルとリソースが異なる AWS アカウント にある場合、信頼できるアカウントの IAM 管理者は、リソースにアクセスするための権限をプリンシパルエンティティ (ユーザーまたはロール) に付与する必要もあります。IAM 管理者は、アイデンティティベースのポリシーをエンティティにアタッチすることで権限を付与します。ただし、リソースベースのポリシーで、同じアカウントのプリンシパルへのアクセス権が付与されている場合は、アイデンティティベースのポリシーを追加する必要はありません。詳細については、『IAM ユーザーガイド』の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。

リソースベースのポリシーを Lambda 関数またはレイヤーにアタッチできます。このポリシーは、関数またはレイヤーに対してアクションを実行できるプリンシパルを定義します。

リソースベースのポリシーを関数またはレイヤーにアタッチする方法については、「」を参照してくださいLambda でのリソースベースのポリシーの使用。

Lambda のポリシーアクション

ポリシーアクションに対するサポート	あり

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

JSON ポリシーの`Action`要素には、ポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。ポリシーアクションの名前は通常、関連する AWS API オペレーションと同じです。一致する API オペレーションのない権限のみのアクションなど、いくつかの例外があります。また、ポリシーに複数アクションが必要なオペレーションもあります。これらの追加アクションは、依存アクションと呼ばれます。

このアクションは、関連付けられたオペレーションを実行するための権限を付与するポリシーで使用されます。

Lambda アクションのリストを確認するには、「サービス認証リファレンス」の「 で定義されるアクションAWS Lambda」を参照してください。

Lambda のポリシーアクションは、アクションの前に次のプレフィックスを使用します。

```
lambda
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

Lambda アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS Lambda のアイデンティティベースのポリシーの例。

Lambda のポリシーリソース

ポリシーリソースに対するサポート	あり

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

JSON ポリシーの Resource 要素は、アクションが適用される 1 つ以上のオブジェクトを指定します。ステートメントには、Resource または NotResource 要素を含める必要があります。ベストプラクティスとして、Amazon リソースネーム (ARN) を使用してリソースを指定します。これは、リソースレベルの権限と呼ばれる特定のリソースタイプをサポートするアクションに対して実行できます。

オペレーションのリスト化など、リソースレベルの権限をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

Lambda リソースのタイプとその ARNs」の「 で定義されるリソースタイプAWS Lambda」を参照してください。 どのアクションで各リソースの ARN を指定できるかについては、「AWS Lambda で定義されるアクション」を参照してください。

Lambda アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS Lambda のアイデンティティベースのポリシーの例。

Lambda のポリシー条件キー

サービス固有のポリシー条件キーのサポート	はい

管理者は AWS JSON ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどんなリソースにどんな条件でアクションを実行できるかということです。

Condition 要素 (または Condition ブロック) を使用すると、ステートメントが有効になる条件を指定できます。Condition 要素はオプションです。イコールや未満などの条件演算子を使用して条件式を作成することで、ポリシーの条件とリクエスト内の値を一致させることができます。

1 つのステートメントに複数の Condition 要素を指定するか、1 つの Condition 要素に複数のキーを指定すると、AWS は AND 論理演算子を使用してそれらを評価します。単一の条件キーに複数の値を指定すると、AWS は OR 論理演算子を使用して条件を評価します。ステートメントの権限が付与される前にすべての条件が満たされる必要があります。

条件を指定する際にプレースホルダー変数も使用できます。例えば IAM ユーザーに、IAM ユーザー名がタグ付けされている場合のみリソースにアクセスできる権限を付与することができます。詳細については、「IAM ユーザーガイド」の「IAM ポリシー要素: 変数およびタグ」を参照してください。

AWS はグローバル条件キーとサービス固有の条件キーをサポートしています。すべての AWS グローバル条件キーを確認するには、『IAM ユーザーガイド』の「AWS グローバル条件コンテキストキー」を参照してください。

Lambda 条件キーのリストを確認するには、「サービス認証リファレンス」の「 の条件キーAWS Lambda」を参照してください。どのアクションおよびリソースと条件キーを使用できるかについては、「AWS Lambdaで定義されるアクション」を参照してください。

Lambda アイデンティティベースのポリシーの例を表示するには、「」を参照してくださいAWS Lambda のアイデンティティベースのポリシーの例。

Lambda ACLs

ACL のサポート	なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするための権限を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

Lambda での ABAC

ABAC (ポリシー内のタグ) のサポート	部分的

属性ベースのアクセスコントロール (ABAC) は、属性に基づいて権限を定義する認可戦略です。AWS では、これらの属性はタグと呼ばれます。タグは、IAM エンティティ (ユーザーまたはロール)、および多数の AWS リソースにアタッチできます。エンティティとリソースのタグ付けは、ABAC の最初の手順です。その後、プリンシパルのタグがアクセスしようとしているリソースのタグと一致した場合に操作を許可するように ABAC ポリシーを設計します。

ABAC は、急成長する環境やポリシー管理が煩雑になる状況で役立ちます。

タグに基づいてアクセスを管理するには、`aws:ResourceTag/key-name`、`aws:RequestTag/key-name`、または `aws:TagKeys` の条件キーを使用して、ポリシーの Condition 要素でタグ情報を提供します。

サービスがすべてのリソースタイプに対して 3 つの条件キーのすべてをサポートする場合、そのサービスでのサポート状況の値は「はい」になります。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、『IAM ユーザーガイド』の「ABAC とは?」を参照してください。ABAC をセットアップするステップを説明するチュートリアルについては、「IAM ユーザーガイド」の「属性に基づくアクセスコントロール (ABAC) を使用する」を参照してください。

Lambda リソースのタグ付けの詳細については、「」を参照してくださいLambda での属性ベースのアクセスコントロールの使用。

Lambda での一時的な認証情報の使用

一時的な認証情報のサポート	あり

AWS のサービス には、一時的な認証情報を使用してサインインしても機能しないものがあります。一時的な認証情報で機能する AWS のサービスなどの詳細については、「IAM ユーザーガイド」の「IAM と連携する AWS のサービス」を参照してください。

ユーザー名とパスワード以外の方法で AWS Management Console にサインインする場合は、一時的な認証情報を使用していることになります。例えば、会社の Single Sign-On (SSO) リンクを使用して AWS にアクセスすると、そのプロセスは自動的に一時認証情報を作成します。また、ユーザーとしてコンソールにサインインしてからロールを切り替える場合も、一時的な認証情報が自動的に作成されます。ロールの切り替えに関する詳細については、「IAM ユーザーガイド」の「ロールへの切り替え (コンソール)」を参照してください。

一時認証情報は、AWS CLI または AWS API を使用して手動で作成できます。作成後、一時的な認証情報を使用して AWS にアクセスできるようになります。AWS は、長期的なアクセスキーを使用する代わりに、一時的な認証情報を動的に生成することをお勧めします。詳細については、「IAM の一時的なセキュリティ認証情報」を参照してください。

Lambda の転送アクセスセッション

転送アクセスセッション (FAS) をサポート	いいえ

IAM ユーザーまたはロールを使用して AWS でアクションを実行するユーザーは、プリンシパルと見なされます。一部のサービスを使用する際に、アクションを実行してから、別のサービスの別のアクションを開始することがあります。FAS は、AWS のサービスを呼び出すプリンシパルの権限を、AWS のサービスのリクエストと合わせて使用し、ダウンストリームのサービスに対してリクエストを行います。FAS リクエストは、サービスが、完了するために他の AWS のサービス またはリソースとのやりとりを必要とするリクエストを受け取ったときにのみ行われます。この場合、両方のアクションを実行するためのアクセス許可が必要です。FAS リクエストを行う際のポリシーの詳細については、「転送アクセスセッション」を参照してください。

Lambda のサービスロール

サービスロールに対するサポート	あり

サービスロールとは、サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成」を参照してください。

Lambda では、サービスロールは実行ロール と呼ばれます。

> **Warning**
> 実行ロールのアクセス許可を変更すると、Lambda の機能が破損する可能性があります。

Lambda のサービスにリンクされたロール

サービスリンクロールのサポート	部分的

サービスリンクロールは、AWS のサービスにリンクされているサービスロールの一種です。サービスがロールを引き受け、ユーザーに代わってアクションを実行できるようになります。サービスリンクロールは、AWS アカウント に表示され、サービスによって所有されます。IAM 管理者は、サービスリンクロールの権限を表示できますが、編集することはできません。

Lambda にはサービスにリンクされたロールはありませんが、Lambda@Edge にはあります。詳細については、「Amazon CloudFront デベロッパーガイド」の「Lambda@Edge のサービスリンクロール」を参照してください。

サービスにリンクされたロールの作成または管理の詳細については、「IAM と提携する AWS のサービス」を参照してください。表の中から、「サービスにリンクされたロール」列が「Yes」になっているサービスを見つけます。サービスリンクロールに関するドキュメントをサービスで表示するには、「はい」 リンクを選択します。

AWS Lambda のアイデンティティベースのポリシーの例

デフォルトで、ユーザーとロールには Lambda リソースを作成または変更する許可がありません。また、AWS Management Console、AWS Command Line InterfaceAWS CLI、または AWS API を使用してタスクを実行することもできません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。その後、管理者がロールに IAM ポリシーを追加すると、ユーザーはロールを引き受けることができます。

これらサンプルの JSON ポリシードキュメントを使用して、IAM アイデンティティベースのポリシーを作成する方法については、『IAM ユーザーガイド』の「IAM ポリシーの作成」を参照してください。

各リソースタイプの ARNs「 のアクション、リソース、および条件キーAWS Lambda」を参照してください。

トピック

- ポリシーのベストプラクティス
- Lambda コンソールを使用する
- 自分の許可の表示をユーザーに許可する

ポリシーのベストプラクティス

ID ベースのポリシーは、ユーザーのアカウントで誰かが Lambda リソースを作成、アクセス、または削除できるどうかを決定します。これらのアクションを実行すると、AWS アカウント に料金が発生する可能性があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS マネージドポリシーを使用して開始し、最小特権の権限に移行する – ユーザーとワークロードへの権限の付与を開始するには、多くの一般的なユースケースのために権限を付与する AWS マネージドポリシーを使用します。これらは AWS アカウントで使用できます。ユースケースに応じた AWS カスタマーマネージドポリシーを定義することで、権限をさらに減らすことをお勧めします。詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」または「AWS ジョブ機能のマネージドポリシー」を参照してください。
- 最小特権を適用する – IAM ポリシーで権限を設定するときは、タスクの実行に必要な権限のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権権限とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、「IAM ユーザーガイド」の「IAM でのポリシーと権限」を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する - ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。例えば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。また、AWS CloudFormation などの特定の AWS のサービス を介して使用する場合、条件を使用してサービスアクションへのアクセスを許可することもできます。詳細については、「IAM ユーザーガイド」の [IAM JSON policy elements: Condition] (IAM JSON ポリシー要素: 条件) を参照してください。
- IAM Access Analyzer を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する - IAM Access Analyzer は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM Access Analyzer は 100 を超えるポリシーチェックと実用的なリコメンデーションを提供し、安全で機能的なポリシーを作成できるようサポートします。詳細については、『IAM ユーザーガイド』の「IAM Access Analyzer ポリシーの検証」を参照してください。

- 多要素認証 (MFA) を要求する - AWS アカウント内の IAM ユーザーまたはルートユーザーを要求するシナリオがある場合は、セキュリティを強化するために MFA をオンにします。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、「IAM ユーザーガイド」の「MFA 保護 API アクセスの設定」を参照してください。

IAM でのベストプラクティスの詳細については、『IAM ユーザーガイド』の「IAM でのセキュリティのベストプラクティス」を参照してください。

Lambda コンソールを使用する

AWS Lambda コンソールにアクセスするには、最小限の許可セットが必要です。これらのアクセス許可により、 の Lambda リソースの詳細をリストおよび表示できますAWS アカウント。最小限必要なアクセス許可よりも制限が厳しいアイデンティティベースのポリシーを作成すると、そのポリシーを持つエンティティ (ユーザーまたはロール) ではコンソールが意図したとおりに機能しません。

AWS CLI または AWS API のみを呼び出すユーザーには、最小限のコンソール権限を付与する必要はありません。代わりに、実行しようとしている API オペレーションに一致するアクションのみへのアクセスを許可します。

関数開発に最小限のアクセスを許可するポリシーの例については、「」を参照してください関数にユーザーアクセス許可を付与するサンプルポリシーの作成 Lambda API に加えて、Lambda コンソールでは他のサービスを使用して、トリガーの設定を表示し、新しいトリガーを追加できるようにします。ユーザーが他のサービスで Lambda を使用する場合は、それらのサービスにもアクセスする必要があります。Lambda を使用して他のサービスを設定する方法の詳細については、他の AWS サービスからのイベントを使用した Lambda の呼び出し を参照してください。

自分の許可の表示をユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI か AWS API を使用してプログラム的に、このアクションを完了するアクセス許可が含まれています。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
```

```
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS の AWS Lambda マネージドポリシー

AWS マネージドポリシーは、AWS が作成および管理するスタンドアロンポリシーです。AWS マネージドポリシーは、多くの一般的なユースケースで権限を提供できるように設計されているため、ユーザー、グループ、ロールへの権限の割り当てを開始できます。

AWS マネージドポリシーは、ご利用の特定のユースケースに対して最小特権の権限を付与しない場合があることにご注意ください。AWS のすべてのお客様が使用できるようになるのを避けるためです。ユースケース別にカスタマー管理ポリシーを定義することで、権限を絞り込むことをお勧めします。

AWS マネージドポリシーで定義したアクセス権限は変更できません。AWS が AWS マネージドポリシーに定義されている権限を更新すると、更新はポリシーがアタッチされているすべてのプリンシパルアイデンティティ (ユーザー、グループ、ロール) に影響します。新しい AWS のサービスを起動

するか、既存のサービスで新しい API オペレーションが使用可能になると、AWS が AWS マネージドポリシーを更新する可能性が最も高くなります。

詳細については、「IAM ユーザーガイド」の「AWS マネージドポリシー」を参照してください。

トピック

- AWS マネージドポリシー： AWSLambda_FullAccess
- AWS マネージドポリシー： AWSLambda_ReadOnlyAccess
- AWS マネージドポリシー： AWSLambdaBasicExecutionRole
- AWS マネージドポリシー： AWSLambdaDynamoDBExecutionRole
- AWS マネージドポリシー： AWSLambdaENIManagementAccess
- AWS マネージドポリシー： AWSLambdaExecute
- AWS マネージドポリシー： AWSLambdaInvocation-DynamoDB
- AWS マネージドポリシー： AWSLambdaKinesisExecutionRole
- AWS マネージドポリシー： AWSLambdaMSKExecutionRole
- AWS マネージドポリシー： AWSLambdaRole
- AWS マネージドポリシー： AWSLambdaSQSQueueExecutionRole
- AWS マネージドポリシー： AWSLambdaVPCAccessExecutionRole
- Lambda での AWS マネージドポリシーの更新

AWS マネージドポリシー： AWSLambda_FullAccess

このポリシーは Lambda アクションに対するフルアクセスを付与します。また、Lambda リソースの開発と保守に使用される他の AWS サービスへのアクセス許可も付与します。

ユーザー、グループおよびロールに `AWSLambda_FullAccess` ポリシーをアタッチできます。

許可の詳細

このポリシーには、以下の許可が含まれています。

- `lambda` – プリンシパルに Lambda への完全なアクセスを許可します。
- `cloudformation` — プリンシパルが AWS CloudFormation スタックを記述し、それらのスタック内のリソースを一覧表示できるようにします。
- `cloudwatch` — プリンシパルが Amazon CloudWatch メトリクスを一覧表示し、メトリクスデータを取得できるようにします。

- ec2 — プリンシパルがセキュリティグループ、サブネット、および VPC を記述できるようにします。
- iam — プリンシパルがポリシー、ポリシーバージョン、ロール、ロールポリシー、アタッチされたロールポリシー、およびロールのリストを取得できるようにします。また、このポリシーでは、プリンシパルが Lambda にロールを渡すことも許可されます。PassRole 許可は、関数に実行ロールを割り当てる際に使用します。
- kms - プリンシパルがエイリアスを一覧表示できるようにします。
- logs — プリンシパルが Amazon CloudWatch ロググループを記述できるようにします。Lambda 関数に関連付けられているロググループに対して、このポリシーにより、プリンシパルはログストリームの記述、ログイベントの取得、およびログイベントのフィルタリングを行うことができるようになります。
- states — プリンシパルが AWS Step Functions 状態のマシンを記述および一覧表示できるようにします。
- tag — プリンシパルがタグに基づいてリソースを取得できるようにします。
- xray — プリンシパルが AWS X-Ray トレースサマリーを取得し、ID で指定されたトレースのリストを取得できるようにします。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambda_FullAccess」の「」を参照してください。

AWS マネージドポリシー： AWSLambda_ReadOnlyAccess

このポリシーは、Lambda リソースと、Lambda リソースの開発と保守に使用される他の AWS サービスへの読み取り専用アクセスを許可します。

ユーザー、グループおよびロールに AWSLambda_ReadOnlyAccess ポリシーをアタッチできます。

許可の詳細

このポリシーには、以下の許可が含まれています。

- lambda - プリンシパルがすべてのリソースを取得して一覧表示できるようにします。
- cloudformation — プリンシパルが AWS CloudFormation スタックを記述および一覧表示して、それらのスタック内のリソースを一覧表示できるようにします。
- cloudwatch — プリンシパルが Amazon CloudWatch メトリクスを一覧表示し、メトリクスデータを取得できるようにします。

- ec2 — プリンシパルがセキュリティグループ、サブネット、および VPC を記述できるようにします。
- iam — プリンシパルがポリシー、ポリシーバージョン、ロール、ロールポリシー、アタッチされたロールポリシー、およびロールのリストを取得できるようにします。
- kms - プリンシパルがエイリアスを一覧表示できるようにします。
- logs — プリンシパルが Amazon CloudWatch ロググループを記述できるようにします。Lambda 関数に関連付けられているロググループに対して、このポリシーにより、プリンシパルはログストリームの記述、ログイベントの取得、およびログイベントのフィルタリングを行うことができるようになります。
- states — プリンシパルが AWS Step Functions 状態のマシンを記述および一覧表示できるようにします。
- tag — プリンシパルがタグに基づいてリソースを取得できるようにします。
- xray — プリンシパルが AWS X-Ray トレースサマリーを取得し、ID で指定されたトレースのリストを取得できるようにします。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambda_ReadOnlyAccess」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaBasicExecutionRole

このポリシーは、ログを CloudWatch Logs にアップロードするアクセス許可を付与します。

ユーザー、グループおよびロールに AWSLambdaBasicExecutionRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaBasicExecutionRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaDynamoDBExecutionRole

このポリシーは、Amazon DynamoDB ストリームからレコードを読み取り、 CloudWatch Logs に書き込むためのアクセス許可を付与します。

ユーザー、グループおよびロールに AWSLambdaDynamoDBExecutionRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaDynamoDBExecutionRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaENIManagementAccess

このポリシーは、VPC 対応の Lambda 関数で使用される Elastic Network Interface を作成、記述、削除する許可を付与します。

ユーザー、グループおよびロールに `AWSLambdaENIManagementAccess` ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaENIManagementAccess」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaExecute

このポリシーはPUT、Amazon Simple Storage Service への および `GET` アクセスと、 CloudWatch ログへのフルアクセスを付与します。

ユーザー、グループおよびロールに `AWSLambdaExecute` ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaExecute」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaInvocation-DynamoDB

このポリシーは Amazon DynamoDB Streams への読み取りアクセスを許可します。

ユーザー、グループおよびロールに `AWSLambdaInvocation-DynamoDB` ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイド」のAWSLambdaInvocation「-DynamoDB」を参照してください。

AWS マネージドポリシー： AWSLambdaKinesisExecutionRole

このポリシーは、Amazon Kinesis データストリームからイベントを読み取り、 CloudWatch ログに書き込むアクセス許可を付与します。

ユーザー、グループおよびロールに AWSLambdaKinesisExecutionRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaKinesisExecutionRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaMSKExecutionRole

このポリシーは、Amazon Managed Streaming for Apache Kafka クラスターからのレコードの読み取りとアクセス、Elastic Network Interface の管理、および CloudWatch Logs への書き込みを行うためのアクセス許可を付与します。

ユーザー、グループおよびロールに AWSLambdaMSKExecutionRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaMSKExecutionRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaRole

このポリシーは、Lambda 関数を呼び出す許可を付与します。

ユーザー、グループおよびロールに AWSLambdaRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaSQSQueueExecutionRole

このポリシーは、Amazon Simple Queue Service キューからのメッセージの読み取りと削除のアクセス許可を付与し、 CloudWatch ログへの書き込みアクセス許可を付与します。

ユーザー、グループおよびロールに AWSLambdaSQSQueueExecutionRole ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaSQSQueueExecutionRole」の「」を参照してください。

AWS マネージドポリシー： AWSLambdaVPCAccessExecutionRole

このポリシーは、Amazon Virtual Private Cloud 内の Elastic Network Interface を管理し、CloudWatch ログに書き込むためのアクセス許可を付与します。

ユーザー、グループおよびロールに `AWSLambdaVPCAccessExecutionRole` ポリシーをアタッチできます。

JSON ポリシードキュメントやポリシーバージョンなど、このポリシーの詳細については、「 AWS マネージドポリシーリファレンスガイドAWSLambdaVPCAccessExecutionRole」の「」を参照してください。

Lambda での AWS マネージドポリシーの更新

変更	説明	日付
AWSLambdaVPCAccessExecutionRole – 変更	Lambda は、アクション を許可するようにAWSLambdaVPCAccessExecutionRole ポリシーを更新しましたec2:DescribeSubnets 。	2024 年 1 月 5 日
AWSLambda_ReadOnlyAccess – 変更	Lambda は、プリンシパルが AWS CloudFormation スタックを一覧表示できるように AWSLambda_ReadOnlyAccess ポリシーを更新しました。	2023 年 7 月 27 日
AWS Lambda は変更の追跡を開始しました	AWS Lambda が AWS マネージドポリシーの変更の追跡を開始しました。	2023 年 7 月 27 日

AWS Lambda ID とアクセスのトラブルシューティング

次の情報は、Lambda と IAM の使用に伴い発生する可能性のある、一般的な問題の診断や修復に役立ちます。

トピック

Lambda でアクションを実行する権限がない

「I am not authorized to perform an action in Amazon Bedrock」というエラーが表示された場合、そのアクションを実行できるようにポリシーを更新する必要があります。

次の例は、mateojackson という IAM ユーザーがコンソールを使用して架空の *my-example-widget* リソースに関する詳細を表示しようとしたとき、架空の lambda:*GetWidget* アクセス許可がない場合に発生するエラーを示しています。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

この場合、lambda:*GetWidget* アクションを使用して *my-example-widget*リソースへのアクセスを許可するように、mateojackson ユーザーのポリシーを更新する必要があります。

サポートが必要な場合は、AWS 管理者に問い合わせてください。管理者とは、サインイン認証情報を提供した担当者です。

iam を実行する権限がありません。PassRole

iam:PassRole アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して Lambda にロールを渡せるようにする必要があります。

一部の AWS のサービスでは、新しいサービスロールやサービスリンクロールを作成せずに、既存のロールをサービスに渡すことができます。そのためには、サービスにロールを渡す権限が必要です。

以下の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して Lambda でアクションを実行しようすると、発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与された権限が必要です。Mary には、ロールをサービスに渡す権限がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

この場合、Mary のポリシーを更新して、Mary に `iam:PassRole` アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者に問い合わせてください。管理者とは、サインイン認証情報を提供した担当者です。

自分の 以外のユーザーに Lambda リソースAWS アカウントへのアクセスを許可したい

他のアカウントのユーザーや組織外の人が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセス制御リスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください。

- Lambda でこれらの機能がサポートされるかどうかを確認するには、AWS Lambda と IAM の連携方法 を参照してください。
- 所有している AWS アカウント全体のリソースへのアクセス権を提供する方法については、IAM ユーザーガイドの「所有している別の AWS アカウント アカウントへのアクセス権を IAM ユーザーに提供」を参照してください。
- サードパーティーの AWS アカウント にリソースへのアクセス権を提供する方法については、『IAM ユーザーガイド』の「第三者が所有する AWS アカウント へのアクセス権を付与する」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、『IAM ユーザーガイド』の「外部で認証されたユーザー (ID フェデレーション) へのアクセスの許可」を参照してください。
- クロスアカウントアクセスでのロールとリソースベースのポリシーの使用の違いの詳細については、『IAM ユーザーガイド』の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。

Lambda 関数とレイヤーのガバナンス戦略を作成する

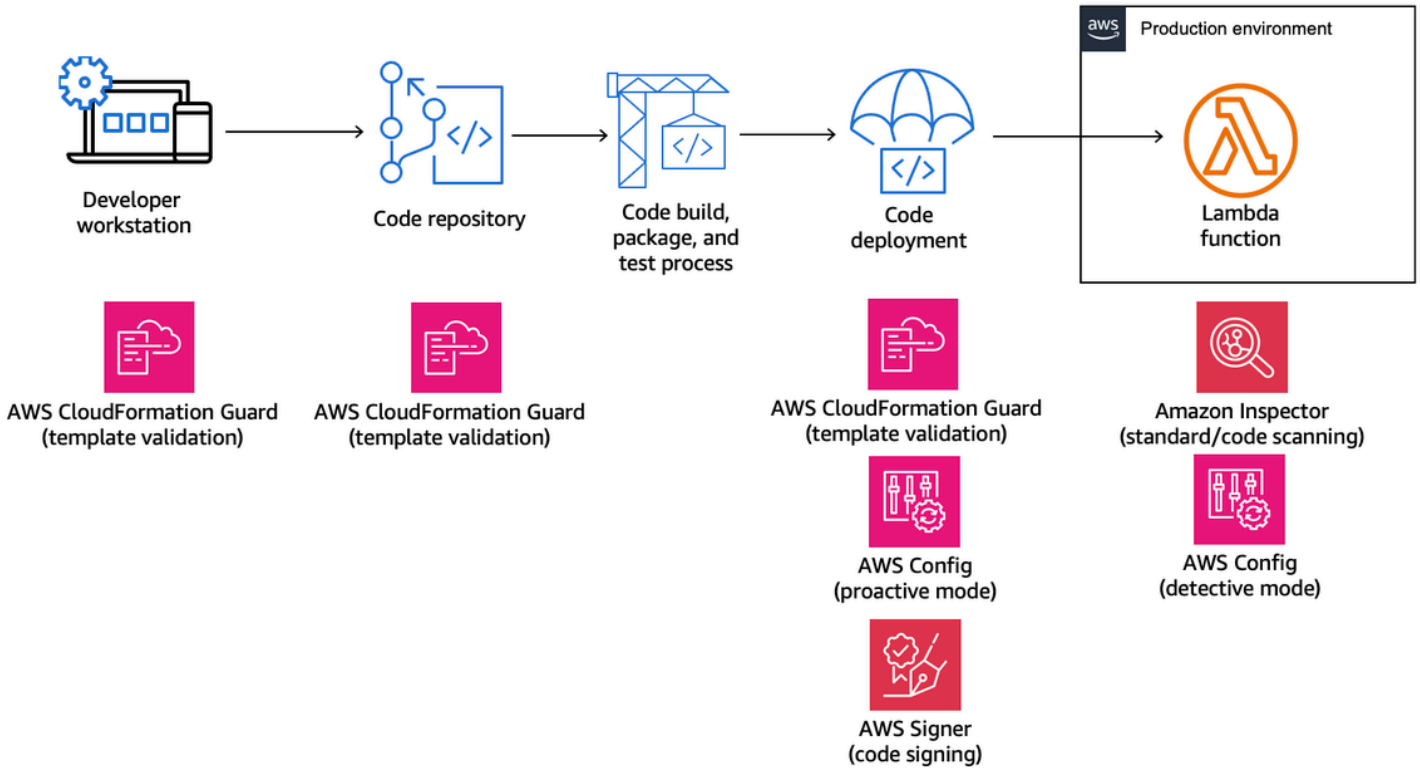
サーバーレスのクラウドネイティブアプリケーションを構築してデプロイするには、適切なガバナンスとガードレールを備えた俊敏性と市場投入までのスピードを考慮する必要があります。ビジネスレベルの優先事項を設定し、最優先事項として俊敏性を重視したり、ガバナンス、ガードレール、統制によるリスク回避を重視したりします。現実的には、「二者択一」戦略ではなく、ソフトウェア開発ライフサイクルにおけるアジリティとガードレールの両方のバランスを取る「両立」戦略を採用する

ことになります。これらの要件が会社のライフサイクルのどの段階にあっても、ガバナンス機能はプロセスやツールチェーンの実装要件になる可能性が高くなります。

組織が Lambda に実装する可能性のあるガバナンスコントロールの例をいくつか紹介します。

- Lambda 関数へのパブリックアクセスを許可してはなりません。
- Lambda 関数は VPC にアタッチされている必要があります。
- Lambda 関数では非推奨のランタイムを使用してはなりません。
- Lambda 関数は、必要なタグのセットでタグ付けする必要があります。
- Lambda レイヤーは組織の外部からアクセスできないようにする必要があります。
- セキュリティグループがアタッチされている Lambda 関数では、関数とセキュリティグループのタグが一致している必要があります。
- レイヤーがアタッチされた Lambda 関数は承認されたバージョンを使用する必要があります
- Lambda 環境変数は、カスタマーマネージド型キーを使用して保存時に暗号化する必要があります。

次の図は、ソフトウェアの開発とデプロイのプロセス全体にわたってコントロールとポリシーを実装する詳細なガバナンス戦略の例です。



以下のトピックでは、スタートアップとエンタープライズの両方を対象に、組織で Lambda 関数を開発およびデプロイするためのコントロールを実装する方法について説明します。組織でツールが設定済みである場合があります。以下のトピックでは、これらのコントロールをモジュール方式で行うため、実際に必要なコンポーネントを自由に選択できます。

トピック

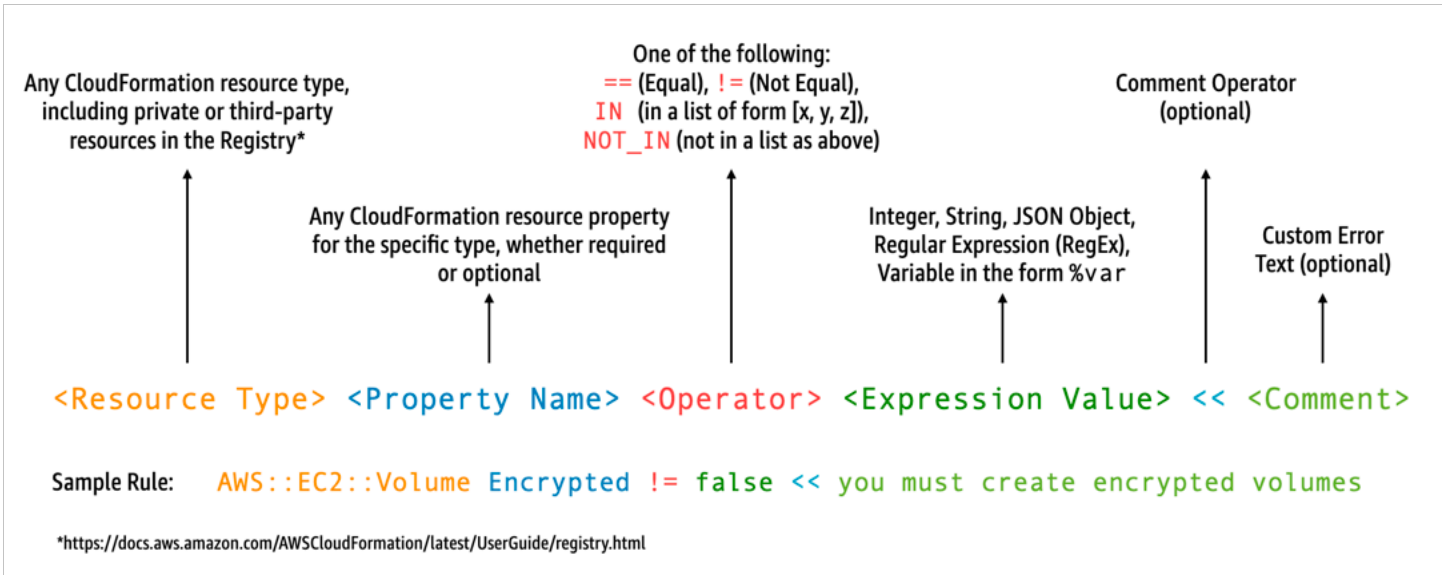
- AWS CloudFormation Guard による Lambda のプロアクティブコントロール
- AWS Config を使用して Lambda の予防的コントロールを実装する
- AWS Config を使用して非準拠の Lambda デプロイと設定を検出する
- AWS Signer で Lambda コード署名を行う
- Amazon Inspector を使用して Lambda のセキュリティ評価を自動化する
- Lambda のセキュリティとコンプライアンスのためのオブザーバビリティの実装

AWS CloudFormation Guard による Lambda のプロアクティブコントロール

AWS CloudFormation Guard は、オープンソースの汎用 policy-as-code 評価ツールです。Infrastructure as Code (IaC) テンプレートとサービス構成をポリシールールに照らして検証することで、ガバナンスとコンプライアンスの予防を行えます。これらのルールは、チームや組織の要件に基づいてカスタマイズできます。Lambda 関数の場合、Lambda 関数の作成または更新時に必要なプロパティ設定を定義することで、Guard ルールを使用してリソースの作成と設定の更新を制御できます。

コンプライアンス管理者は、Lambda 関数のデプロイと更新に必要なコントロールとガバナンスポリシーのリストを定義します。プラットフォーム管理者は、コードリポジトリを備えたコミット前の検証 Webhook として CI/CD パイプラインにコントロールを実装し、ローカルワークステーションでテンプレートとコードを検証するためのコマンドラインツールを開発者に提供します。開発者はコードを作成し、コマンドラインツールでテンプレートを検証し、コードをリポジトリにコミットします。リポジトリは、AWS 環境へのデプロイ前に CI/CD パイプラインを介して自動的に検証されます。

Guard では、以下のようにドメイン固有の言語を使用してルールを記述し、コントロールを実装できます。



例えば、開発者が必ず最新のランタイムのみを選択するように指定するとします。2 つの異なるポリシーを指定できます。1 つは廃止済みのランタイムを識別するためのもので、もう 1 つは間もなく廃止されるランタイムを識別するためのものです。これを行うには、以下の `etc/rules.guard` ファイルを記述します。

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

ここで、Lambda 関数を定義する次のiac/lambda.yaml CloudFormation テンプレートを記述するとします。

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

Guard ユーティリティをインストールしたら、テンプレートを検証します。

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

出力は次のようになります。

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
        ComparisonError {
          Message          = Lambda function is using a runtime that is targeted for
 deprecation.
          Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
        }
          PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
          Operator        = NOT IN
          Value           = "python3.7"
          ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
          Code:
                86.  Fn:
                87.    Type: AWS::Lambda::Function
                88.    Properties:
                89.      Runtime: python3.7
                90.      CodeUri: src
                91.      Handler: fn.handler

      }
    }
  }
}
```

Guard を使用すると、開発者はローカルの開発者ワークステーションを参照し、組織で許可されているランタイムを使用するにはテンプレートを更新する必要があることを確認できます。これは、コードリポジトリにコミットし、その後 CI/CD パイプライン内のチェックに失敗する前に発生します。これにより開発者がコンプライアンスに準拠したテンプレートを開発する方法に関するフィードバックを得ることができ、ビジネス価値をもたらすコードの作成に時間を割くことができます。この制御は、ローカルの開発者ワークステーション、コミット前の検証 Webhook、デプロイ前の CI/CD パイプラインに適用できます。

注意

AWS Serverless Application Model (AWS SAM) テンプレートを使用して Lambda 関数を定義する場合、以下のように `AWS::Serverless::Function` リソースタイプを検索するように Guard ルールを更新する必要があることに注意してください。

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

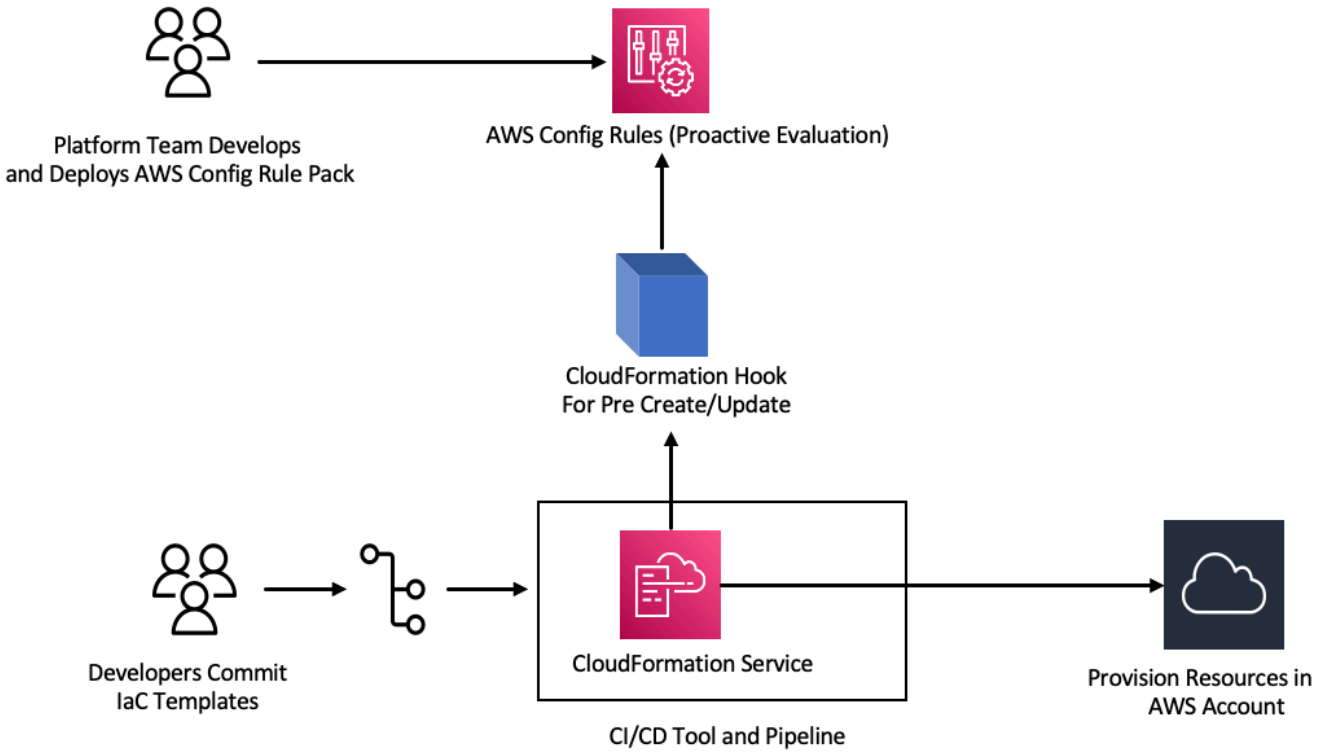
Guard では、プロパティがリソース定義に含まれていることも想定しています。一方、AWS SAM テンプレートではプロパティを別の [Globals] セクションで指定できます。Globals セクションで定義されているプロパティは、Guard ルールでは検証されません。

Guard のトラブルシューティングドキュメントで説明されているように、Guard は `!GetAtt`、`!Sub` のような短縮形式の組み込み関数をサポートしておらず、`Fn::GetAtt`、`Fn::Sub` の拡張形式を使用する必要があることに注意してください。(前の例では Role プロパティを評価していないため、簡潔にするために短縮形式の組み込み関数を使用しました)。

AWS Config を使用して Lambda の予防的コントロールを実装する

開発プロセスのできるだけ早い段階で、サーバーレスアプリケーションのコンプライアンスを確保することが不可欠です。このトピックでは、AWS Config を使用して予防的制御を実装する方法について説明します。これにより、開発プロセスの早い段階でコンプライアンスチェックを実装でき、CI/CD パイプラインにも同じコントロールを実装できます。また、一元管理されるルールのリポジトリでコントロールが標準化され、AWS アカウント全体にコントロールを一貫して適用できるようになります。

例えば、コンプライアンス管理者が、すべての Lambda 関数に AWS X-Ray トレースが含まれるようにする要件を定義したとします。AWS Config のプロアクティブモードを使用すると、デプロイ前に Lambda 関数リソースのコンプライアンスチェックを実行できます。不適切に設定された Lambda 関数をデプロイするリスクを軽減し、インフラストラクチャに関するフィードバックをコードテンプレートとして迅速に提供することで、開発者の時間を節約できます。以下は AWS Config による予防コントロールのフローを視覚化したものです。



すべての Lambda 関数でトレースを有効にする必要があるという要件を考えてみましょう。これを受けて、プラットフォームチームは、特定の AWS Config ルールをすべてのアカウントでプロアク

ティブに実行する必要があると判断しました。このルールは、X-Ray トレーシング設定が設定されていない Lambda 関数を非準拠リソースとしてフラグします。チームはルールを作成し、それをコンフォーマンスパックにパッケージ化し、そのコンフォーマンスパックをすべての AWS アカウントにデプロイして、組織内のすべてのアカウントがこれらのコントロールを統一的に適用できるようにします。ルールは AWS CloudFormation Guard 2.x.x 構文で記述し、次のような形式を取ります。

```
rule name when condition { assertion }
```

以下は、Lambda 関数でトレースが有効になっていることを確認する Guard ルールのサンプルです。

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

プラットフォームチームは、すべての AWS CloudFormation デプロイで事前作成/更新フックを呼び出すことを義務付けることで、さらなる措置を講じます。また、このフックを開発してパイプラインを構成し、コンプライアンスルールの一元管理を強化し、すべてのデプロイメントで一貫した適用を維持する全責任を負います。フックを開発、パッケージ化、登録するには、CloudFormation コマンドラインインターフェイス (CFN-CLI) ドキュメントの「Developing AWS CloudFormation Hooks」を参照してください。CloudFormation CLI を使用してフックプロジェクトを作成できます。

```
cfn init
```

このコマンドは、フックプロジェクトに関する基本情報の入力を求め、以下のファイルを含むプロジェクトを作成します。

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

フック開発者は、必要なターゲットリソースタイプを <hook-name>.json 設定ファイルに追加する必要があります。以下の設定では、CloudFormation を使用して Lambda 関数が作成される前に

フックが実行されるように設定されています。preUpdate および preDelete アクションにも同様のハンドラーを追加できます。

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

また、CloudFormation フックに AWS Config API を呼び出すための適切な権限があることを確認する必要があります。そのためには、hook-role.yaml という名前のロール定義ファイルを更新します。ロール定義ファイルには、デフォルトで以下の信頼ポリシーがあります。これにより、CloudFormation がロールを引き受けることができます。

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

このフックで config API を呼び出せるようにするには、ポリシーステートメントに以下の権限を追加する必要があります。次に、cfn submit コマンドを使用してフックプロジェクトを送信します。ここで、CloudFormation は必要な権限を持つロールを作成します。

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
```

```
                Resource: "*
```

次に、Lambda 関数を src/handler.py ファイルに記述する必要があります。このファイルには preCreate、preUpdate、preDelete という名前のメソッドがあり、プロジェクトを開始したときにすでに作成されています。目的は、AWS SDK for Python (Boto3) を使用してプロアクティブモードで AWS Config StartResourceEvaluation API を呼び出す、再利用可能な共通関数を作成することです。この API コールは、リソースプロパティを入力として受け取り、ルール定義と照らし合わせてリソースを評価します。

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

これで、作成前フックのハンドラーから共通関数を呼び出すことができます。ハンドラーの例を示します。

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
```

```
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

このステップの後、フックを登録して、すべての AWS Lambda 関数作成イベントをリッスンするように設定できます。

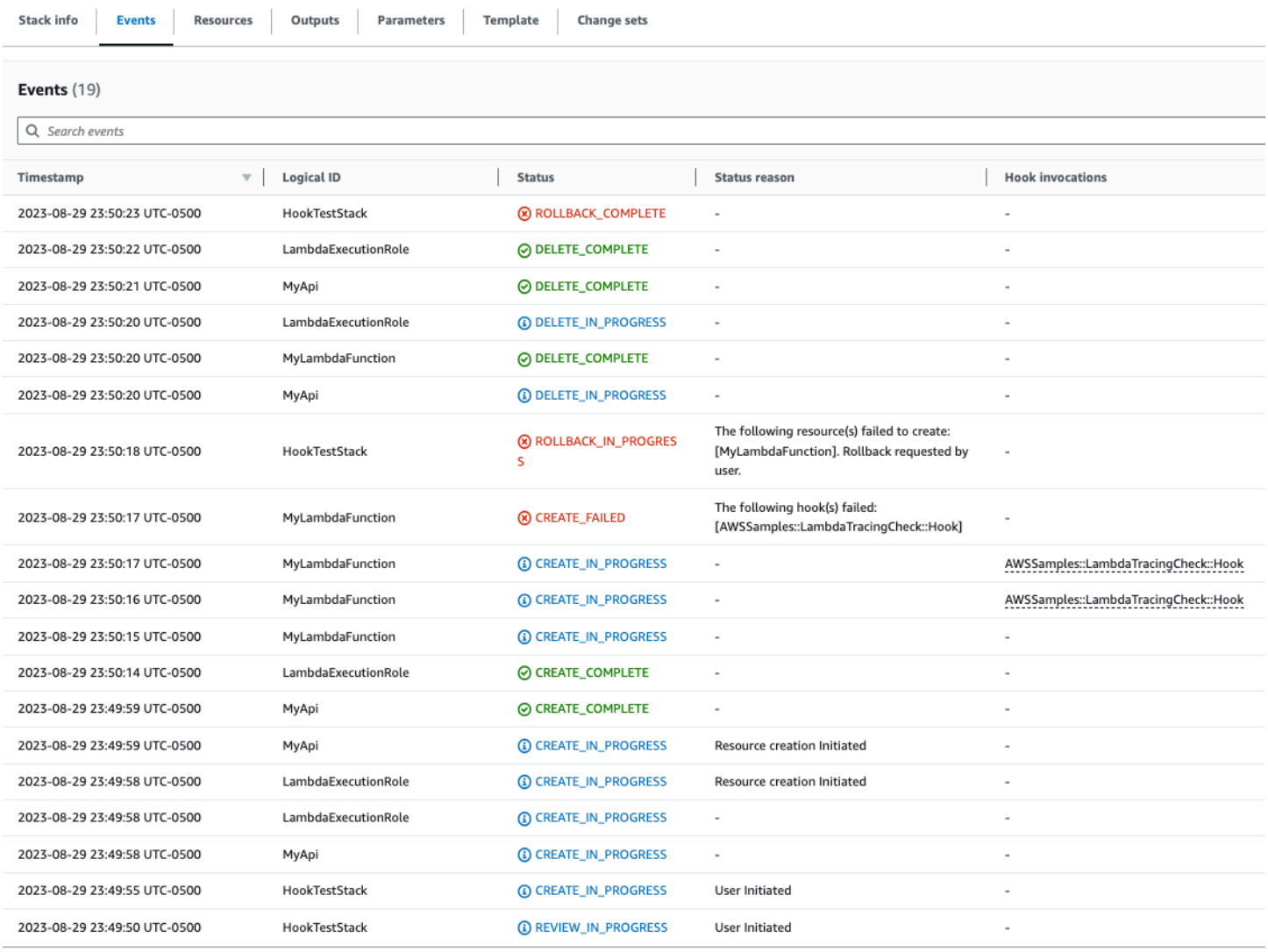
開発者は、Lambda を使用してサーバーレスマイクロサービス用の Infrastructure as Code (IaC) テンプレートを準備します。この準備には、内部標準を順守した後に、テンプレートをローカルでテストしてリポジトリにコミットすることが含まれます。IaC テンプレート例を次に示します。

```
  MyLambdaFunction:
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.8
    TracingConfig:
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

CI/CD プロセスの一環として、CloudFormation テンプレートがデプロイされると、CloudFormation サービスは `AWS::Lambda::Function` リソースタイプをプロビジョニングする直前に事前

作成/更新フックを呼び出します。フックは、プロアクティブモードで実行されている AWS Config ルールを利用して、Lambda 関数設定に必須のトレース設定が含まれていることを確認します。フックからの応答によって次のステップが決まります。準拠していれば、フックは成功を通知し、CloudFormation はリソースのプロビジョニングを続行します。そうでない場合、CloudFormation スタックのデプロイは失敗し、パイプラインはただちに停止し、システムはその詳細を記録して後で確認できるようにします。コンプライアンス通知は、関連する利害関係者に送信されます。

フックの成功/失敗に関する情報は、CloudFormation コンソールで確認できます。



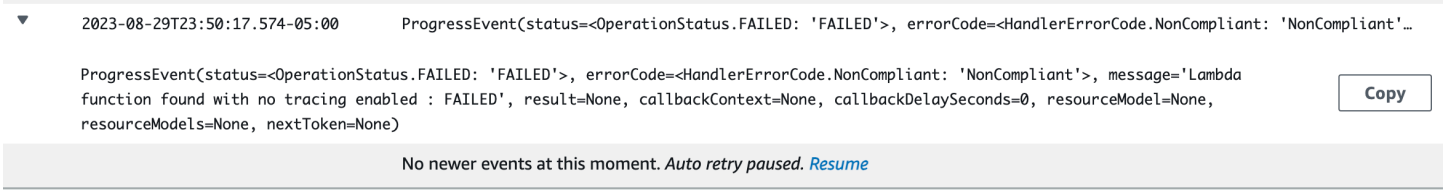
Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

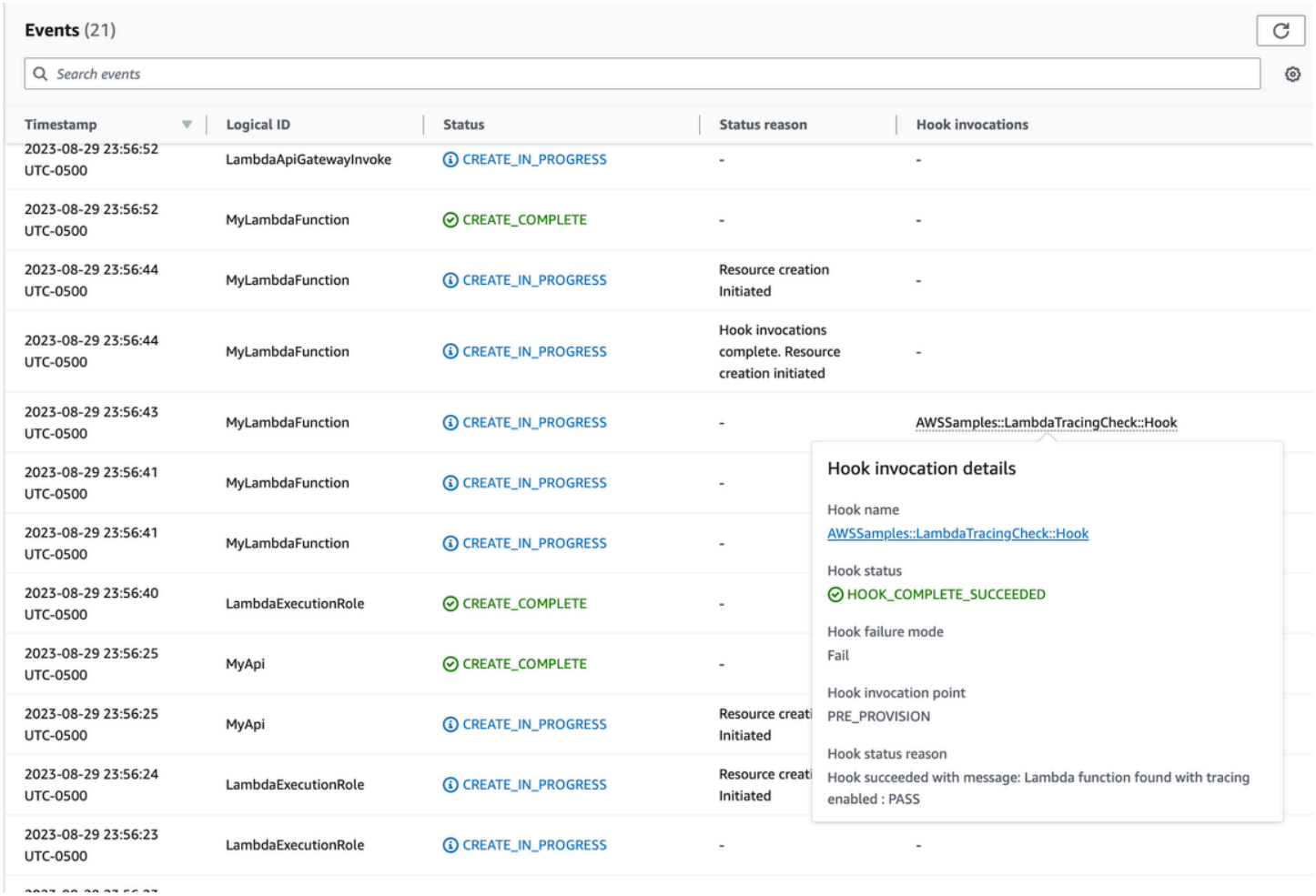
Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

CloudFormation フックのログが有効になっている場合は、フックの評価結果をキャプチャできます。以下は、失敗ステータスのフックのサンプルログです。これは、Lambda 関数で X-Ray が有効になっていないことを示しています。



開発者が IaC を変更して `TracingConfig` `Mode` 値を `Active` に更新して再デプロイすることを選択した場合、フックは正常に実行され、スタックは Lambda リソースの作成に進みます。



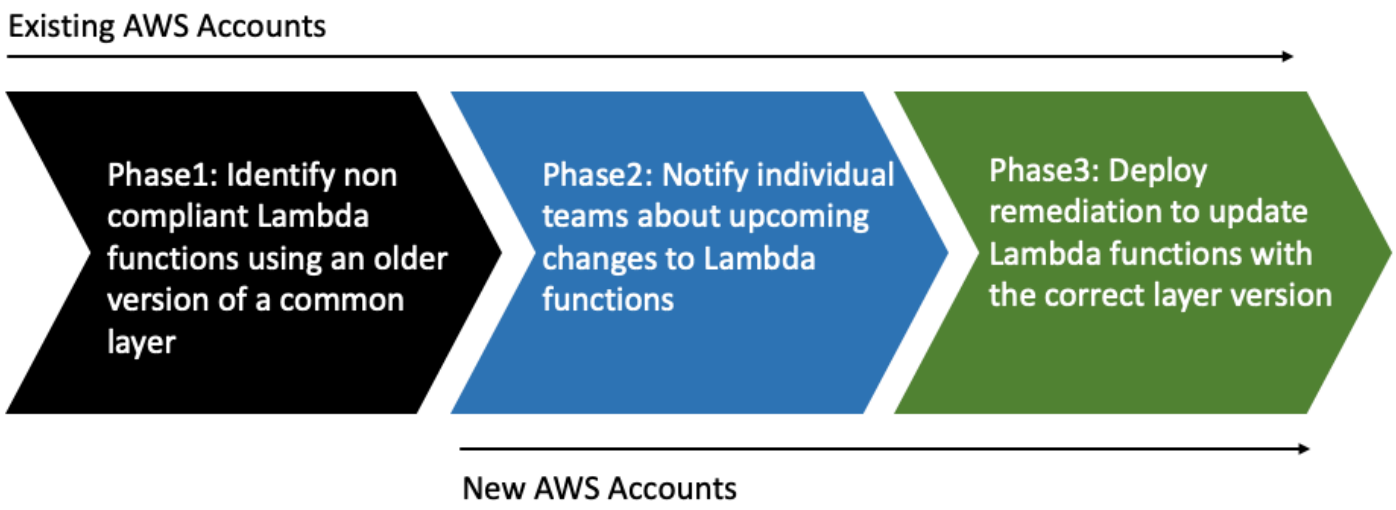
これにより、サーバーレスリソースを開発して AWS アカウントにデプロイするときに、AWS Config による予防的コントロールをプロアクティブモードで実装できます。AWS Config ルールを CI/CD パイプラインに統合することで、アクティブなトレーシング設定がない Lambda 関数など、非準拠のリソースのデプロイを特定し、オプションでブロックできます。これにより、最新のガバナンスポリシーに準拠するリソースのみが AWS 環境にデプロイされます。

AWS Config を使用して非準拠の Lambda デプロイと設定を検出する

事前評価に加えて、AWS Config は、ガバナンスポリシーに準拠していないリソースのデプロイや設定を事後的に検出することもできます。ガバナンスポリシーは、組織が新しいベストプラクティスを学び、実装するにつれて変化するため、これは重要です。

Lambda 関数をデプロイまたは更新するときに、まったく新しいポリシーを設定するシナリオを考えてみましょう。すべての Lambda 関数には、常に特定の承認された Lambda レイヤーバージョンを使用する必要があります。AWS Config を設定し、レイヤー設定の新規または更新された関数をモニタリングするようにできます。AWS Config が承認されたレイヤーバージョンを使用していない機能を検出すると、その関数は非準拠リソースとしてフラグ付けされます。オプションで、AWS Systems Manager 自動化ドキュメントを使用した修復アクションを指定することで、AWS Config がリソースを自動的に修正するように設定できます。例えば、AWS SDK for Python (Boto3) を使用して Python でオートメーションドキュメントを作成できます。これにより、非準拠関数が承認されたレイヤーバージョンを指すように更新されます。したがって、AWS Config は検出と是正の両方の制御を行い、コンプライアンス管理を自動化します。

このプロセスを次の 3 つの重要な実装フェーズに分けてみましょう。



フェーズ 1: アクセスリソースの特定

まず、アカウント全体で AWS Config を有効にし、AWS Lambda 関数を記録するように設定します。これにより、AWS Config は Lambda 関数がいつ作成または更新されたかを確認できます。その後、AWS CloudFormation Guard 構文を使用して特定のポリシー違反をチェックするカスタムポリシールールを設定できます。Guard ルールには次のような一般的な形式があります。

```
rule name when condition { assertion }
```

以下は、古いレイヤーバージョンにレイヤーが設定されていないことを確認するルールのサンプルです。

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

ルールの構文と構造を理解しましょう。

- ルール名: 例では、ルール名は `desiredlayer` です。
- 条件: この句は、ルールを確認する条件を指定します。示されている例では、条件は `configuration.layers !empty` です。つまり、設定内の `layers` プロパティが空でない場合にのみ、リソースが評価されます。
- アサーション: when 句の後に、アサーションによってルールがチェックする内容が決まります。アサーション `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` は、Lambda レイヤの ARN のいずれかが `OldLayerArn` 値と一致しないかどうかを確認します。一致しない場合、アサーションは true でルールは成功し、一致しない場合は失敗します。

`CONFIG_RULE_PARAMETERS` は、AWS Config ルールで設定される特別なパラメータセットです。この場合、`OldLayerArn` は `CONFIG_RULE_PARAMETERS` 内部のパラメータです。これにより、ユーザーは古い、または廃止されたと思われる特定の ARN 値を指定し、ルールはその古い ARN を使用する Lambda 関数がないかをチェックします。

フェーズ 2: 視覚化と設計

AWS Config は、設定データを収集し、そのデータを Amazon Simple Storage Service (Amazon S3) バケットに保存します。Amazon Athena を使用して、S3 バケットから直接このデータをクエリできます。Athena を使用すると、このデータを組織レベルで集約し、すべてのアカウントにわたるリソース構成の全体像を生成できます。リソース設定データの集約を設定するには、AWS クラウド運用と管理ブログの「Athena と Amazon QuickSight による AWS Config データの視覚化」を参照してください。

以下は、特定のレイヤー ARN を使用してすべての Lambda 関数を識別する Athena クエリのサンプルです。

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

クエリの結果は次のとおりです。

Query results | Query stats

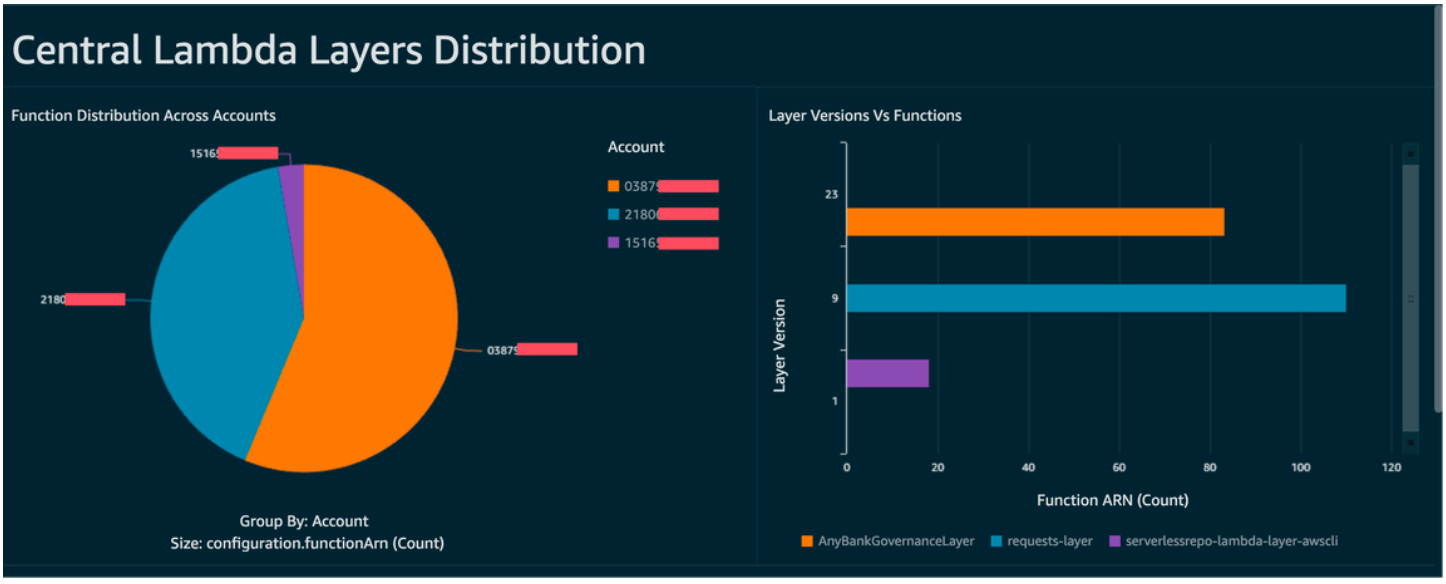
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

Results (27) — Copy — Download results

Search rows — 1

#	Region	FunctionName	memory_size	timeout	version
1	us-east-1	UpdateUIForPublishEvents	128	18	$LATEST
2	us-east-1	SchedulerCLI-InstanceSchedulerMain	128	300	$LATEST
3	us-east-1	my_functions_function10	128	3	$LATEST
4	us-east-1	lex-web-ui-CognitoIdentityP-CleanStackNameFunction-1TSORSH6L6YXQ	128	300	$LATEST
5	us-east-1	GetLatestArn	128	3	$LATEST
6	us-east-1	aws-python-http-api-project-dev-hello	1024	6	$LATEST
7	us-east-1	cloud9-MyTest-MyTest-688JGPVYP37L	128	15	$LATEST
8	us-east-1	my_functions_function1	128	3	$LATEST
9	us-east-1	my_functions_function25	128	3	$LATEST

組織全体で AWS Config データを集約したら、Amazon QuickSight を使用してダッシュボードを作成できます。Athena の結果を Amazon QuickSight にインポートすることで、Lambda 関数がレイヤーバージョンルールにどの程度準拠しているかを視覚化できます。このダッシュボードでは、次のセクションで説明するように、準拠しているリソースと準拠していないリソースをハイライト表示できるため、施行ポリシーを決定するのに役立ちます。以下の画像は、組織内の機能に適用されるレイヤーバージョンの分布をレポートするダッシュボードの例です。



フェーズ 3: 実装と適用

フェーズ 1 で作成したレイヤーバージョンルールを、Systems Manager 自動化ドキュメントによる修復アクションとオプションで組み合わせることができるようになりました。このドキュメントは、AWS SDK for Python (Boto3) で記述した Python スクリプトとして作成します。このスクリプトは Lambda 関数ごとに UpdateFunctionConfiguration API アクションを呼び出し、新しいレイヤー

ARN を使用して関数設定を更新します。あるいは、スクリプトでコードリポジトリにプルリクエストを送信し、レイヤー ARN を更新することもできます。これにより、今後のコードのデプロイも正しいレイヤー ARN で更新されます。

AWS Signer で Lambda コード署名を行う

AWS Signer は完全マネージド型のコード署名サービスで、コードをデジタル署名と照合して検証し、コードが変更されていないこと、信頼できるパブリッシャーからのものであることを確認できます。AWS Signer を AWS Lambda と併用して、AWS 環境へのデプロイ前に機能やレイヤーが変更されていないことを確認できます。これにより、認証情報を取得して新しい機能を作成したり、既存の機能を更新したりする悪意のある攻撃者から組織を保護できます。

Lambda 関数のコード署名を設定するには、まずバージョニングを有効にして S3 バケットを作成します。その後、AWS Signer を使用して署名プロファイルを作成し、プラットフォームに Lambda を指定し、署名プロファイルの有効日数を指定します。例：

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

次に、署名プロファイルを使用して、Lambda で署名設定を作成します。署名設定で、想定していたデジタル署名と一致しないアーティファクトが見つかった場合、「警告」(ただしデプロイは許可) または「強制」(デプロイをブロックする) のいずれかの対処方法を指定する必要があります。以下の例は、デプロイメントを強制し、ブロックするように設定されています。

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

これで、AWS Signer が信頼できないデプロイをブロックするよう Lambda で設定できました。機能リクエストのコーディングが完了し、関数をデプロイする準備ができたと仮定しましょう。最初のステップは、適切な依存関係を含めてコードを圧縮し、作成した署名プロファイルを使用してアーティファクトに署名することです。そのためには、zip アーティファクトを S3 バケットにアップロードし、署名ジョブを開始します。

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

jobId は送信先バケットで作成されるオブジェクトでプレフィックスであり、jobOwner はジョブが実行された AWS アカウント の 12 桁 ID である出力が得られます。

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

これで、署名された S3 オブジェクトと作成したコード署名設定を使用して関数をデプロイできます。

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

あるいは、元の署名されていないソース ZIP アーティファクトを使用して関数デプロイをテストすることもできます。デプロイに失敗し、以下のメッセージが表示されます。

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

AWS Serverless Application Model (AWS SAM) を使用して関数をビルドしてデプロイする場合、package コマンドは zip アーティファクトを S3 にアップロードし、署名ジョブを開始して署名されたアーティファクトを取得します。以下のコマンドとパラメータで、これを行えます。

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
```

```
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer は、アカウントにデプロイされた zip アーティファクトがデプロイ対象として信頼されていることを確認するのに役立ちます。上記のプロセスを CI/CD パイプラインに含め、前のトピックで説明した手法を使用してすべての機能にコード署名設定を添付することを要求できます。Lambda 関数のデプロイでコード署名を使用することにより、関数を作成または更新するための認証情報を取得した悪意のあるアクターが関数に悪意のあるコードを挿入するのを防ぐことができます。

Amazon Inspector を使用して Lambda のセキュリティ評価を自動化する

Amazon Inspector は、既知のソフトウェアの脆弱性や意図しないネットワークの露出について、ワークロードを継続的にスキャンする脆弱性管理サービスです。Amazon Inspector が生成する検出結果は、脆弱性を説明し、影響を受けるリソースを特定し、脆弱性の重要度を評価し、修正ガイダンスを提供します。

Amazon Inspector サポートにより、Lambda 関数とレイヤーのセキュリティ脆弱性評価が継続的かつ自動的に行われます。Amazon Inspector では、2 種類の Lambda スキャンを提供しています。

- Lambda 標準スキャン (デフォルト): Lambda 関数とそのレイヤー内のアプリケーションの依存関係をスキャンして、パッケージの脆弱性がないか調べます。
- Lambda コードスキャン: Lambda 関数内のカスタムアプリケーションコードをスキャンして、コードの脆弱性がないか調べます。Lambda 標準スキャンをアクティブ化することも、Lambda コードスキャンと同時に Lambda 標準スキャンをアクティブ化することもできます。

Amazon Inspector を有効にするには、Amazon Inspector コンソールに移動し、[設定] セクションを展開して [アカウント管理] を選択します。[アカウント] タブで [有効化] を選択し、スキャンオプションのいずれかを選択します。

Amazon Inspector をセットアップするときに、複数のアカウントで Amazon Inspector を有効にし、組織の Amazon Inspector を管理するためのアクセス権限を特定のアカウントに委任できます。有効にする際には、`AWSServiceRoleForAmazonInspector2` ロールを作成して Amazon Inspector にアクセス権限を付与する必要があります。Amazon Inspector コンソールで、ワンクリックオプションを使用してこのロールを作成できます。

Lambda の標準スキャンでは、Amazon Inspector は、次のような状況で Lambda 関数の脆弱性スキャンを開始します。

- Amazon Inspector が既存の Lambda 関数を検出した時。
- 新しい Lambda 関数をデプロイした時。
- 既存の Lambda 関数またはそのレイヤーのアプリケーションコードまたは依存関係に更新がデプロイされた場合。
- Amazon Inspector がデータベースに新しい共通脆弱性識別子 (CVE) 項目を追加し、その CVE が関数に関連している場合。

Lambda コードスキャンでは、Amazon Inspector は、自動推論と機械学習を使用して Lambda 関数のアプリケーションコードを評価し、アプリケーションコードを分析して全体的なセキュリティコンプライアンスを確認します。Amazon Inspector が Lambda 関数のアプリケーションコードに脆弱性を検出すると、Amazon Inspector は詳細な [コード脆弱性] の検出結果を生成します。可能な検出のリストについては、「Amazon CodeGuru Detector Library」を参照してください。

結果を確認するには、Amazon Inspector コンソールにアクセスしてください。[結果] メニューで [Lambda 関数別] を選択すると、Lambda 関数で実行されたセキュリティスキャン結果が表示されます。

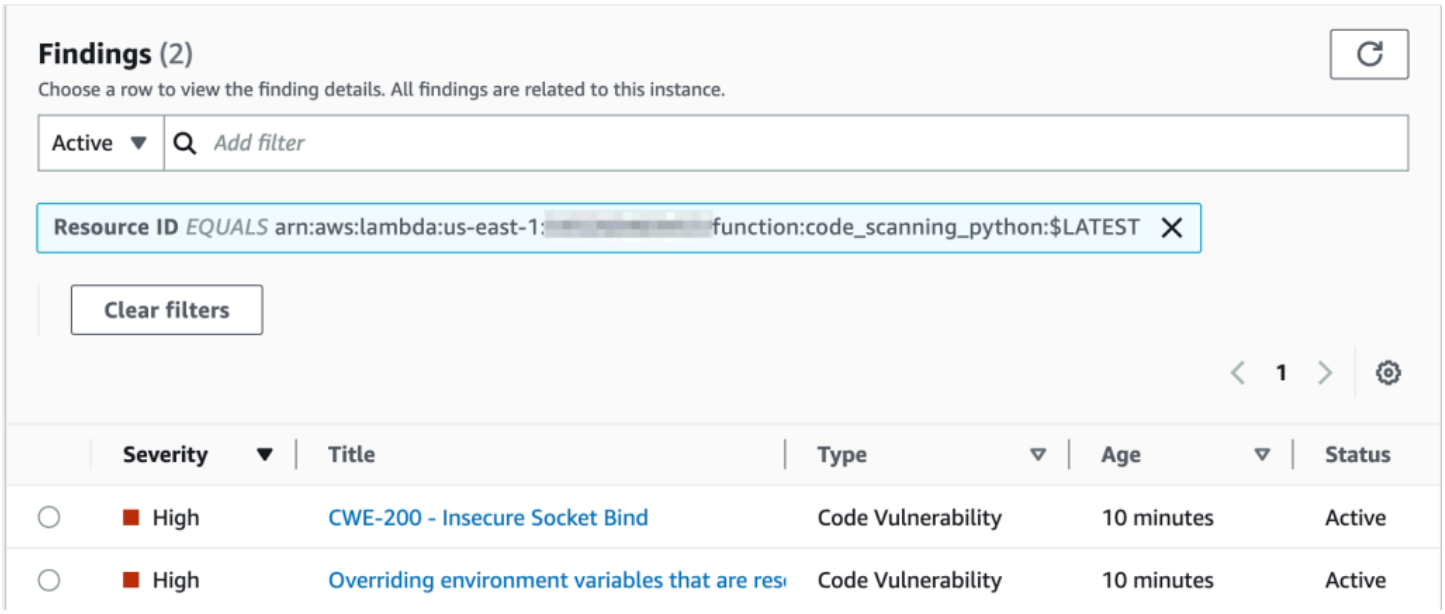
Lambda 関数を標準スキャンから除外するには、関数に次のキーと値のペアをタグ付けします。

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

コードスキャンから Lambda 関数を除外するには、関数に次のキーと値のペアをタグ付けします。

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

例えば、以下の画像では、Amazon Inspector は脆弱性を自動的に検出し、検出結果を [コード脆弱性] という種類に分類しています。これは、脆弱性がコード依存ライブラリのいずれにもなく、関数のコードにあることを示します。特定の機能または複数の機能について、これらの詳細を一度に確認できます。



これらの検出結果を 1 つずつ詳しく調べて、問題を解決する方法を知ることができます。

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

Lambda 関数を使用する際は、必ず Lambda 関数の命名規則に準拠するようにしてください。詳細については、「Lambda 環境変数を使用したコードの値の設定」を参照してください。

修復案を承認する責任はユーザーにあります。修復案を受け入れる前に、必ず確認してください。コードが意図したとおりに動作するように、修正案の編集が必要となる場合があります。

Lambda のセキュリティとコンプライアンスのためのオブザーバビリティの実装

AWS Config は、非準拠の AWS サーバーレスリソースを見つけて修正するのに便利なツールです。サーバーレスリソースに加えた変更はすべて AWS Config に記録されます。さらに、AWS Config では、設定スナップショットデータを S3 に保存できます。Amazon Athena と Amazon を使用してダッシュボード QuickSight を作成し、AWS Configデータを表示できます。AWS Config を使用して非準拠の Lambda デプロイと設定を検出する では、Lambda レイヤーのような特定の設定を視覚化する方法について説明しました。このトピックでは、これらの概念について詳しく説明します。

Lambda 設定の可視性

クエリを使用して、AWS アカウント ID、リージョン、AWS X-Ray トレーシング設定、VPC 設定、メモリサイズ、ランタイム、タグなどの重要な設定を取得できます。Athena からこの情報を取得するために使用できるクエリ例を次に示します。

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
```

```
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

クエリを使用して Amazon QuickSight ダッシュボードを構築し、データを視覚化できます。AWS リソース設定データを集約し、Athena でテーブルを作成し、Athena のデータに基づいて Amazon QuickSight ダッシュボードを構築するには、 AWS クラウド運用と管理ブログの「Athena と Amazon を使用したAWS Configデータの視覚化 QuickSight」を参照してください。特に、このクエリは関数のタグ情報も取得します。これにより、特にカスタムタグを使用する場合に、ワークロードと環境をより深く把握できます。

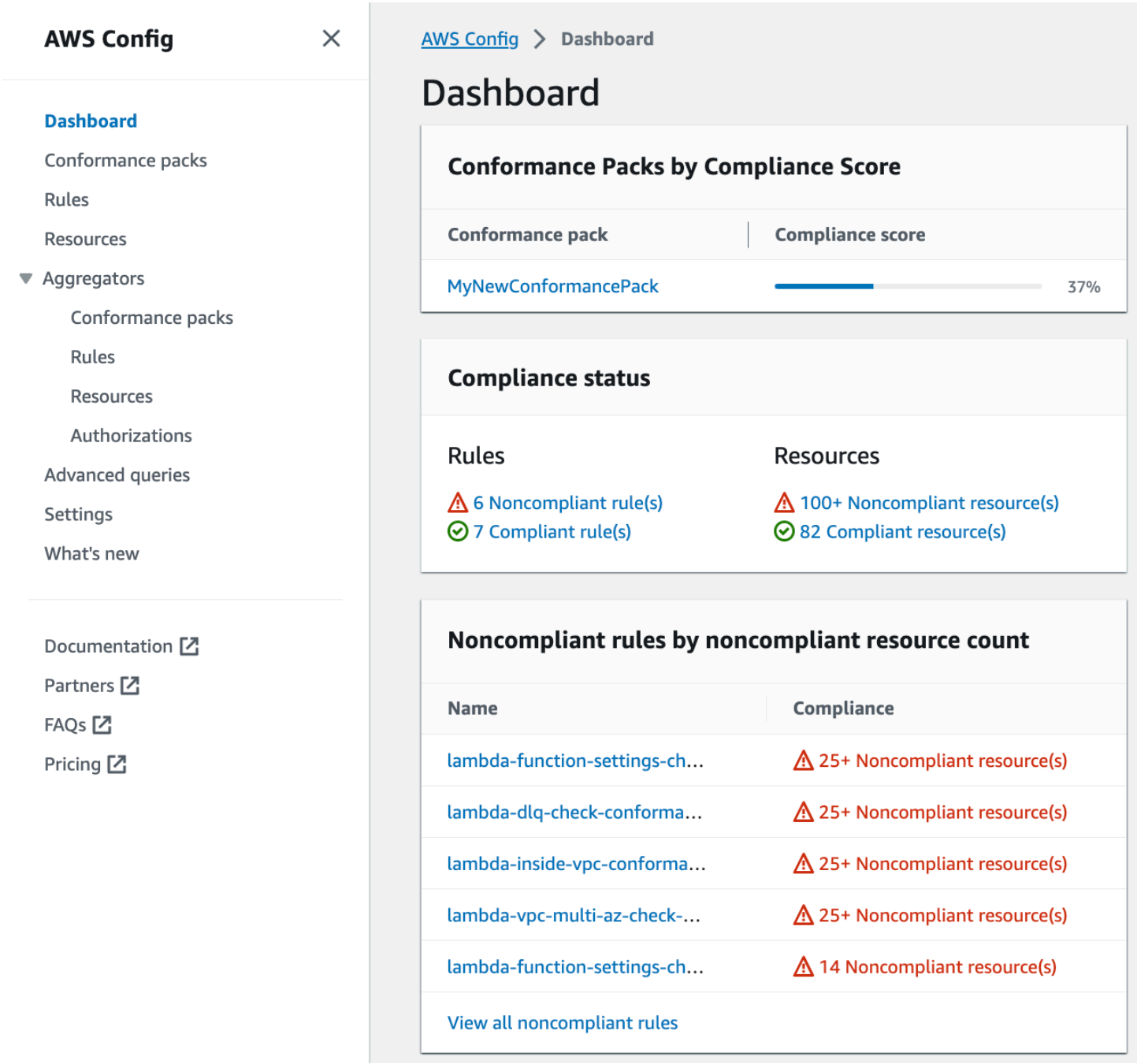
実行できるアクションの詳細については、このトピックで後述する オブザーバビリティに関する検出結果への対処 セクションを参照してください。

Lambda コンプライアンスの可視性

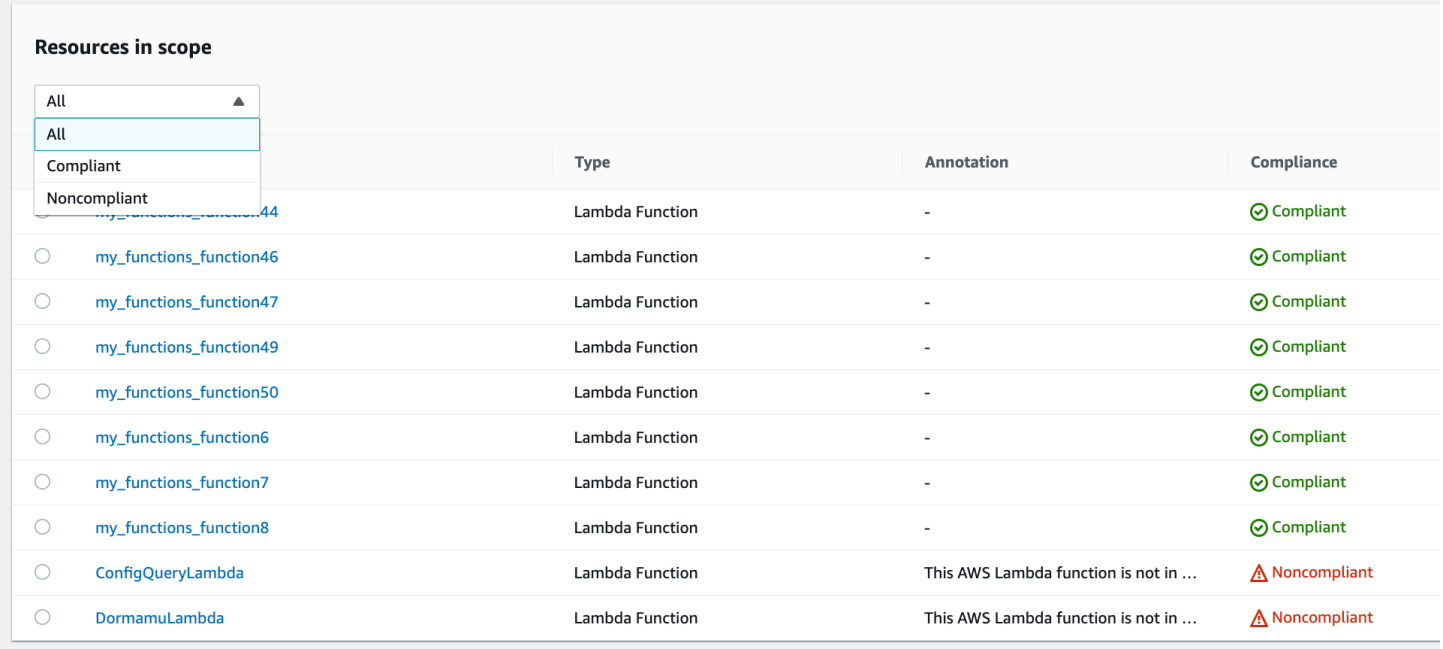
AWS Config で生成されたデータを使用して、コンプライアンスを監視するための組織レベルのダッシュボードを作成できます。これにより、以下の項目を一貫して追跡およびモニタリングできます。

- コンプライアンススコアによるコンプライアンスパック

- 非準拠リソースによるルール
- コンプライアンス状況

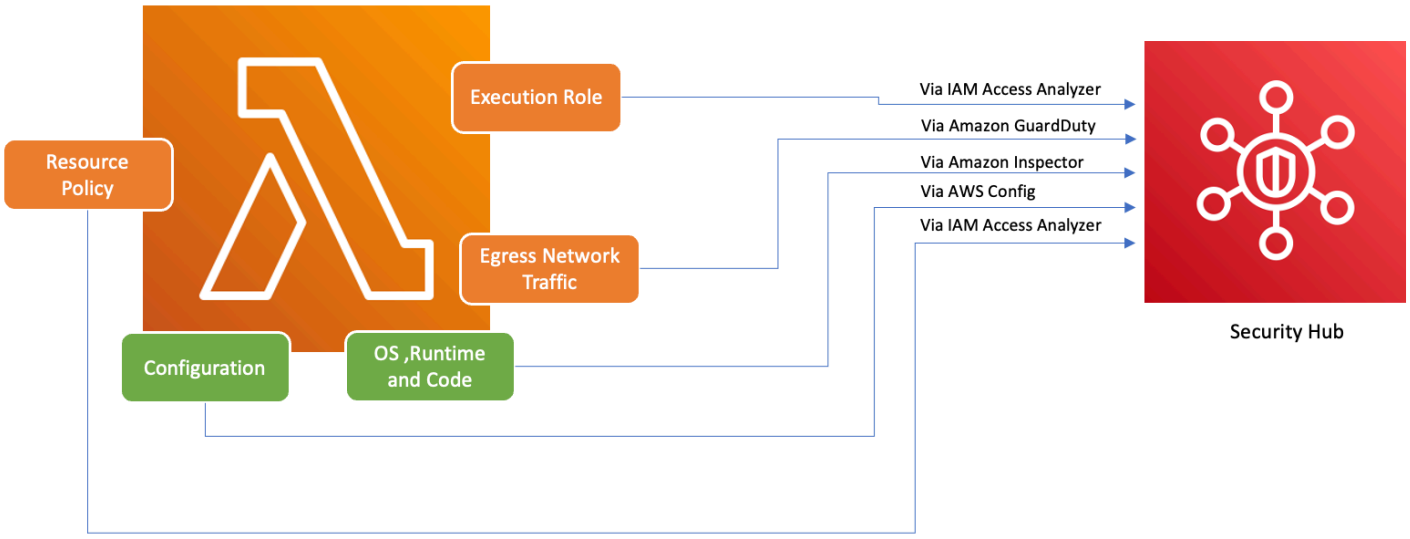


各ルールをチェックして、そのルールに適合していないリソースを特定します。例えば、組織ですべての Lambda 関数を VPC に関連付けることが義務付けられていて、コンプライアンスを識別する AWS Config ルールをデプロイしている場合は、上のリストから `lambda-inside-vpc` ルールを選択できます。



実行できるアクションの詳細については、以下の [オブザーバビリティに関する検出結果への対処] セクションを参照してください。

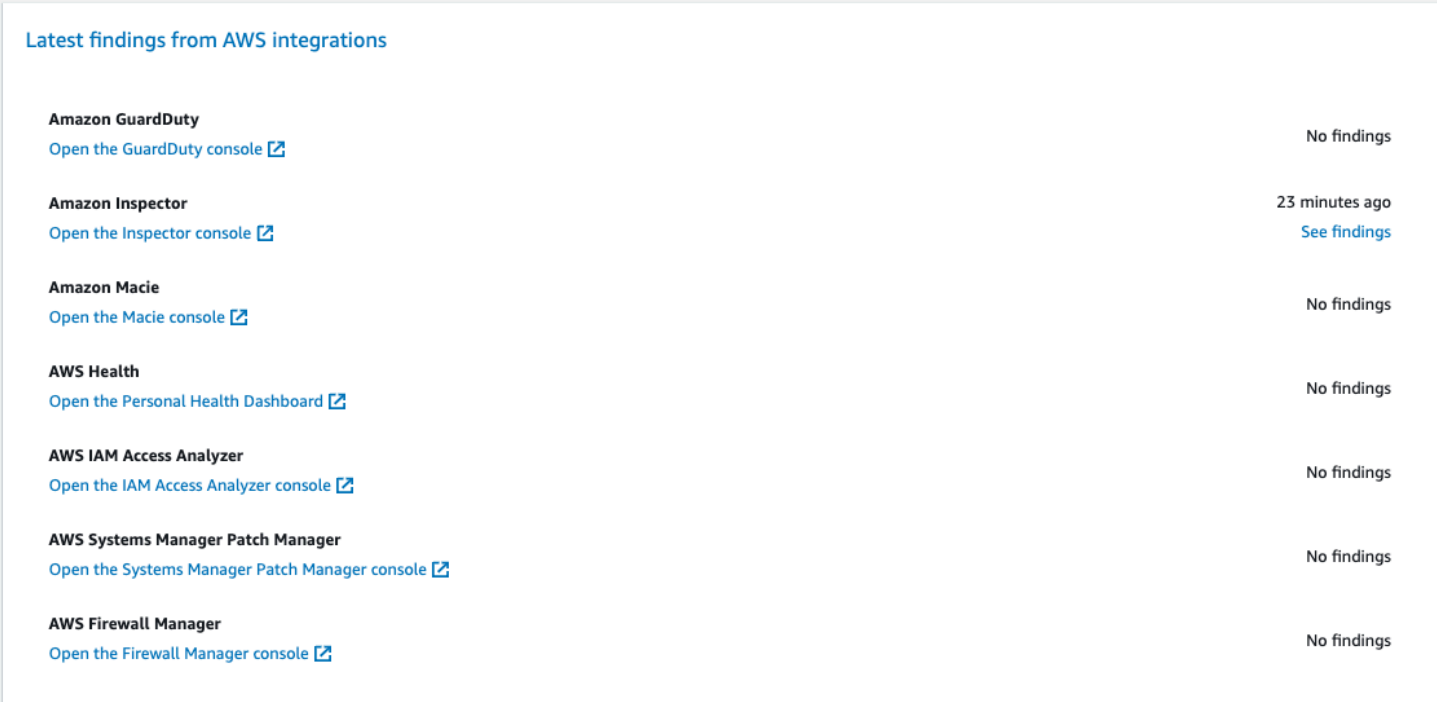
Security Hub を使用した Lambda 関数の境界の可視化



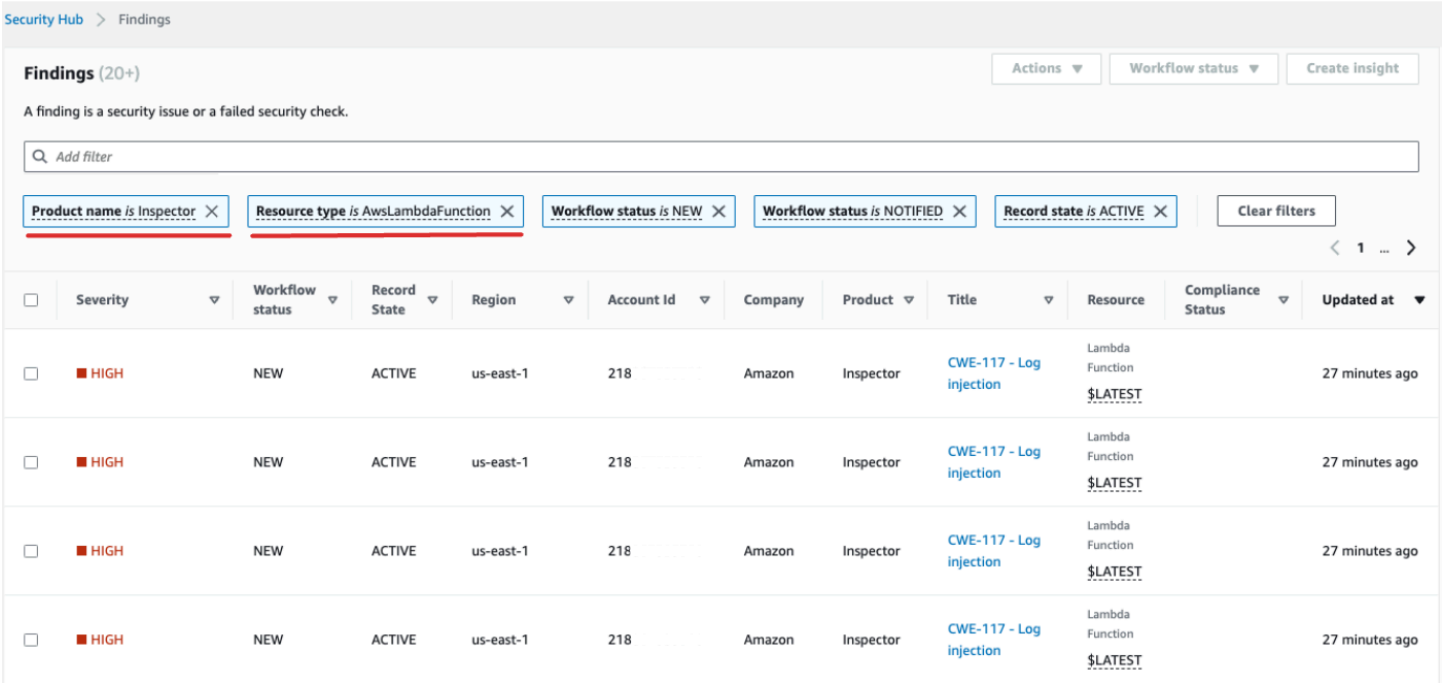
Lambda を含む AWS サービスを安全に使用するために、AWS は基礎セキュリティのベストプラクティス v1.0.0 を導入しました。この一連のベストプラクティスは、AWS 環境内のリソースとデータを保護するための明確なガイドラインを提供し、強固なセキュリティ体制を維持することの重要性を強調しています。AWS Security Hub は、セキュリティとコンプライアンスの統合センターを提供することで、これを補完しています。Amazon Inspector 、、Amazon などの複数の AWSのサービスか

らのセキュリティ結果を集約AWS Identity and Access Management Access Analyzer、整理、優先順位付けします GuardDuty。

Security Hub、Amazon Inspector、IAM Access Analyzer、および がAWS組織内で GuardDuty 有効になっている場合、Security Hub はこれらのサービスの結果を自動的に集約します。例えば、Amazon Inspector を考えてみましょう。Security Hub を使用すると、Lambda 関数のコードとパッケージの脆弱性を効率的に特定できます。Security Hub コンソールで、「AWS 統合からの最新の調査結果」というラベルの付いた一番下のセクションに移動します。ここでは、さまざまな統合 AWS サービスから得られた結果を表示して分析できます。



詳細を確認するには、2 番目の列の [結果を参照] リンクを選択します。これにより、Amazon Inspector などの製品別にフィルタリングされた結果のリストが表示されます。検索を Lambda 関数に限定するには、`ResourceType` を `AwsLambdaFunction` に設定します。これには、Lambda 関数に関連する Amazon Inspector からの結果が表示されます。



では GuardDuty、疑わしいネットワークトラフィックパターンを特定できます。このような異常は、Lambda 関数内に潜在的に悪意のあるコードが存在することを示唆している可能性があります。

IAM Access Analyzer を使用すると、ポリシー、特に外部エンティティへの関数アクセスを許可する条件ステートメントを含むポリシーを確認できます。さらに、IAM Access Analyzer は、Lambda API の AddPermissionオペレーションを と一緒に使用するときに設定されたアクセス許可を評価しますEventSourceToken。

オブザーバビリティに関する検出結果への対処

Lambda 関数の設定は多岐にわたり、要件も異なるため、標準化された修正自動化ソリューションがすべての状況に適しているとは限りません。さらに、変更の実装方法は環境によって異なります。準拠していないと思われる設定が見つかった場合は、以下のガイドラインを検討してください。

1. タグ付け戦略

   包括的なタグ付け戦略を実装することをお勧めします。各 Lambda 関数には、次のような重要な情報をタグ付けする必要があります。

   - オーナー: 関数の責任を負う個人またはチーム。
   - 環境: 本番、ステージング、開発、またはサンドボックス。
   - アプリケーション: この関数が属するより広いコンテキスト (該当する場合)。

2. オーナーへの働きかけ

重大な変更 (VPC 設定の調整など) は自動化せず、準拠していない機能 (所有者タグで識別) の所有者に事前に連絡し、次のいずれかを行うための十分な時間を確保してください。

- Lambda 関数の非準拠設定を調整します。
- 説明をして例外をリクエストするか、コンプライアンス基準を改定してください。

3. 構成管理データベース (CMDB) のメンテナンス

   タグはコンテキストをすぐに提供できますが、一元管理された CMDB を管理することでより深い洞察を得ることができます。各 Lambda 関数、依存関係、その他の重要なメタデータに関するより詳細な情報を保持できます。CMDB は、監査、コンプライアンスチェック、および関数所有者の特定において非常に貴重なリソースです。

サーバーレスインフラストラクチャの状況は絶えず進化しているため、監視に対して積極的な姿勢をとることが不可欠です。AWS Config、Security Hub、Amazon Inspector などのツールを使用すると、潜在的な異常や非準拠の設定を迅速に特定できます。ただし、ツールだけでは完全なコンプライアンスや最適な構成を保証することはできません。これらのツールを、十分に文書化されたプロセスやベストプラクティスと組み合わせることが重要です。

- フィードバックループ: 是正措置を講じたら、必ずフィードバックループを実施してください。つまり、コンプライアンス違反のリソースを定期的に見直して、更新されていないか、同じ問題がまだ発生していないかを確認します。
- 文書化: 観察結果、実施した措置、および認められた例外事項を必ず文書化してください。適切な文書化は、監査時に役立つだけでなく、今後のコンプライアンスとセキュリティを向上させるためのプロセスを強化するのにも役立ちます。
- トレーニングと啓発: すべての利害関係者、特に Lambda 関数の所有者に定期的にトレーニングを実施し、ベストプラクティス、組織ポリシー、コンプライアンス義務について周知徹底するようにします。定期的なワークショップ、ウェビナー、トレーニングセッションなどは、セキュリティとコンプライアンスに関して全員が同じ認識を持つようにするのに大いに役立ちます。

結論として、ツールやテクノロジーは潜在的な問題を検出して報告するための強力な機能を備えていますが、理解、コミュニケーション、トレーニング、文書化といった人的要素が極めて重要です。これらを合わせることで、Lambda 関数と広範なインフラストラクチャがコンプライアンス、安全性を維持し、ビジネスニーズに合わせて最適化されることを保証する強力な組み合わせになります。

AWS Lambda のコンプライアンス検証

サードパーティーの監査者は、複数の AWS Lambda コンプライアンスプログラムの一環として AWS のセキュリティとコンプライアンスを評価します。これらのプログラムには、SOC、PCI、FedRAMP、HIPAA などが含まれます。

特定のコンプライアンスプログラムの範囲内の AWS サービスのリストについては、「コンプライアンスプログラムによる AWS 対象範囲内のサービス」を参照してください。一般的な情報については、「AWS コンプライアンスプログラム」を参照してください。

AWS Artifact を使用して、サードパーティーの監査レポートをダウンロードできます。詳細については、AWS Artifact のレポートのダウンロードを参照してください。

Lambda を使用する際のお客様のコンプライアンス責任は、お客様のデータの機密性や貴社のコンプライアンス目的、適用可能な法律および規制によって決定されます。ガバナンスコントロールを実装して、会社の Lambda 関数がコンプライアンス要件を満たしていることを確認できます。詳細については、「Lambda 関数とレイヤーのガバナンス戦略を作成する」を参照してください。

AWS Lambda での耐障害性

AWS のグローバルインフラストラクチャは AWS リージョンとアベイラビリティーゾーンを中心に構築されます。AWSリージョンには、低レイテンシー、高いスループット、そして高度の冗長ネットワークで接続されている複数の物理的に独立し隔離されたアベイラビリティーゾーンがあります。アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS のリージョンとアベイラビリティーゾーンの詳細については、「AWS グローバルインフラストラクチャ」を参照してください。

Lambda では、AWS グローバルインフラストラクチャに加えて、データの耐障害性とバックアップのニーズに対応できるように複数の機能を提供しています。

- バージョニング - バージョニングは、Lambda で使用して、開発時に関数のコードと設定を保存できます。エイリアスと合わせてバージョニングを使用して、Blue/Green デプロイおよびローリングデプロイを実行できます。詳細については、「Lambda 関数のバージョン」を参照してください。

- スケーリング — 関数が前のリクエストの処理中にリクエストを受信すると、Lambda が、増えた負荷を処理するために、関数の別のインスタンスを起動します。Lambda は、リージョンごとに 1,000 回の同時実行を処理するように、自動的にスケーリングされます。クォータは必要に応じて増やすことができます。詳細については、「Lambda 関数のスケーリングについて」を参照してください。
- 高可用性 - Lambda は、複数のアベイラビリティーゾーンで関数を実行し、1 つのゾーンでサービスの中断が発生した場合にも、関数をイベントの処理に使用できることを保証します。お客様のアカウントで Virtual Private Cloud (VPC) に接続するように関数を設定する場合は、複数のアベイラビリティーゾーンでサブネットを指定することで、高可用性を確保します。詳細については、「Lambda 関数に Amazon VPC 内のリソースへのアクセスを許可する」を参照してください。
- リザーブド同時実行 - 関数が常にスケーリングして、追加リクエストを処理できるようにするため、関数に同時実行を予約できます。関数に予約された同時実行を設定することにより、指定した数の同時呼び出し数までスケーリングできますが、これを超えることはありません。これにより、利用可能なすべての同時実行数が他の関数に消費されているために、リクエストを失うことがありません。詳細については、「関数に対する予約済み同時実行数の設定」を参照してください。
- 非同期呼び出し - 非同期呼び出しと、他のサービスによってトリガーされた呼び出しのサブセットの場合、エラーが発生すると、Lambda は、再試行の間に遅延を置きながら、自動的に再試行します。関数を同期的に呼び出すその他のクライアントと AWS のサービスが、再試行の実行を担当します。詳細については、「Lambda での再試行動作について」を参照してください。
- デッドレターキュー - 非同期呼び出しでは、すべての再試行に失敗した場合、Lambda を設定してデッドレターキューにリクエストを送信することができます。デッドレターキューは、Amazon SNS のトピックまたはトラブルシューティングまたは再処理のためにイベントを受信する Amazon SQS キューです。詳細については、「デッドレターキュー」を参照してください。

AWS Lambda 内のインフラストラクチャセキュリティ

マネージドサービスである AWS Lambda は AWS グローバルネットワークセキュリティで保護されています。AWS セキュリティサービスと AWS がインフラストラクチャを保護する方法については、「AWS クラウドセキュリティ」を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して AWS 環境を設計するには、「セキュリティの柱 - AWS Well-Architected フレームワーク」の「インフラストラクチャ保護」を参照してください。

AWS が発行している API 呼び出しを使用して、ネットワーク経由で Lambda にアクセスします。クライアントは以下をサポートする必要があります:

- Transport Layer Security (TLS)。TLS 1.2 が必須です。TLS 1.3 が推奨されます。

- DHE (Ephemeral Diffie-Hellman) や ECDHE (Elliptic Curve Ephemeral Diffie-Hellman) などの Perfect Forward Secrecy (PFS) を使用した暗号スイート。これらのモードは、Java 7 以降など、ほとんどの最新システムでサポートされています。

また、リクエストには、アクセスキー ID と、IAM プリンシパルに関連付けられているシークレットアクセスキーを使用して署名する必要があります。または、AWS Security Token ServiceAWS STS を使用して、一時的なセキュリティ認証情報を生成し、リクエストに署名することもできます。

Lambda 関数のモニタリングおよびトラブルシューティング

AWS Lambda は他の AWS サービスとの統合により、Lambda 関数のモニタリングおよびトラブルシューティングに役立ちます。Lambda はユーザーの代わりに自動的に Lambda 関数をモニタリングし、Amazon CloudWatch を通じてメトリクスを報告します。Lambda は、コードを実行する際のコードのモニタリングに役立つように、リクエストの数、リクエストあたりの呼び出し時間、エラーとなったリクエストの数を自動的に追跡します。

他の AWS サービスを使用して、Lambda 関数のトラブルシューティングを行うことができます。このセクションでは、これらの AWS サービスを使用して、Lambda 関数とアプリケーションのモニタリング、トレース、デバッグ、トラブルシューティングを行う方法について説明します。各ランタイムでの関数のロギングとエラーに関する詳細については、それぞれのランタイムセクションを参照してください。

Lambda アプリケーションのモニタリングの詳細については、Serverless Land の「モニタリングとオブザーバビリティ」を参照してください。

セクション

- Lambda コンソールでの関数のモニタリング
- Lambda 関数のメトリクスの使用
- AWS Lambda での Amazon CloudWatch Logs の使用
- AWS CloudTrail を使用した AWS Lambda API コールのロギング
- AWS X-Ray を使用した Lambda 関数呼び出しの視覚化
- Amazon CloudWatch Lambda Insights を使用した関数パフォーマンスのモニタリング
- Lambda 関数での CodeGuru Profiler の使用
- 他の AWS サービスを使用するワークフローの例

Lambda コンソールでの関数のモニタリング

Lambda サービスは、ユーザーに代わって関数をモニタリングし、Amazon にメトリクスを送信します CloudWatch。Lambda コンソールにより、これらのメトリクスのモニタリンググラフが作成され、各 Lambda 関数の [Monitoring] (モニタリング) ページに表示されます。

Lambda コンソールでは、単一のペインでメトリクス、ログ、トレースを表示できます。コンソールには、すべてのペインに共通して適用される時間範囲、タイムゾーン、更新オプションのフィルターが用意されています。メトリクス、ログ、トレースを簡単に関連付けることができるため、Lambda 関数のエラーをトラブルシューティングする際の平均復旧時間 (MTTR、Mean Time to Recovery) を短縮できます。

料金

CloudWatch には期限なしの無料利用枠があります。無料利用枠のしきい値を超えると、 はメトリクス、ダッシュボード、アラーム、ログ、インサイトに対して CloudWatch 課金されます。詳細については、「Amazon の CloudWatch 料金」を参照してください。

Lambda コンソールを使用する

Lambda 関数とアプリケーションは、Lambda コンソールでモニタリングできます。

関数をモニタリングするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [Monitor] (モニタリング) タブを選択します。

モニタリンググラフのタイプ

次のセクションでは、Lambda コンソールのモニタリンググラフについて説明します。

Lambda モニタリンググラフ

- 呼び出し - 関数が呼び出された回数。
- [Duration] (期間) – 関数コードがイベントの処理に費やす平均、最小、および最大時間。
- [Error count and success rate (%)] (エラー数と成功率 (%)) – エラー数と、エラーなしで完了した呼び出しの割合。

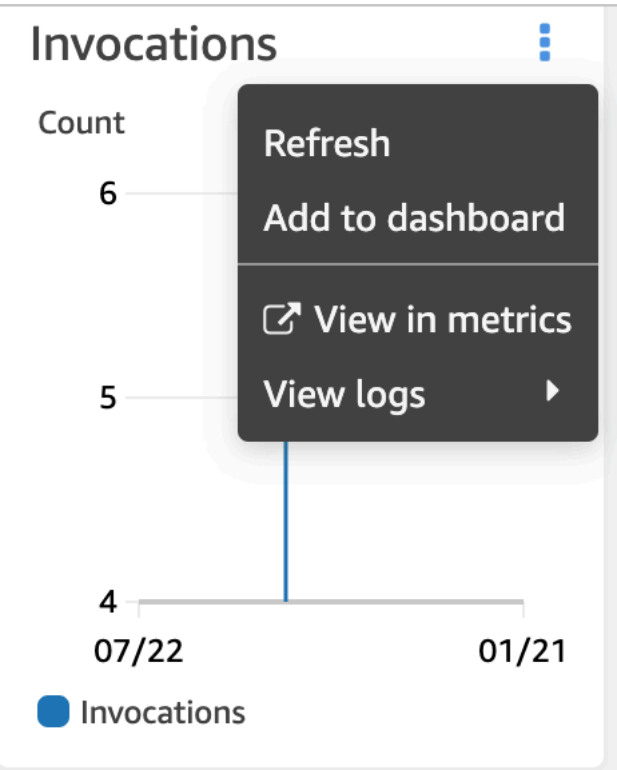
- [Throttles] (スロットル) – 同時実行制限が原因で失敗した呼び出しの回数。
- IteratorAge - ストリームイベントソースの場合、Lambda がバッチを受信して関数を呼び出したときのバッチ内の最後の項目の経過時間。
- 非同期配信エラー - 送信先キューまたはデッドレターキューへの書き込みを Lambda が試みたときに発生したエラーの数。
- 同時実行 - イベントを処理している関数インスタンスの数。

Lambda コンソールでのグラフの表示

次のセクションでは、Lambda コンソールで CloudWatch モニタリンググラフを表示し、CloudWatch メトリクスダッシュボードを開く方法について説明します。

関数のモニタリンググラフを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [Monitor] (モニタリング) タブを選択します。
4. 定義済みの時間範囲から選択するか、カスタムの時間範囲を選択します。
5. でグラフの定義を表示するには CloudWatch、3 つの縦のドット (Widget actions) を選択し、メトリクスで表示を選択して、 CloudWatch コンソールでメトリクスダッシュボードを開きます。

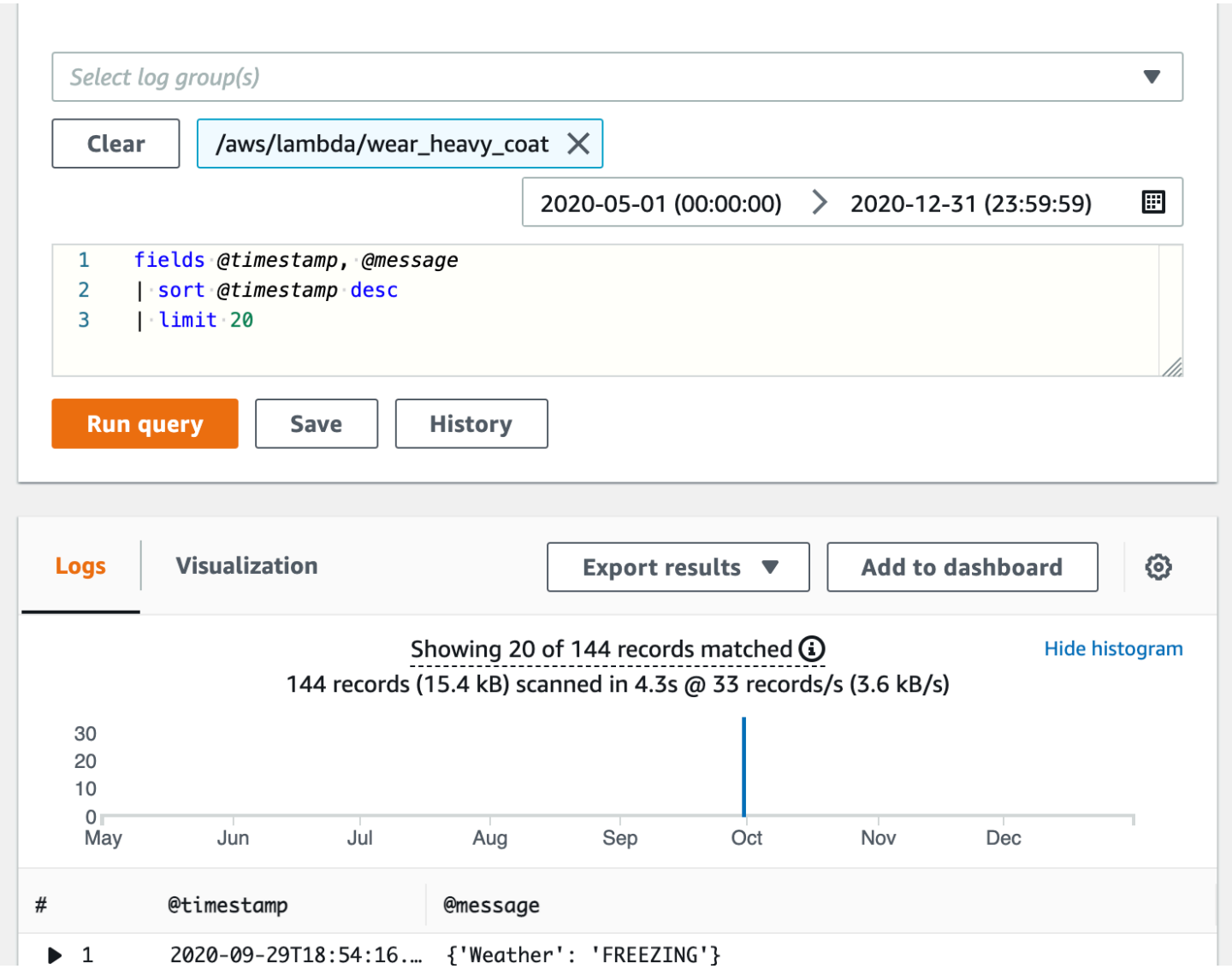


CloudWatch Logs コンソールでのクエリの表示

次のセクションでは、 CloudWatch Logs Insights から CloudWatch Logs コンソールのカスタムダッシュボードにレポートを表示および追加する方法について説明します。

関数のレポートを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [Monitor] (モニタリング) タブを選択します。
4. でログを表示する CloudWatchを選択します。
5. [View in Logs Insights] (Logs Insights で表示) を選択します。
6. 定義済みの時間範囲から選択するか、カスタムの時間範囲を選択します。
7. [Run query] (クエリの実行) を選択します。
8. (オプション) [Save] (保存) を選択します。



次のステップ

- Lambda が記録して に送信するメトリクスについては、 CloudWatch 「」を参照してくださいLambda 関数のメトリクスの使用。
- CloudWatch Lambda Insights を使用して、 で Lambda 関数のランタイムパフォーマンスメトリクスとログを収集および集計する方法について説明しますAmazon CloudWatch Lambda Insights を使用した関数パフォーマンスのモニタリング。

Lambda 関数のメトリクスの使用

AWS Lambda 関数がイベントの処理を終了すると、Lambda は呼び出しに関するメトリクスを Amazon CloudWatch に送信します。これらのメトリクスについては料金は発生しません。

CloudWatch コンソールでは、これらのメトリクスを使用してグラフとダッシュボードを作成できます。使用率、パフォーマンス、エラー率の変化に対応するようにアラームを設定できます。Lambda は、メトリクスデータを 1 分間隔で CloudWatch に送信します。Lambda 関数をより迅速に把握するには、Serverless Land で説明されている高解像度のカスタムメトリクスを作成できます。料金は、カスタムメトリクスと CloudWatch アラームに対して適用されます。詳細については、「Amazon CloudWatch 料金表」をご覧ください。

このページでは、CloudWatch コンソールで使用可能な Lambda 関数の呼び出し、パフォーマンス、および同時実行のメトリクスについて説明します。

セクション

CloudWatch コンソールでのメトリクスの表示

CloudWatch コンソールを使用して、関数名、エイリアス、またはバージョンで関数メトリクスをフィルターおよびソートできます。

CloudWatch コンソールでメトリクスを表示するには

1. CloudWatch コンソールで [Metrics] (メトリクス) ページ (AWS/Lambda 名前空間) を開きます。
2. [参照] タブの [メトリクス] で、次のいずれかのディメンションを選択します。
   - 関数名を基準 (FunctionName) - 関数のすべてのバージョンおよびエイリアスの集計メトリクスを表示します。
   - リソースを基準 (Resource) - 関数のバージョンまたはエイリアスのメトリクスを表示します。
   - 実行バージョンを基準 (ExecutedVersion) - エイリアスおよびバージョンの組み合わせのメトリクスを表示します。ExecutedVersion ディメンションを使用して、両方とも加重エイリアスのターゲットである 2 つのバージョンの関数のエラー率を比較します。
   - [全関数] (なし) - 現在の AWS リージョン 内のすべての関数の集計メトリクスを表示します。

3. メトリクスを選択し、[グラフに追加] または別のグラフオプションを選択します。

デフォルトでは、グラフはすべてのメトリクスで Sum 統計を使用します。別の統計を選択してグラフをカスタマイズするには、[Graphed metrics] タブのオプションを使用します。

> **Note**
>
> メトリクスのタイムスタンプには、関数が呼び出された時間が反映されます。呼び出し時間によっては、数分後にメトリクスが生成される場合があります。例えば、関数のタイムアウトが 10 分の場合は、正確なメトリクスを得るために過去 10 分以上を確認します。

Amazon CloudWatch の詳細については、Amazon CloudWatch ユーザーガイドを参照してください。

メトリクスの種類

次のセクションでは、CloudWatch コンソールで使用できる Lambda メトリクスのタイプについて説明します。

呼び出しメトリクス

呼び出しメトリクスは、Lambda 関数の呼び出しの結果を示すバイナリインジケータです。例えば、関数がエラーを返した場合、Lambda は値 1 の `Errors` メトリクスを送信します。1 分ごとに発生した関数エラーの数を取得するには、1 分間の `Errors` メトリクスの `Sum` を表示します。

> **Note**
>
> `Sum` 統計と共に、次の呼び出しメトリクスを表示します。

- `Invocations` – 関数コードが呼び出された回数 (成功した呼び出しや関数エラーが発生した呼び出しを含む)。呼び出しリクエストがスロットリングされた場合、呼び出しは記録されません。それ以外の場合は、呼び出しエラーになります。`Invocations` の値は請求対象リクエストの数に等しくなります。
- `Errors` - 関数エラーが発生した呼び出しの数。関数エラーには、コードによってスローされた例外と、Lambda ランタイムによってスローされた例外が含まれます。ランタイムは、タイムアウ

トや設定エラーなどの問題に対してエラーを返します。エラー率を計算するには、Errors の値を Invocations の値で割ります。エラーメトリクスのタイムスタンプは、エラーが発生した時点ではなく、関数が呼び出された時間を反映していることに注意してください。

- DeadLetterErrors – 非同期呼び出しの場合、Lambda がイベントをデッドレターキュー (DLQ) に送信しようとしたが、失敗した回数。デッドレターエラーは、リソースの設定ミス、またはサイズ制限が原因で発生する可能性があります。
- DestinationDeliveryFailures– 非同期呼び出しおよびサポートされているイベントソースマッピングの場合、Lambda がイベントを送信先に送信しようとして失敗した回数。イベントソースマッピングの場合、Lambda はストリームソース (DynamoDB および Kinesis) の送信先をサポートします。配信エラーは、アクセス許可エラー、リソースの設定ミス、またはサイズ制限が原因で発生する可能性があります。設定した送信先が Amazon SQS FIFO キューまたは Amazon SNS FIFO トピックなどのサポートされていないタイプの場合にも、エラーが発生する可能性があります。
- Throttles - スロットリングされた呼び出しリクエストの数。すべての関数インスタンスがリクエストを処理していて、スケールアップできる同時実行がない場合、Lambda は TooManyRequestsException エラーを出して追加のリクエストを拒否します。スロットリングされたリクエストやその他の呼び出しエラーは、Invocations または Errors のいずれかとしてカウントされません。
- OversizedRecordCount — Amazon DocumentDB イベントソースの場合における、関数が変更ストリームから受け取るイベントのうちサイズが 6 MB を超えるイベントの数。Lambda はメッセージをドロップし、このメトリクスを送信します。
- ProvisionedConcurrencyInvocations - プロビジョニングされた同時実行を使用して関数コードが呼び出された回数。
- ProvisionedConcurrencySpilloverInvocations - プロビジョニングされたすべての同時実行が使用されているときに、標準同時実行を使用して関数コードが呼び出された回数。
- RecursiveInvocationsDropped — 関数が無限再帰ループの一部であることが検出されたために Lambda が関数の呼び出しを停止した回数。Lambda 再帰ループ検出を使用した無限ループの防止 は、サポートされている AWS SDK によって追加されたメタデータを追跡することで、リクエストチェーンの一部として関数が呼び出された回数を監視します。関数がリクエストチェーンの一部として 16 回を超えて呼び出された場合、Lambda は次の呼び出しをドロップします。

パフォーマンスメトリクス

パフォーマンスメトリクスは、単一の関数呼び出しに関するパフォーマンスの詳細を提供します。たとえば、Duration メトリクスは、関数がイベントの処理に費やす時間をミリ秒単位で示します。

関数がイベントを処理する速度を把握するには、Average または Max 統計を使用してこれらのメトリクスを表示します。

- Duration - 関数コードがイベントの処理に費やす時間。呼び出しの請求期間は、最も近いミリ秒に切り上げた Duration の値です。Duration にコールドスタート時間は含まれません。
- PostRuntimeExtensionsDuration - 関数コードの完了後、拡張のためにランタイムがコードの実行に費やした累積時間。
- IteratorAge — DynamoDB、Kinesis、および Amazon DocumentDB イベントソースの場合における、イベントの最後のレコードの経過時間。このメトリクスは、ストリームがレコードを受信してから、イベント ソース マッピングがイベントを関数に送信するまでの時間を測定します。
- OffsetLag - セルフマネージド Apache Kafka および Amazon Managed Streaming for Apache Kafka (Amazon MSK) イベントソースの場合、トピックに書き込まれた最後のレコードと関数が処理した最後のレコードとのオフセットの差分。Kafka トピックは複数のパーティションを持つことができますが、このメトリクスはトピックレベルでのオフセット遅延を測定します。

また、Duration はパーセンタイル (p) 統計もサポートしています。Average 統計と Maximum 統計を歪める外れ値を除外するには、パーセンタイルを使用します。例えば、p95 統計は、呼び出しの 95% の最大所要時間を示します。ただし、最も遅い 5% は除きます。詳細については、「Amazon CloudWatch ユーザーガイド」の「パーセンタイル」を参照してください。

同時実行メトリクス

Lambda は、関数、バージョン、エイリアス、または AWS リージョン 全体でイベントを処理するインスタンスの総数として同時実行メトリクスを報告します。同時実行の制限に達するまであとどれくらいかを確認するには、Max 統計とともにこれらのメトリクスを表示します。

- ConcurrentExecutions - イベントを処理している関数インスタンスの数。この数値がリージョンの同時実行クォータ、または関数で設定した予約済み同時実行制限に達すると、Lambda は追加の呼び出しリクエストをスロットリングします。
- ProvisionedConcurrentExecutions - プロビジョニングされた同時実行を使用してイベントを処理している関数インスタンスの数。Lambda は、プロビジョニングされた同時実行を使用するエイリアスまたはバージョンの呼び出しごとに、現在の数を出力します。
- ProvisionedConcurrencyUtilization - バージョンまたはエイリアスの場合、ProvisionedConcurrentExecutions の値をプロビジョニングされた同時実行の合計量で割ります。例えば、関数に 10 のプロビジョニングされ

た同時実行数を設定して ProvisionedConcurrentExecutions が 7 の場合、ProvisionedConcurrencyUtilization は 0.7 になります。

- UnreservedConcurrentExecutions – リージョンの場合、同時実行が予約されていない関数によって処理されているイベントの数。
- ClaimedAccountConcurrency — リージョンに関する、オンデマンド呼び出しでは使用できない同時実行の量。ClaimedAccountConcurrency は、UnreservedConcurrentExecutions に割り当てられた同時実行数を加えたものに等しくなります (つまり、予約された同時実行数の合計にプロビジョニングされた同時実行数の合計を加えたもの)。詳細については、「ClaimedAccountConcurrency メトリクスの処理」を参照してください。

非同期呼び出しメトリクス

非同期呼び出しメトリクスは、イベントソースからの非同期呼び出しおよび直接呼び出しに関する詳細を提供します。しきい値とアラームを設定して、特定の変更を通知できます。例えば、処理用にキューに入れられるイベントの数が意図せず増えた場合 (AsyncEventsReceived) や、イベントが処理されるのを長時間待っていた場合 (AsyncEventAge) です。

- AsyncEventsReceived — Lambda が処理のために正常にキューに入れられたイベントの数。このメトリクスは、Lambda 関数が受け取るイベントの数に関するインサイトを提供します。このメトリクスをモニタリングし、しきい値のアラームを設定して問題がないか確認します。例えば、Lambda に送信された望ましくない数のイベントを検出したり、誤ったトリガーや関数設定に起因する問題を迅速に診断したりします。AsyncEventsReceived と Invocations が一致しない場合は、処理にばらつきがあるか、イベントがドロップされているか、キューが未処理になっている可能性があります。
- AsyncEventAge — Lambda がイベントを正常にキューに入れてから、関数が呼び出されるまでの時間。このメトリクスの値は、呼び出しの失敗またはスロットリングによってイベントが再試行されるときに増加します。このメトリクスをモニタリングし、キューの蓄積が発生したときのさまざまな統計のしきい値にアラームを設定します。このメトリクスの増加をトラブルシューティングするには、Errors メトリクスを調べて関数エラーを特定し、Throttles メトリクスを見て同時実行の問題を特定します。
- AsyncEventsDropped — 関数を正常に実行せずにドロップされたイベントの数。デッドレターキュー (DLQ) または OnFailure 送信先を設定すると、イベントはドロップされる前にそこに送信されます。イベントは、さまざまな理由でドロップされます。例えば、イベントがイベントの最大有効期間を超えたり、再試行回数が上限に達したり、予約された同時実行数が 0 に設定されたりすることがあります。イベントがドロップされる理由をトラブルシューティングするには、関数

エラーを特定する Throttlesメトリクスと、同時実行の問題を特定する Errors メトリクスを調べてください。

AWS Lambda での Amazon CloudWatch Logs の使用

AWS Lambda は、ユーザーに代わって Lambda 関数を自動でモニタリングし、関数の障害をトラブルシューティングするのに役立ちます。関数の実行ロールに必要なアクセス許可がある限り、Lambda は関数によって処理されたすべてのリクエストのログをキャプチャし、Amazon CloudWatch Logs に送信します。

コードが正常に動作しているかどうかを検証できるように、ログ記録ステートメントをコードに挿入できます。Lambda は、CloudWatch Logs と自動的に統合し、コードのすべてのログを、Lambda 関数に関連付けられた CloudWatch ロググループに送信します。

デフォルトでは、Lambda は `/aws/lambda/<function name>` という名前のロググループにログを送信します。関数から別のグループにログを送信する場合は、Lambda コンソール、AWS Command Line Interface (AWS CLI)、または Lambda API を使用してこれを設定できます。詳細については、「the section called “CloudWatch ロググループの設定”」を参照してください。

Lambda 関数のログは、Lambda コンソール、CloudWatch コンソール、AWS Command Line Interface(AWS CLI)、または CloudWatch API を使って表示することができます。

> **Note**
>
> 関数の呼び出し後にログが表示されるまで、5 ~ 10 分かかることがあります。

セクション

前提条件

実行ロールには、ログを CloudWatch Logs にアップロードするためのアクセス権限が必要です。CloudWatch Logs のアクセス許可は、Lambda が付与する

AWSLambdaBasicExecutionRoleAWS 管理ポリシーを使って追加します。このポリシーをロールに割り当てるには、次のコマンドを実行します。

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

詳細については、「the section called “AWS マネージドポリシー”」を参照してください。

料金

Lambda ログの使用には追加料金は発生しませんが、標準の CloudWatch Logs 料金が適用されます。詳細については、「CloudWatch 料金表」を参照してください。

Lambda 関数の高度なログ記録コントロールの設定

関数のログのキャプチャ、処理、使用方法をより細かく制御できるように、Lambda には以下のログ記録設定オプションが用意されています。

- ログの形式 - 関数のログをプレーンテキスト形式と構造化された JSON 形式から選択します
- ログレベル - JSON 構造化ログの場合、Lambda が CloudWatch に送信するログの詳細レベル (ERROR、DEBUG、INFO など) を選択します
- ロググループ - 関数がログを送信する CloudWatch ロググループを選択します

JSON とプレーンテキストのログフォーマットの設定

ログ出力を JSON キー値のペアとしてキャプチャすると、関数のデバッグ時の検索やフィルタリングが容易になります。JSON 形式のログでは、タグやコンテキスト情報をログに追加することもできます。これにより、大量のログデータを自動的に分析するのに役立ちます。開発ワークフローがプレーンテキストで Lambda ログを使用する既存のツールに依存している場合を除き、ログ形式には JSON を選択することをお勧めします。

すべての Lambda マネージドランタイムについて、関数のシステムログを CloudWatch Logs に非構造化プレーンテキストで送信するか、JSON 形式で送信するかを選択できます。システムログは Lambda が生成するログで、プラットフォームイベントログと呼ばれることもあります。

サポートされているランタイムでは、サポートされている組み込みログ記録メソッドのいずれかを使用すると、Lambda は関数のアプリケーションログ (関数コードが生成するログ) を構造化された JSON 形式で出力することもできます。これらのランタイムに対して関数のログ形式を設定すると、選択した設定がシステムログとアプリケーションログの両方に適用されます。

サポートされているランタイムでは、関数がサポートされているログ記録ライブラリまたはメソッドを使用している場合は、Lambda が構造化された JSON でログをキャプチャするために既存のコードを変更する必要はありません。

> **Note**
>
> JSON ログフォーマットを使用すると、メタデータが追加され、一連のキー値のペアを含む JSON オブジェクトとしてログメッセージがエンコードされます。そのため、関数のログメッセージのサイズが大きくなる可能性があります。

サポートされているランタイムとログ記録メソッド

Lambda は現在、以下のランタイムの JSON 構造化アプリケーションログを出力するオプションをサポートしています。

ランタイム	サポートバージョン
Java	Amazon Linux 1 上の Java 8 を除くすべての Java ランタイム
Node.js	Node.js 16 およびそれ以降
Python	Python 3.7 以降

Lambda が関数のアプリケーションログを構造化された JSON 形式で CloudWatch に送信するには、関数が以下の組み込みログ記録ツールを使用してログを出力する必要があります。

- Java - `LambdaLogger` ロガーまたは Log4j2。
- Node.js - コンソールメソッド `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error` および `console.warn`
- Python - 標準の Python `logging` ライブラリ

サポートされているランタイムで高度なログ記録コントロールを使用する方法の詳細については、「the section called “ログ記録”」、「the section called “ログ記録”」および「the section called “ログ記録”」を参照してください。

他のマネージド Lambda ランタイムについては、現在、Lambda は構造化された JSON 形式でのシステムログのキャプチャのみをネイティブでサポートしています。ただし、Powertools for AWS Lambda などのログ記録ツールを使用して JSON 形式のログを出力することで、どのランタイムでも構造化 JSON 形式のアプリケーションログをキャプチャできます。

既定のログ形式

現在、すべての Lambda ランタイムのデフォルトのログ形式はプレーンテキストです。

既に Powertools for AWS Lambda などのログ記録ライブラリを使用して JSON 構造化形式で関数ログを生成している場合は、JSON ログ形式を選択すればコードを変更する必要はありません。Lambda は既に JSON でエンコードされたログを二重にエンコードしないため、関数のアプリケーションログは以前と同様にキャプチャされます。

システムログの JSON 形式

関数のログ形式を JSON として設定すると、各システムログ項目 (プラットフォームイベント) は、以下のキーを含むキー値のペアを含む JSON オブジェクトとしてキャプチャされます。

- `"time"` - ログメッセージが生成された時刻
- `"type"` - 記録されるイベントのタイプ
- `"record"` - ログ出力の内容

`"record"` 値の形式は、記録されるイベントのタイプによって異なります。詳細については、「the section called “Telemetry API Event オブジェクトタイプ”」を参照してください。システムログイベントに割り当てられるログレベルの詳細については、「the section called “システムログレベルのイベントマッピング”」を参照してください。

比較のため、以下の 2 つの例は、プレーンテキスト形式と構造化された JSON 形式の両方で同じログ出力を示しています。ほとんどの場合、システムログイベントには、プレーンテキストで出力される場合よりも JSON 形式で出力される方が多くの情報が含まれることに注意してください。

Example プレーンテキスト:

```
2023-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.9.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example 構造化された JSON:

```
{
  "time": "2023-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.9.v18",
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> the section called “Telemetry API” は常に JSON 形式で START や REPORT などのプラットフォームイベントを送信します。Lambda が CloudWatch に送信するシステムログの形式を設定しても、Lambda Telemetry API の動作には影響しません。

アプリケーションログの JSON 形式

関数のログ形式を JSON として設定すると、サポートされているログ記録ライブラリとメソッドを使用して書き込まれたアプリケーションログ出力は、以下のキーを持つキー値のペアを含む JSON オブジェクトとしてキャプチャされます。

- `"timestamp"` - ログメッセージが生成された時刻
- `"level"` - メッセージに割り当てられたログレベル
- `"message"` - ログメッセージの内容
- `"requestId"` (Python および Node.js) または `"AWSrequestId"` (Java) - 関数呼び出しの一意のリクエスト ID

関数が使用するランタイムとログ記録方法によっては、この JSON オブジェクトには追加のキーペアが含まれる場合もあります。例えば、Node.js では、関数が `console` メソッドを使用して複数の引数を使用しているエラーオブジェクトをログに記録する場合、JSON オブジェクトには、`errorMessage`、`errorType`、`stackTrace` というキーを含む追加のキーと値のペアが含まれ

ます。さまざまな Lambda ランタイムの JSON 形式のログの詳細については、「the section called “ログ記録”、the section called “ログ記録”、the section called “ログ記録”」を参照してください。

> **Note**
>
> Lambda がタイムスタンプ値に使用するキーは、システムログとアプリケーションログでは異なります。システムログの場合、Lambda はキー `"time"` を使用して Telemetry API との一貫性を維持します。アプリケーションログについては、Lambda はサポートされているランタイムの規則に従い、`"timestamp"` を使用します。

比較のため、以下の 2 つの例は、プレーンテキスト形式と構造化された JSON 形式の両方で同じログ出力を示しています。

Example プレーンテキスト:

```
2023-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example 構造化された JSON:

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

関数のログ形式の設定

関数のログ形式を設定するには、Lambda コンソールまたは AWS Command Line Interface (AWS CLI) を使用できます。CreateFunction と UpdateFunctionConfiguration Lambda API コマンド、AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function リソース、および AWS CloudFormation AWS::Lambda::Function リソースを使用して、関数のログ形式を設定することもできます。

関数のログ形式を変更しても、CloudWatch Logs に保存されている既存のログには影響しません。新しいログのみが更新された形式を使用します。

関数のログ形式を JSON に変更し、ログレベルを設定しない場合、Lambda は関数のアプリケーションログレベルとシステムログレベルを自動的に INFO に設定します。つまり、Lambda はレベル

INFO 以下のログ出力のみを CloudWatch Logs に送信します。アプリケーションおよびシステムログレベルのフィルタリングの詳細については、the section called “ログレベルのフィルタリング” を参照してください。

> **Note**
>
> Python ランタイムでは、関数のログ形式がプレーンテキストに設定されている場合、デフォルトのログレベル設定は WARN です。つまり、Lambda は WARN 以下のレベルのログ出力のみを CloudWatch Logs に送信します。関数のログ形式を JSON に変更すると、このデフォルト動作が変わります。Python におけるログ記録の詳細については、「the section called “ログ記録”」を参照してください。

埋め込みメトリックフォーマット (EMF、Embedded Metric Format) ログを生成する Node.js 関数の場合、関数のログ形式を JSON に変更すると、CloudWatch がメトリクスを認識できなくなる可能性があります。

> **Important**
>
> 関数が Powertools for AWS Lambda (TypeScript) またはオープンソースの EMF クライアントライブラリを使用して EMF ログを生成する場合は、Powertools および EMF ライブラリを最新バージョンに更新して、CloudWatch が引き続きログを正しく解析できるようにしてください。JSON ログ形式に切り替える場合は、関数に埋め込まれているメトリクスとの互換性を確認するためのテストを実施することもお勧めします。EMF ログを生成する node.js 関数に関するその他のアドバイスについては、「the section called “構造化された JSON ログでの埋め込みメトリクスフォーマット (EMF、Embedded Metric Format) クライアントライブラリの使用”」を参照してください。

関数のログ形式を設定するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. 関数設定ページで、[モニタリングおよび運用ツール] を選択します。
4. [ログ記録設定] ペインで、[編集] を選択します。
5. [ログの内容] の [ログ形式] で [テキスト] または [JSON] を選択します。
6. [Save] を選択します。

既存の関数 (AWS CLI) のログ形式を変更するには

- 既存の関数のログ形式を変更するには、update-function-configuration コマンドを使用します。LoggingConfig の LogFormat オプションを JSON または Text に設定します。

```
aws lambda update-function-configuration \
--function-name myFunction --logging-config LogFormat=JSON
```

関数の作成時にログ形式を設定するには (AWS CLI)

- 新しい関数を作成するときにログ形式を設定するには、create-function コマンドの --logging-config オプションを使用します。LogFormat を JSON または Text に設定します。以下のコマンド例は、Node.js 18 ランタイムを使用して構造化 JSON でログを出力する関数を作成します。

  関数の作成時にログ形式を指定しない場合、Lambda は選択したランタイムバージョンのデフォルトのログ形式を使用します。デフォルトのログ記録形式の詳細については、「the section called “既定の口グ形式”」を参照してください。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --logging-config LogFormat=JSON
```

ログレベルのフィルタリング

Lambda は関数のログをフィルタリングして、特定の詳細レベル以下のログのみが CloudWatch Logs に送信されるようにすることができます。関数のシステムログ (Lambda が生成するログ) とアプリケーションログ (関数コードが生成するログ) には、ログレベルのフィルタリングを個別に設定できます。

the section called “サポートされているランタイムと口グ記録メソッド” では、Lambda が関数のアプリケーションログをフィルタリングするために、関数コードに変更を加える必要はありません。

その他のすべてのランタイムとログ記録メソッドでは、関数コードはキー "level" とキー値のペアを含む JSON 形式のオブジェクトとして stdout または stderr にログイベントを出力する必要があります。例えば、Lambda は以下の stdout への出力を DEBUG レベルのログとして解釈します。

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2023-11-02T16:51:31.587199Z"}')
```

"level" 値フィールドが無効または欠落している場合、Lambda はログ出力にレベル INFO を割り当てます。Lambda がタイムスタンプフィールドを使用するには、有効な RFC 3339 タイムスタンプ形式で時間を指定する必要があります。有効なタイムスタンプを指定しない場合、Lambda はログに INFO レベルを割り当ててタイムスタンプを追加します。

タイムスタンプキーに名前を付ける場合は、使用しているランタイムの規則に従ってください。Lambda は、マネージドランタイムで使用されるほとんどの一般的な命名規則をサポートしています。例えば、.NET ランタイムを使用する関数では、Lambda はキー "Timestamp" を認識します。

> **Note**
>
> ログレベルのフィルタリングを使用するには、JSON ログ形式を使用するように関数を設定する必要があります。現在、すべての Lambda マネージドランタイムのデフォルトのログ形式はプレーンテキストです。関数のログ形式を JSON に設定する方法については、「the section called “関数のログ形式の設定”」を参照してください。

アプリケーションログ (関数コードによって生成されるログ) については、以下のログレベルから選択できます。

ログレベル	標準的な使用状況
TRACE (最も詳細)	コードの実行パスを追跡するために使用される最も詳細な情報
DEBUG	システムデバッグの詳細情報
INFO	関数の通常の動作を記録するメッセージ
WARN	対処しないと予期しない動作を引き起こす可能性がある潜在的なエラーに関するメッセージ
ERROR	コードが期待どおりに動作しなくなる問題に関するメッセージ

ログレベル	標準的な使用状況
FATAL (詳細度が最も低い)	アプリケーションの機能停止を引き起こす重大なエラーに関するメッセージ

ログレベルを選択すると、Lambda はそのレベル以下のログを CloudWatch Logs に送信します。例えば、関数のアプリケーションログレベルを WARN に設定した場合、Lambda は INFO レベルと DEBUG レベルでログ出力を送信しません。ログフィルタリングのデフォルトのアプリケーションログレベルは INFO です。

Lambda が関数のアプリケーションログをフィルタリングすると、レベルのないログメッセージにはログレベル INFO が割り当てられます。

システムログ (Lambda サービスによって生成されるログ) については、以下のログレベルから選択できます。

ログレベル	使用方法
DEBUG (詳細度が最も高い)	システムデバッグの詳細情報
INFO	関数の通常の動作を記録するメッセージ
WARN (詳細度が最も低い)	対処しないと予期しない動作を引き起こす可能性がある潜在的なエラーに関するメッセージ

ログレベルを選択すると、Lambda はそのレベル以下のログを送信します。例えば、関数のシステムログレベルを INFO に設定した場合、Lambda は DEBUG レベルでログ出力を送信しません。

デフォルトでは、Lambda はシステムログレベルを INFO に設定します。この設定では、Lambda は自動的に CloudWatch に `"start"` および `"report"` ログメッセージを送信します。詳細度がより高いシステムログまたは詳細度がより低いシステムログを受信するには、ログレベルを DEBUG または WARN に変更します。Lambda がさまざまなシステムログイベントをマッピングするログレベルのリストを確認するには、「the section called “システムログレベルのイベントマッピング”」を参照してください。

ログレベルフィルタリングの設定

関数にアプリケーションとシステムログレベルのフィルタリングを設定するには、Lambda コンソールまたは AWS Command Line Interface (AWS CLI) を使用できます。CreateFunction と UpdateFunctionConfiguration Lambda API コマンド、AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function リソース、および AWS CloudFormation AWS::Lambda::Function リソースを使用して、関数のログレベルを設定することもできます。

関数のログレベルをコード内で設定した場合、この設定は他のログレベル設定よりも優先されることに注意してください。例えば、Python `logging` `setLevel()` メソッドを使用して関数のログレベルを INFO に設定した場合、この設定は Lambda コンソールを使用して設定した WARN の設定よりも優先されます。

既存の関数のアプリケーションまたはシステムログレベルを設定するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. 関数設定ページで、[モニタリングおよび運用ツール] を選択します。
4. [ログ記録設定] ペインで、[編集] を選択します。
5. [ログの内容] の [ログ形式] で [JSON] が選択されていることを確認します。
6. ラジオボタンを使用して、関数に必要な [アプリケーションログレベル] と [システムログレベル] を選択します。
7. [Save] を選択します。

既存の関数のアプリケーションまたはシステムログレベルを設定するには (AWS CLI)

- 既存の関数のアプリケーションログレベルまたはシステムログレベルを変更するには、`update-function-configuration` コマンドを使用します。`--system-log-level` を `DEBUG`、`INFO`、または `WARN` のいずれかに設定します。`--application-log-level` を `DEBUG`、`INFO`、`WARN`、`ERROR`、または `FATAL` のいずれかに設定します。

```
aws lambda update-function-configuration \
--function-name myFunction --system-log-level WARN \
--application-log-level ERROR
```

関数の作成時にログレベルのフィルタリングを設定するには

- 新しい関数を作成するときにログレベルのフィルタリングを設定するには、create-function コマンドの `--system-log-level` および `--application-log-level` オプションを使用します。`--system-log-level` を DEBUG、INFO、または WARN のいずれかに設定します。`--application-log-level` を DEBUG、INFO、WARN、WARN、または FATAL のいずれかに設定します。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --system-log-level WARN \
--application-log-level ERROR
```

システムログレベルのイベントマッピング

Lambda によって生成されるシステムレベルのログイベントについて、以下の表は各イベントに割り当てられるログレベルを定義しています。表に記載されているイベントの詳細については、「the section called “Event スキーマリファレンス”」を参照してください。

イベント名	条件	割り当てられたログレベル
initStart	runtimeVersion が設定されている	INFO
initStart	runtimeVersion が設定されていない	DEBUG
initRuntimeDone	status=success	DEBUG
initRuntimeDone	status!=success	WARN
initReport	initializationType=snapstart	INFO
initReport	initializationType!=snapstart	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion が設定されている	INFO

イベント名	条件	割り当てられたログレベル
restoreStart	runtimeVersion が設定されていない	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
start	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
レポート	status=success	INFO
レポート	status!=success	WARN
拡張子	state=success	INFO
拡張子	state!=success	WARN
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

Note

the section called “Telemetry API” は常にプラットフォームイベントの完全なセットを出力します。Lambda が CloudWatch に送信するシステムログのレベルを設定しても、Lambda Telemetry API の動作には影響しません。

カスタムランタイムによるアプリケーションログレベルのフィルタリング

関数にアプリケーションログレベルのフィルタリングを設定すると、バックグラウンドで Lambda は AWS_LAMBDA_LOG_LEVEL 環境変数を使用してランタイムのアプリケーションログレベルを設定します。また、Lambda は AWS_LAMBDA_LOG_FORMAT 環境変数を使用して関数のログ形式を設定します。これらの変数を使用して、Lambda の高度なログ記録コントロールをカスタムランタイムに統合できます。

Lambda コンソール、AWS CLI、および Lambda API でカスタムランタイムを使用して関数のログ記録設定を設定できるようにするには、これらの環境変数の値をチェックするようにカスタムランタイムを設定します。その後、選択したログ形式とログレベルに従ってランタイムのロガーを設定できます。

CloudWatch ロググループの設定

デフォルトでは、CloudWatch は関数が最初に呼び出されたときに /aws/lambda/<function name> という名前のロググループを自動的に作成します。既存のロググループにログを送信するように関数を設定したり、関数の新しいロググループを作成したりするには、Lambda コンソールまたは AWS CLI を使用できます。CreateFunction および UpdateFunctionConfiguration Lambda API コマンドと AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function リソースを使用してカスタムロググループを設定することもできます。

同じ CloudWatch ロググループにログを送信するように、複数の Lambda 関数を設定できます。例えば、1 つのロググループを使用して、特定のアプリケーションを構成するすべての Lambda 関数のログを保存できます。Lambda 関数にカスタムロググループを使用する場合、Lambda が作成するログストリームには関数名と関数バージョンが含まれます。これにより、同じロググループを複数の関数に使用しても、ログメッセージと関数の間のマッピングは保持されます。

カスタムロググループのログストリーム命名形式は以下の規則に従います。

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

カスタムロググループを設定する場合、ロググループに選択する名前は CloudWatch Logs の命名規則に従う必要があることに注意してください。また、カスタムロググループ名には文字列 aws/ で始まるものを使用できません。aws/ で始まるカスタムロググループを作成すると、Lambda はロググループを作成できなくなります。この結果、関数のログは CloudWatch に送信されなくなります。

関数のロググループを変更するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数を選択します。
3. 関数設定ページで、[モニタリングおよび運用ツール] を選択します。
4. [ログ記録設定] ペインで、[編集] を選択します。
5. [ログ記録グループ] ペインの [CloudWatch ロググループ] で、[カスタム] を選択します。
6. [カスタムロググループ] に、関数からのログの送信先にする CloudWatch ロググループの名前を入力します。既存のロググループの名前を入力すると、関数はそのグループを使用します。入力した名前のロググループが存在しない場合、Lambda はその名前で関数の新しいロググループを作成します。

関数のロググループを変更するには (AWS CLI)

- 既存の関数のロググループを変更するには、`update-function-configuration` コマンドを使用します。既存のロググループの名前を指定すると、関数はそのグループを使用します。指定した名前のロググループが存在しない場合、Lambda はその名前で関数の新しいロググループを作成します。

```
aws lambda update-function-configuration \
--function-name myFunction --log-group myLogGroup
```

関数の作成時にカスタムロググループを指定するには (AWS CLI)

- AWS CLI を使用して新しい Lambda 関数を作成するときにカスタムロググループを指定するには、`--log-group` オプションを使用します。既存のロググループの名前を指定すると、関数はそのグループを使用します。指定した名前のロググループが存在しない場合、Lambda はその名前で関数の新しいロググループを作成します。

  以下のコマンド例では、`myLogGroup` という名前のロググループにログを送信する Node.js Lambda 関数を作成します。

```
aws lambda create-function --function-name myFunction --runtime nodejs18.x \
--handler index.handler --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/LambdaRole --log-group myLogGroup
```

実行ロールのアクセス許可

関数が CloudWatch Logs にログを送信するには、その関数に `logs:PutLogEvents` アクセス許可が必要です。Lambda コンソールを使用して関数のロググループを設定するときに、関数にこのアクセス許可がない場合、Lambda はデフォルトでそれを関数の実行ロールに追加します。Lambda がこのアクセス許可を追加すると、任意の CloudWatch Logs ロググループにログを送信するアクセス許可が関数に付与されます。

Lambda が関数の実行ロールを自動的に更新しないようにして、代わりに手動で関数の実行ロールを編集するには、[アクセス許可] を展開し、[必要なアクセス許可を追加] のチェックを外します。

AWS CLI を使用して関数のロググループを設定しても、Lambda は `logs:PutLogEvents` アクセス許可を自動的に追加しません。まだアクセス許可がない場合は、関数の実行ロールにアクセス許可を追加します。このアクセス許可は、AWSLambdaBasicExecutionRole マネージドポリシーに含まれています。

Lambda コンソールでログにアクセスする

Lambda コンソールを使用してログを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [モニター] を選択します。
4. [CloudWatch のログを表示] を選択します。

AWS CLI を使用したログへのアクセス

AWS CLI は、コマンドラインシェルでコマンドを使用して AWS サービスとやり取りするためのオープンソースツールです。このセクションの手順を完了するには、以下が必要です。

- AWS Command Line Interface (AWS CLI) バージョン 2
- AWS CLI - `aws configure` によるクイック設定

AWS CLI および `--log-type` コマンドオプションを使用して、呼び出しのログを取得します。レスポンスには、`LogResult`フィールドが含まれ、このフィールドには、呼び出しから base64 コードされた最大 4 KB のログが含まれます。

Example ログ ID を取得します

次の例は、LogResultという名前の関数のmy-functionフィールドからログ ID を取得する方法を示しています。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

次のような出力が表示されます。

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example ログをデコードします

同じコマンドプロンプトで、base64 ユーティリティを使用してログをデコードします。次の例は、my-functionの base64 でエンコードされたログを取得する方法を示しています 。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

以下の出力が表示されます。

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

base64このユーティリティは、Linux、macOS、および Windows の Ubuntu で使用できます。macOS ユーザーは、base64 -Dを使用する必要があります 。

Example get-logs.sh スクリプト

同じコマンドプロンプトで、次のスクリプトを使用して、最後の 5 つのログイベントをダウンロードします。このスクリプトはsedを使用して出力ファイルから引用符を削除し、ログが使用可能になるまで15秒待機します。この出力には Lambda からのレスポンスと、get-log-events コマンドからの出力が含まれます。

次のコードサンプルの内容をコピーし、Lambda プロジェクトディレクトリに get-logs.sh として保存します。

AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。これをデフォルト設定にするには、aws configure set cli-binary-format raw-in-base64-out を実行します。詳細については、バージョン 2 の AWS Command Line Interface ユーザーガイドの「AWS CLI でサポートされているグローバルコマンドラインオプション」を参照してください。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS および Linux (専用)

同じコマンドプロンプトで、macOS と Linux ユーザーが次のコマンドを実行して、スクリプトが実行可能であることを確認する必要があります。

```
chmod -R 755 get-logs.sh
```

Example 最後の 5 つのログイベントを取得します

同じコマンドプロンプトで、次のスクリプトを実行して、最後の 5 つのログイベントを取得します。

```
./get-logs.sh
```

以下の出力が表示されます。

```
{
```

```
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

ランタイム関数のロギング

コードが期待どおりに動作していることをデバッグして検証するには、プログラミング言語の標準ログ機能を使用してログを出力します。Lambda ランタイムは、関数のログ出力を CloudWatch Logs にアップロードします。言語固有の手順については、以下のトピックをご参照ください。

- Node.js の AWS Lambda 関数ログ作成
- Python の AWS Lambda 関数ログ作成
- Ruby の AWS Lambda 関数ログ作成
- Java の AWS Lambda 関数ログ作成
- Go の AWS Lambda 関数ログ作成
- C# での Lambda 関数のログ記録
- PowerShell の AWS Lambda 関数ログ作成

次のステップ

- ロググループ、および、CloudWatch コンソールを通じてロググループにアクセスする方法の詳細については、Amazon CloudWatch ユーザーガイドのシステム、アプリケーション、カスタムログファイルのモニタリングを参照してください。

AWS CloudTrail を使用した AWS Lambda API コールのロギング

AWS Lambda は、ユーザー、ロール、または AWS のサービス によって実行されたアクションの記録を提供するサービスである AWS CloudTrail と統合されています。CloudTrail は Lambda に対する API コールをイベントとしてキャプチャします。キャプチャされた呼び出しには、Lambda コンソールの呼び出しと、Lambda API 操作へのコード呼び出しが含まれます。CloudTrail で収集された情報を使用して、Lambda に対するリクエスト、リクエスト元の IP アドレス、リクエストの作成日時、その他の詳細を確認できます。

各イベントまたはログエントリには、リクエストの生成者に関する情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます:

- ルートユーザーまたはユーザー認証情報のどちらを使用してリクエストが送信されたか
- リクエストが IAM Identity Center ユーザーに代わって行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが、別の AWS のサービス によって送信されたかどうか。

アカウントを作成すると、AWS アカウント で CloudTrail がアクティブになり、自動的に CloudTrail の[イベント履歴] にアクセスできるようになります。CloudTrail の [イベント履歴] では、AWS リージョン で過去 90 日間に記録された 管理イベントの表示、検索、およびダウンロードが可能で、変更不可能な記録を確認できます。詳細については、「AWS CloudTrail ユーザーガイド」の「CloudTrail イベント履歴の使用」を参照してください。[イベント履歴] の閲覧には CloudTrail の料金はかかりません。

AWS アカウント で過去 90 日間のイベントを継続的に記録するには、証跡または CloudTrail Lake イベントデータストアを作成します。

CloudTrail 証跡

証跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。AWS Management Console を使用して作成した証跡はマルチリージョンです。AWS CLI を使用する際は、単一リージョンまたは複数リージョンの証跡を作成できます。アカウント内のすべて AWS リージョン でアクティビティを把握するため、マルチリージョン証跡を作成することをお勧めします。単一リージョンの証跡を作成する場合、証跡の AWS リージョン に記録されたイベントのみを表示できます。証跡の詳細については、「AWS CloudTrail ユーザーガイド」の「AWS アカウント の証跡の作成」および「組織の証跡の作成」を参照してください。

証跡を作成すると、進行中の管理イベントのコピーを 1 つ無料で CloudTrail から Amazon S3 バケットに配信できますが、Amazon S3 ストレージには料金がかかります。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。Amazon S3 の料金に関する詳細については、「Amazon S3 の料金」を参照してください。

CloudTrail Lake イベントデータストア

CloudTrail Lake を使用すると、イベントに対して SQL ベースのクエリを実行できます。CloudTrail Lake は、行ベースの JSON 形式の既存のイベントを Apache ORC 形式に変換します。ORC は、データを高速に取得するために最適化された単票ストレージ形式です。イベントはイベントデータストアに集約されます。イベントデータストアは、高度なイベントセレクタを適用することによって選択する条件に基いた、イベントのイミュータブルなコレクションです。どのイベントが存続し、クエリに使用できるかは、イベントデータストアに適用するセレクタが制御します。CloudTrail Lake の詳細については、「AWS CloudTrail ユーザーガイド」の「Lake の使用AWS CloudTrail」を参照してください。

CloudTrail Lake のイベントデータストアとクエリにはコストがかかります。イベントデータストアを作成する際に、イベントデータストアに使用する料金オプションを選択します。料金オプションによって、イベントの取り込みと保存にかかる料金、および、そのイベントデータストアのデフォルトと最長の保持期間が決まります。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

CloudTrail の Lambda データイベント

データイベントでは、リソース上またはリソース内で実行されるリソースオペレーション (Amazon S3 オブジェクトの読み取りまたは書き込みなど) についての情報が得られます。これらのイベントは、データプレーンオペレーションとも呼ばれます。データイベントは、多くの場合、高ボリュームのアクティビティです。デフォルトで、ほとんどのデータイベントは CloudTrail でログ記録されません。また、CloudTrail のイベント履歴にも記録されません。

サポートされているサービスに対してデフォルトでログ記録される CloudTrail のデータイベントは、`LambdaESMDisabled` です。このイベントを使用して Lambda イベントソースマッピングの問題をトラブルシューティングする方法の詳細については、「the section called “CloudTrail を使用した無効な Lambda イベントソースのトラブルシューティング”」を参照してください。

追加の変更がイベントデータに適用されます。CloudTrail の料金の詳細については、「AWS CloudTrail の料金」を参照してください。

CloudTrail コンソール、AWS CLI、または CloudTrail API オペレーションを使用して、`AWS::Lambda::Function` リソースタイプのデータイベントを口グ記録できます。データイベントを口グに記録する方法の詳細については、「 AWS CloudTrailユーザーガイド」の「 AWS Management Consoleを使用したデータイベントの口グ記録」および「AWS Command Line Interfaceを使用したデータイベントの口グ記録」を参照してください。

次の表に、データイベントを口グに記録できる Lambda リソースタイプを示します。データイベントタイプ (コンソール) 列には、CloudTrail コンソールの[データイベントタイプ]リストから選択する値が表示されます。resources.type 値列には、AWS CLI または CloudTrail API を使用して高度なイベントセレクタを設定するときに指定する `resources.type` 値が表示されます。CloudTrail に記録されたデータ API 列には、リソース タイプの CloudTrail に口グ記録された API コールが表示されます。

データイベントタイプ (コンソール)	resources.type 値	CloudTrail に口グ記録されたデータ API
Lambda	`AWS::Lambda::Function`	Invoke

`eventName`、`readOnly`、および `resources.ARN` フィールドでフィルタリングして、自分にとって重要なイベントのみを口グに記録するように高度なイベントセレクタを設定できます。次の例は、特定の関数のみのイベントを口グ記録するデータイベント設定の JSON ビューです。オブジェクトの詳細については、「AWS CloudTrail API リファレンス」の「AdvancedFieldSelector」を参照してください。

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
```

```
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

CCloudTrail の Lambda 管理イベント

管理イベントでは、AWS アカウント のリソースに対して実行される管理オペレーションについての情報が得られます。これらのイベントは、コントロールプレーンオペレーションとも呼ばれます。CloudTrail は、デフォルトで管理イベントをログ記録します。

Lambda は、次のアクションを管理イベントとして CloudTrail ログファイルに記録することをサポートしています。

> **Note**
>
> CloudTrail ログファイルでは、 `eventName` に日付やバージョン情報が含まれる場合がありますが、同じ公開 API アクションを指しています。例えば、`GetFunction` のアクションが `GetFunction20150331v2` として表示される場合があります。次のリストは、イベント名が API アクション名と異なるタイミングを指定します。

- AddLayerVersionPermission
- AddPermission (イベント名: `AddPermission20150331v2`)
- CreateAlias (イベント名: `CreateAlias20150331`)
- CreateEventSourceMapping (イベント名: `CreateEventSourceMapping20150331`)
- CreateFunction (イベント名: `CreateFunction20150331`)

  (`Environment` および `ZipFile` パラメータは、`CreateFunction` では CloudTrail ログから省かれます)。
- CreateFunctionUrlConfig
- DeleteAlias (イベント名: `DeleteAlias20150331`)

- DeleteCodeSigningConfig
- DeleteEventSourceMapping (イベント名: DeleteEventSourceMapping20150331)
- DeleteFunction (イベント名: DeleteFunction20150331)
- DeleteFunctionConcurrency (イベント名: DeleteFunctionConcurrency20171031)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias (イベント名: GetAlias20150331)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion (イベント名: PublishLayerVersion20181031)

  (ZipFile パラメータは、PublishLayerVersion では CloudTrail ログから省かれます)。
- PublishVersion (イベント名: PublishVersion20150331)
- PutFunctionConcurrency (イベント名: PutFunctionConcurrency20171031)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission (イベント名: RemovePermission20150331v2)
- TagResource (イベント名: TagResource20170331v2)
- UntagResource (イベント名: UntagResource20170331v2)
- UpdateAlias (イベント名: UpdateAlias20150331)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping (イベント名: UpdateEventSourceMapping20150331)

- UpdateFunctionCode (イベント名: UpdateFunctionCode20150331v2)

  (ZipFile パラメータは、UpdateFunctionCode では CloudTrail ログから省かれます)。
- UpdateFunctionConfiguration (イベント名: UpdateFunctionConfiguration20150331v2)

  (Environment パラメータは、UpdateFunctionConfiguration では CloudTrail ログから省かれます)。
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

CloudTrail を使用した無効な Lambda イベントソースのトラブルシューティング

API アクション UpdateEventSourceMapping を使用してイベントソースマッピングの状態を変更すると、API コールは管理イベントとして CloudTrail でログ記録されます。イベントソースマッピングは、エラーが原因で Disabled 状態に直接移行することもあります。

次のサービスでは、イベントソースが Disabled 状態に移行する際、Lambda により CloudTrail に LambdaESMDisabled データイベントが発行されます。

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda は、他のイベントソースマッピングタイプでこのイベントをサポートしていません。

サポートされているサービスのイベントソースマッピングが Disabled 状態に移行した際にアラートを受信するには、CloudTrail イベント LambdaESMDisabled を使用して Amazon CloudWatch でアラームを設定します。CloudWatch アラームの設定についての詳細は、「Creating CloudWatch alarms for CloudTrail events: examples」を参照してください。

LambdaESMDisabled イベントメッセージの serviceEventDetails エンティティには、次のいずれかのエラーコードが含まれます。

RESOURCE_NOT_FOUND

リクエストで指定されたリソースは存在しません。

FUNCTION_NOT_FOUND

イベントソースにアタッチされている関数が存在しません。

REGION_NAME_NOT_VALID

イベントソースまたは関数に指定されたリージョン名が無効です。

AUTHORIZATION_ERROR

アクセス許可が設定されていないか、正しく設定されていません。

FUNCTION_IN_FAILED_STATE

関数コードがコンパイルされない、回復不能な例外が発生した、または不正なデプロイが発生しました。

Lambda イベントの例

各イベントは任意の送信元からの単一のリクエストを表し、リクエストされた API オペレーション、オペレーションの日時、リクエストパラメーターなどに関する情報を含みます。CloudTrail ログファイルは、パブリック API コールの順序付けられたスタックトレースではないため、イベントは特定の順序で表示されません。

以下は、GetFunction アクションと DeleteFunction アクションの CloudTrail ログエントリの例です。

> **Note**
>
> eventNameには、"GetFunction20150331"のような日付やバージョン情報が含まれることがありますが、同じ公開 API を指しています。

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
```

```
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
```

```
  ]
}
```

CloudTrail レコードの内容については、「AWS CloudTrail ユーザーガイド」の「CloudTrail record contents」を参照してください。

AWS X-Ray を使用した Lambda 関数呼び出しの視覚化

AWS X-Ray を使用して、アプリケーションのコンポーネントの視覚化、パフォーマンスのボトルネックの特定、およびエラーの原因となったリクエストのトラブルシューティングを行うことができます。Lambda 関数はトレースデータを X-Ray に送信し、X-Ray はデータを処理してサービスマップと検索可能なトレースサマリーを生成します。

関数を呼び出すサービスで X-Ray のトレースを有効にすると、トレースは Lambda から自動的に X-Ray に送信されます。Amazon API Gateway などのアップストリームサービスや、X-Ray SDK でインストルメントされた Amazon EC2 でホストされているアプリケーションは、着信リクエストをサンプリングし、トレースを送信するかどうかを Lambda に指示するトレースヘッダーを追加します。Amazon SQS などの上流のメッセージプロデューサーからのトレースは、下流の Lambda 関数からのトレースに自動的にリンクされるため、アプリケーション全体のエンドツーエンドのビューが作成されます。詳細については、「AWS X-Ray デベロッパーガイド」の「イベント駆動型アプリケーションのトレース」を参照してください。

> **Note**
>
> X-Ray トレースは、現在、Amazon Managed Streaming for Apache Kafka (Amazon MSK)、セルフマネージド Apache Kafka、ActiveMQ と RabbitMQ 向けの Amazon MQ、または Amazon DocumentDB イベントソースマッピングを備えた Lambda 関数ではサポートされていません。

コンソールを使用して、Lambda 関数のアクティブトレースをオンにするには、次のステップに従います。

アクティブトレースをオンにするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] を選択してから、[モニタリングおよび運用ツール] を選択します。
4. [編集] を選択します。
5. [X-Ray] で、[アクティブトレース] をオンに切り替えます。
6. [Save] を選択します。

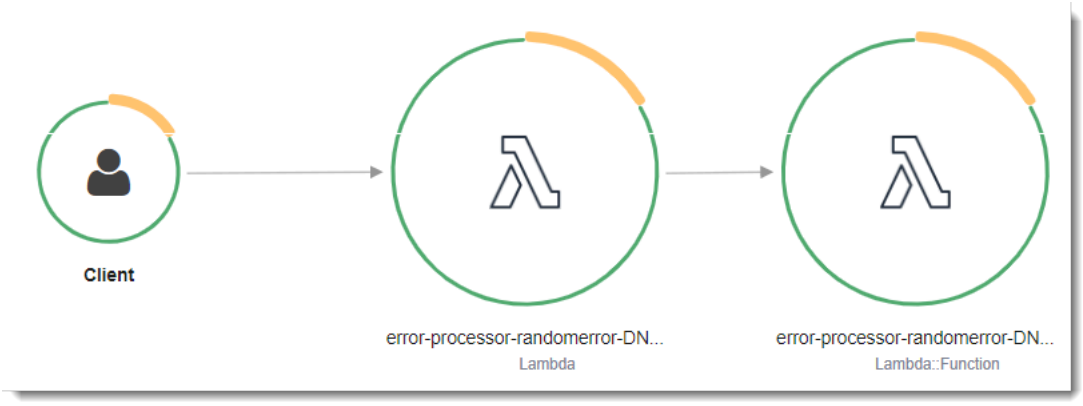
> **料金**
>
> X-Ray トレースは、毎月、AWS 無料利用枠で設定された一定限度まで無料で利用できます。X-Ray の利用がこの上限を超えた場合は、トレースによる保存と取得に対する料金が発生します。詳細については、「AWS X-Ray 料金表」を参照してください。

関数には、トレースデータを X-Ray にアップロードするためのアクセス許可が必要です。Lambda コンソールでトレースを有効にすると、Lambda は必要な権限を関数の [実行ロール] に追加します。それ以外の場合は、AWSXRayDaemonWriteAccess ポリシーを実行ロールに追加します。

X-Ray は、アプリケーションへのすべてのリクエストをトレースするとは限りません。X-Ray は、サンプリングアルゴリズムを適用することで効率的なトレースを行うと同時に、すべてのリクエストについての代表的なサンプルを示します。サンプルレートは 1 秒あたり 1 回のリクエストで、追加リクエストの 5% です。

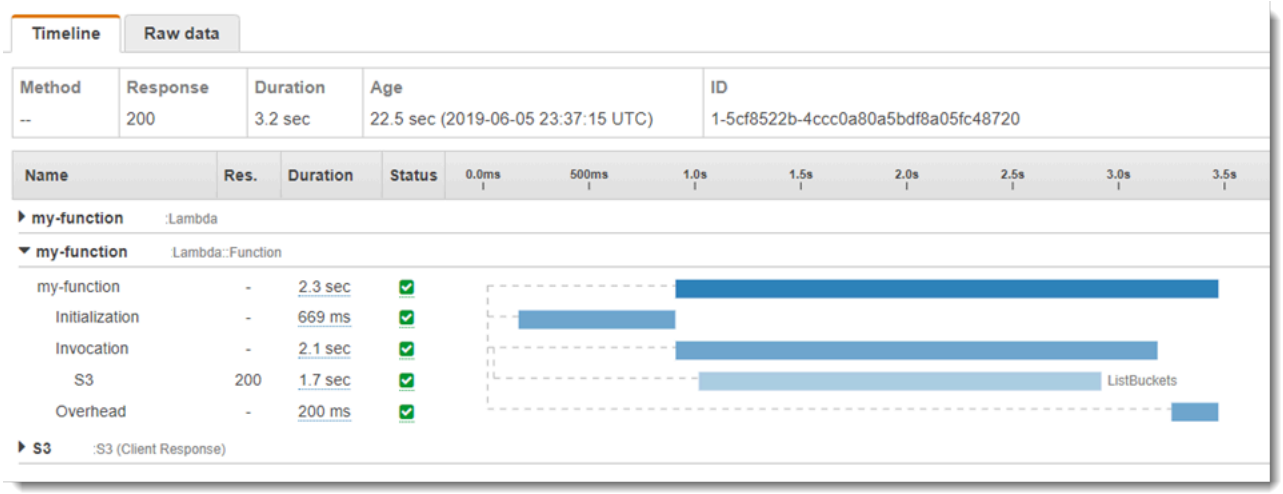
> **Note**
>
> 関数の X-Ray サンプルレートは設定することはできません。

X-Ray では、トレースは 1 つ以上のサービスによって処理されるリクエストに関する情報を記録します。Lambda はトレースごとに 2 つのセグメントを記録します。これにより、サービスグラフに 2 つのノードが作成されます。次の図は、これら 2 つのノードを強調表示しています。



左に示された 1 つめのノードは、呼び出しリクエストを受信する Lambda サービスを表しています。2 つめのノードは、特定の Lambda 関数を表しています。次の例は、これら 2 つのセグメントを使用したトレースを示しています。いずれも my-function と名付けられていますが、1 つは `AWS::Lambda` の起点があり、もう 1 つは `AWS::Lambda::Function` の起点がありま

す。AWS::Lambda セグメントにエラーが表示される場合は、Lambda サービスに問題があります。AWS::Lambda::Function セグメントにエラーが表示される場合、関数に問題があります。



関数セグメント (AWS::Lambda::Function) には、Initialization、Invocation、Restore (Lambda SnapStart のみ)、および Overhead のサブセグメントが含まれます。詳細については、「Lambda 実行環境のライフサイクル」を参照してください。

> **Note**
>
> X-Ray は、Lambda 関数内の未処理の例外を Error ステータスとして扱います。X-Ray が Fault ステータスを記録するのは、Lambda で内部サーバーエラーが発生した場合のみです。詳細については、「X-Ray Developer Guide」(X-Ray デベロッパーガイド) の「Errors, faults, and exceptions」(エラー、障害、および例外) を参照してください。

Initialization サブセグメントは、Lambda 実行環境ライフサイクルの初期フェーズを表します。このフェーズ中、Lambda は、設定したリソースを使用して実行環境を作成またはフリーズ解除し、関数コードとすべてのレイヤーをダウンロードして、拡張機能とランタイムの初期化、関数の初期化コードを実行します。

Invocation サブセグメントは呼び出しフェーズを現し、Lambda は関数ハンドラーを呼び出します。これはランタイムと拡張機能の登録から始まり、ランタイムが応答を送信する準備ができたら完了します。

(Lambda SnapStart のみ) Restore サブセグメントは、Lambda がスナップショットを復元し、ランタイム (JVM) をロードして、afterRestore ランタイムフックを実行するのにかかる時間を表しています。スナップショットを復元するプロセスには、MicroVM 外部でのアクティビティに費やす時

間が含まれる場合があります。この時間は、Restore サブセグメントで報告されます。MicroVM 外部でスナップショットの復元に費やした時間については課金されません。

Overhead サブセグメントは、ランタイムが応答を送信してから、次の呼び出しのシグナルを送信するまでの間に発生するフェーズを表します。この間、ランタイムは呼び出しに関連するすべてのタスクを終了し、サンドボックスをフリーズする準備をします。

> **Note**
>
> 時折、X-Ray トレースの関数初期化フェーズと呼び出しフェーズの間に大きなギャップがあることに気付くことがあります。プロビジョニングされた同時実行を使用する関数の場合、これが発生するのは Lambda が呼び出しに先立って関数インスタンスを初期化するからです。予約されていない (オンデマンド) 同時実行を使用する関数の場合、Lambda は、呼び出しがない場合でも先を見越して関数インスタンスを初期化することがあります。視覚的には、これらのケースの両方が、初期化フェーズと呼び出しフェーズ間における時間のギャップとして表示されます。

> **Important**
>
> Lambda では、X-Ray SDK を使用して、ダウンストリーム呼び出し、注釈、およびメタデータ用の追加のサブセグメントで、Invocation サブセグメントを拡張できます。関数セグメントに直接アクセスしたり、ハンドラーの呼び出しスコープの外で行われた作業を記録したりすることはできません。

Lambda のトレースに関する各言語での概要については、次のトピックを参照してください。

- AWS Lambda での Node.js コードの作成
- AWS Lambda での Python コードの作成
- AWS Lambda での Ruby の作成
- AWS Lambda での Java コードの作成
- AWS Lambda での Go コードの作成
- AWS Lambda での C# コードの作成

アクティブなインストルメンテーションをサポートするサービスの詳細なリストについては、AWS X-Ray デベロッパーガイドの Supported AWS services を参照してください。

セクション

実行ロールのアクセス許可

Lambda には、トレースデータを X-Ray に送信するために次のアクセス許可が必要です。これを関数の実行ロールに追加します。

- xray:PutTraceSegments
- xray:PutTelemetryRecords

これらのアクセス許可は、AWSXRayDaemonWriteAccess 管理ポリシーに含まれています。

AWS X-Ray デーモン

トレースデータを X-Ray API に直接送信する代わりに、X-Ray SDK はデーモンプロセスを使用します。AWS X-Ray デーモンは、Lambda 環境で実行され、セグメントとサブセグメントを含む UDP トラフィックをリッスンするアプリケーションです。受信データをバッファリングし、バッチで X-Ray に書き込みます。これにより、呼び出しのトレースに必要な処理とメモリのオーバーヘッドが削減されます。

Lambda ランタイムでは、デーモンは関数の設定されているメモリの 3% または 16 MB のいずれか大きい方まで使用することができます。呼び出し中に関数のメモリが不足すると、ランタイムはデーモンプロセスを最初に終了してメモリを解放します。

デーモンプロセスは、Lambda によるフルマネージドプロセスであり、ユーザーが設定することはできません。関数呼び出しによって生成されたすべてのセグメントは、Lambda 関数と同じアカウントに記録されます。デーモンは、他のアカウントにリダイレクトするように設定できません。

詳細については、X-Ray デベロッパーガイド の The X-Ray daemon を参照してください。

Lambda API でのアクティブトレースの有効化

AWS CLI または AWS SDK を使用してトレース設定を管理するには、以下の API オペレーションを使用します。

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

以下の例の AWS CLI コマンドは、my-function という名前の関数に対するアクティブトレースを有効にします。

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

トレースモードは、関数のバージョンを公開するときのバージョン固有の設定の一部です。公開後のバージョンのトレースモードを変更することはできません。

AWS CloudFormation でのアクティブトレースの有効化

AWS CloudFormation テンプレート内で `AWS::Lambda::Function` リソースに対するアクティブトレースを有効化するには、`TracingConfig` プロパティを使用します。

Example function-inline.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` リソースに、`Tracing` プロパティを使用します。

Example template.yml - トレース設定

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
```

```
...
```

Amazon CloudWatch Lambda Insights を使用した関数パフォーマンスのモニタリング

Amazon CloudWatch Lambda Insights は、サーバーレスアプリケーションの Lambda 関数ランタイムパフォーマンスメトリクスとログを収集および集計します。このページでは、Lambda Insights を有効にして Lambda 関数に関する問題の診断に使用する方法について説明します。

セクション

Lambda Insights によるサーバーレスアプリケーションのモニタリング方法

CloudWatch Lambda Insights は、 で実行されているサーバーレスアプリケーション用のモニタリングおよびトラブルシューティングソリューションですAWS Lambda このソリューションでは、CPU 時間、メモリ、ディスク、ネットワーク使用率などのシステムレベルのメトリクスが収集、集約、要約されます。また、コールドスタートや Lambda ワーカーシャットダウンなどの診断情報が収集、集約、要約されるため、Lambda 関数に関する問題を特定し、迅速に解決できます。

Lambda Insights は、Lambda レイヤーとして提供される新しい CloudWatch Lambda 拡張機能を使用します。この拡張機能を、サポートされているランタイムで、Lambda 関数で有効にすると、システムレベルのメトリクスが収集され、その Lambda 関数の呼び出しごとに 1 つのパフォーマンスログイベントが発生します。CloudWatch は、埋め込みメトリクスフォーマットを使用して、ログイベントからメトリクスを抽出します。詳細については、「AWS Lambda 拡張機能の使用」を参照してください。

Lambda Insights レイヤーは、`/aws/lambda-insights/` ロググループ用に `CreateLogStream` および `PutLogEvents` を拡張します。

料金

Lambda 関数に対して Lambda Insights を有効にすると、Lambda Insights は関数ごとに 8 つのメトリックスを報告し、関数呼び出しごとに約 1 KB のログデータが CloudWatch に送信されます。料金は、Lambda Insights によって関数に関してレポートされたメトリックスとログに対してのみ発生します。最低料金やサービス使用義務はありません。関数が呼び出されない場合、Lambda Insights に対する支払いはありません。料金の例については、Amazon CloudWatch の料金を参照してください。

ランタイムのサポート

Lambda Insights は、Lambda 拡張機能をサポートする任意のランタイムで使用できます。

Lambda コンソールで Lambda Insights を有効にする

新規および既存の Lambda 関数で、Lambda Insights 拡張モニタリングを有効にできます。サポートされているランタイムの Lambda コンソールの関数で Lambda Insights を有効にすると、Lambda は Lambda Insights 拡張機能をレイヤーとして関数に追加し、関数の実行ロールに必要な `CloudWatchLambdaInsightsExecutionRolePolicy` ポリシーを検証するか、アタッチしようとします。

Lambda コンソールで Lambda Insights を有効にするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] タブを選択します。
4. 左側のメニューで [モニタリングおよび運用ツール] を選択します。
5. [その他の監視ツール] ペインで、[編集] を選択します。
6. [CloudWatch Lambda インサイト] で、[拡張モニタリング] をオンにします。
7. [Save] を選択します。

Lambda Insights をプログラムで有効にする

Lambda Insights は、AWS Command Line Interface (AWS CLI)、AWS Serverless Application Model (SAM) CLI、AWS CloudFormation、または、AWS Cloud Development Kit (AWS CDK)を使用して有効にすることもできます。Lambda Insights を、プログ

ラムを使って、サポートされているランタイムの関数で有効にすると、CloudWatch は、CloudWatchLambdaInsightsExecutionRolePolicy ポリシーを関数の実行ロールにアタッチします。

詳細については、Amazon CloudWatch ユーザーガイドの Lambda Insights を使用するを参照してください。

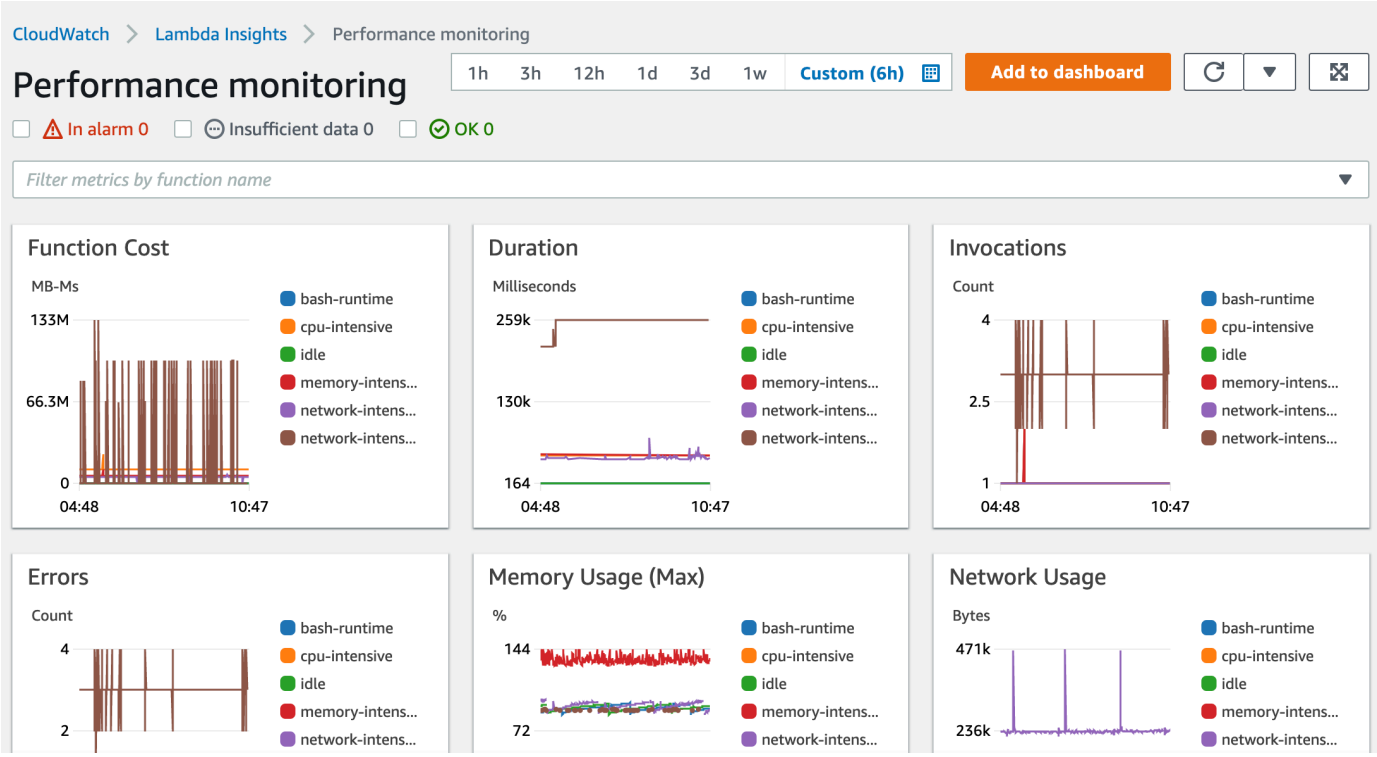
Lambda Insights ダッシュボードの使用

Lambda Insights のダッシュボードには、CloudWatch コンソールに multi-function overview と single-function view という 2 つの画面があります。multi-function overview では、現在の AWS アカウントとリージョンの Lambda 関数のランタイムメトリクスが集計されます。single-function view では、単一の Lambda 関数で使用可能なランタイムメトリクスが表示されます。

Lambda Insights ダッシュボードの CloudWatch コンソールにある multi-function overview を使用して、使用率の高い Lambda 関数と使用率の低い Lambda 関数を識別できます。Lambda Insights ダッシュボードの CloudWatch コンソールにある single-function view を使用して、個々のリクエストのトラブルシューティングを行うことができます。

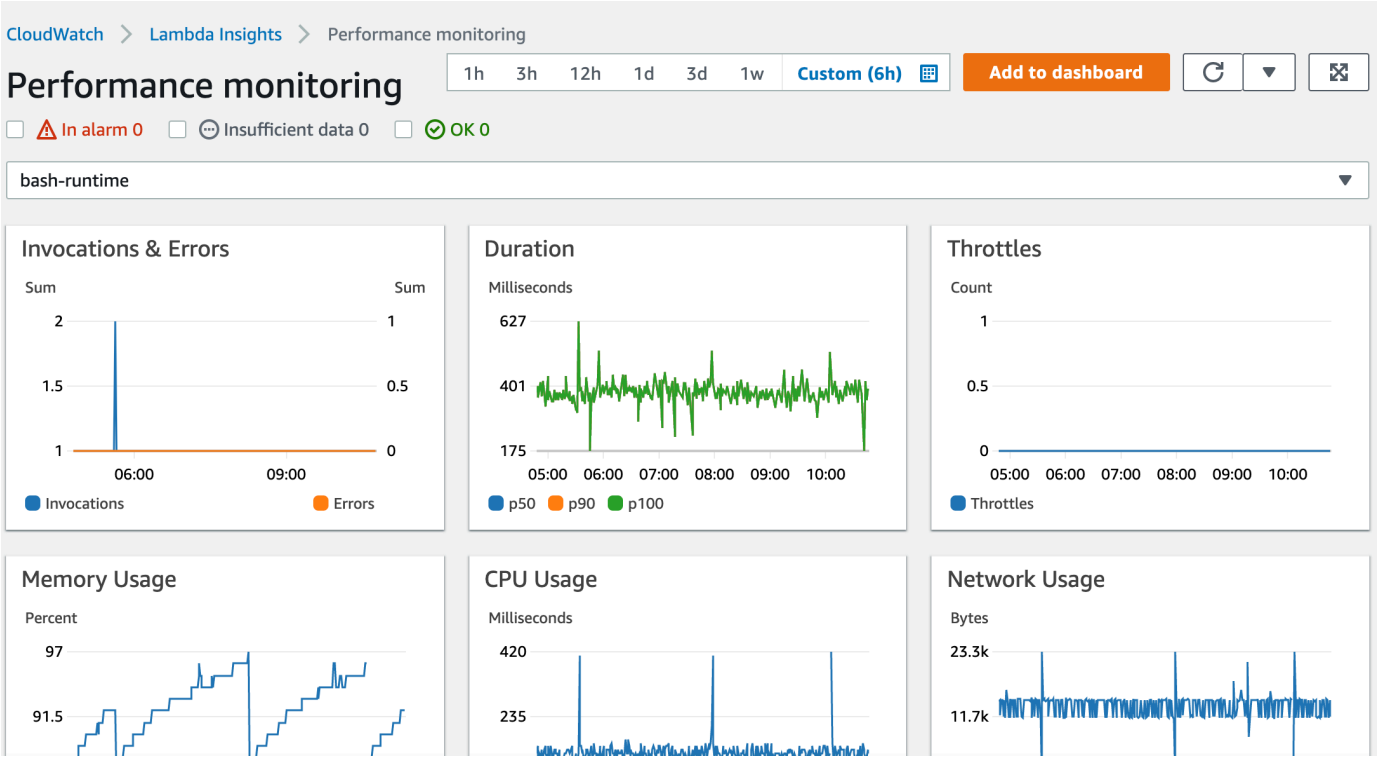
すべての関数のランタイムメトリクスを表示するには

1. CloudWatch コンソールで [Multi-function (複数の関数)] ページを開きます。
2. 定義済みの時間範囲から選択するか、カスタムの時間範囲を選択します。
3. (オプション) CloudWatch ダッシュボードにウィジェットを追加するには、[Add to dashboard (ダッシュボードに追加)] を選択します。



1 つの関数のランタイムメトリクスを表示するには

1. CloudWatch コンソールで [Single-function (単一の関数)] ページを開きます。
2. 定義済みの時間範囲から選択するか、カスタムの時間範囲を選択します。
3. (オプション) CloudWatch ダッシュボードにウィジェットを追加するには、[Add to dashboard (ダッシュボードに追加)] を選択します。



詳細については、CloudWatch ダッシュボードでのウィジェットの作成と操作を参照してください。

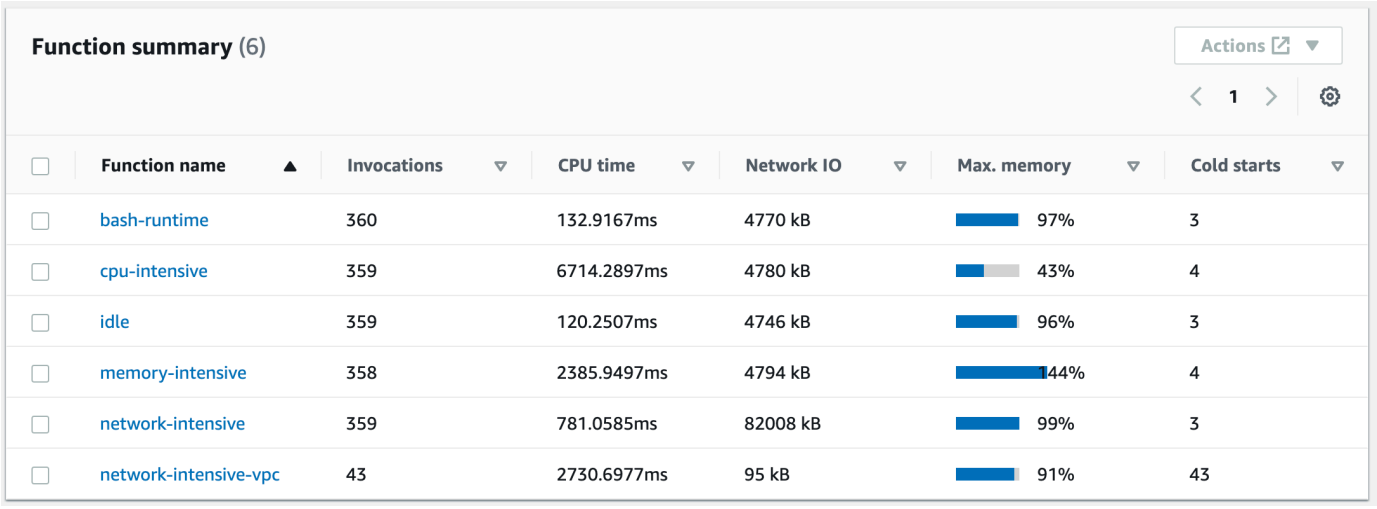
関数の異常を検出するワークフローの例

Lambda Insights ダッシュボードの multi-function overview を使用して、関数でのコンピュートメモリの異常を特定および検出できます。例えば、multi-function overview で、関数が大量のメモリを使用していることが示されている場合、[Memory Usage] ペインで詳細なメモリ使用率メトリクスを表示できます。その後、[Metrics] ダッシュボードに移動して、異常検出を有効にするか、アラームを作成できます。

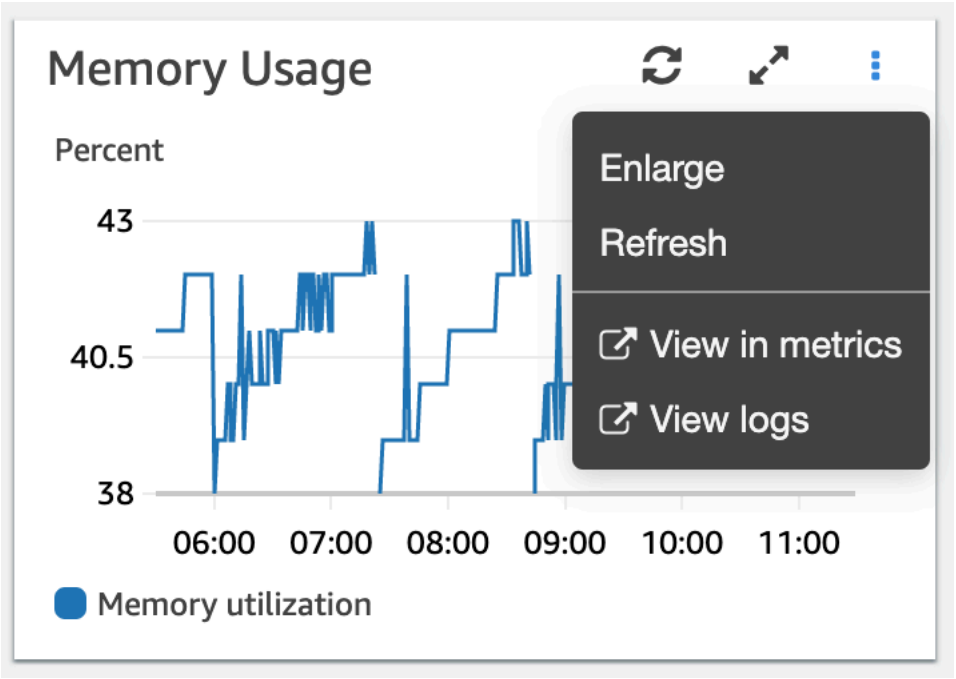
関数の異常検出を有効にするには

1. CloudWatch コンソールで [Multi-function (複数の関数)] ページを開きます。
2. [Function summary] で、関数の名前を選択します。

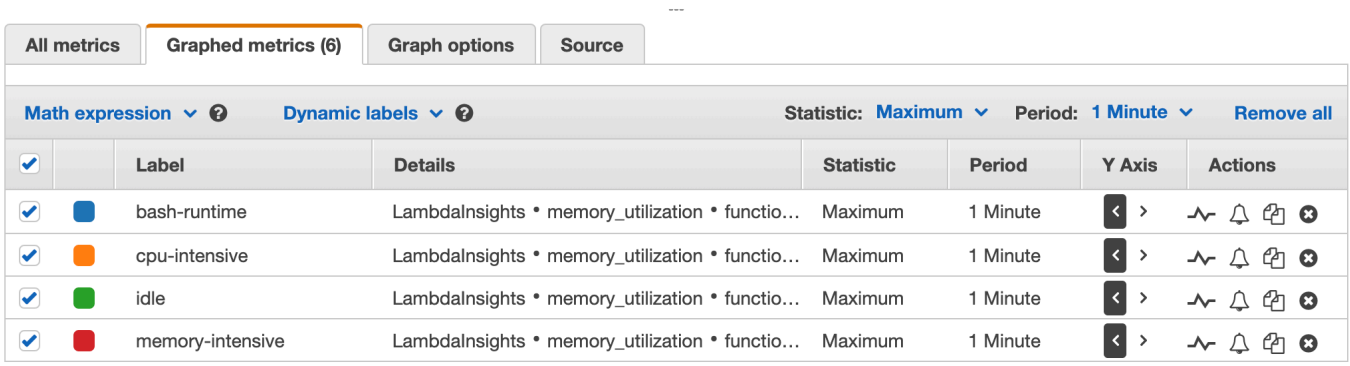
   single-function view が開き、関数のランタイムメトリクスが表示されます。



3. [Memory Usage] ペインで、3 つの縦のドットを選択し、[View in metrics] を選択して [Metrics] ダッシュボードを開きます。



4. [Graphed Metrics] タブの [Actions] 列で、最初のアイコンを選択して関数の異常検出を有効にします。



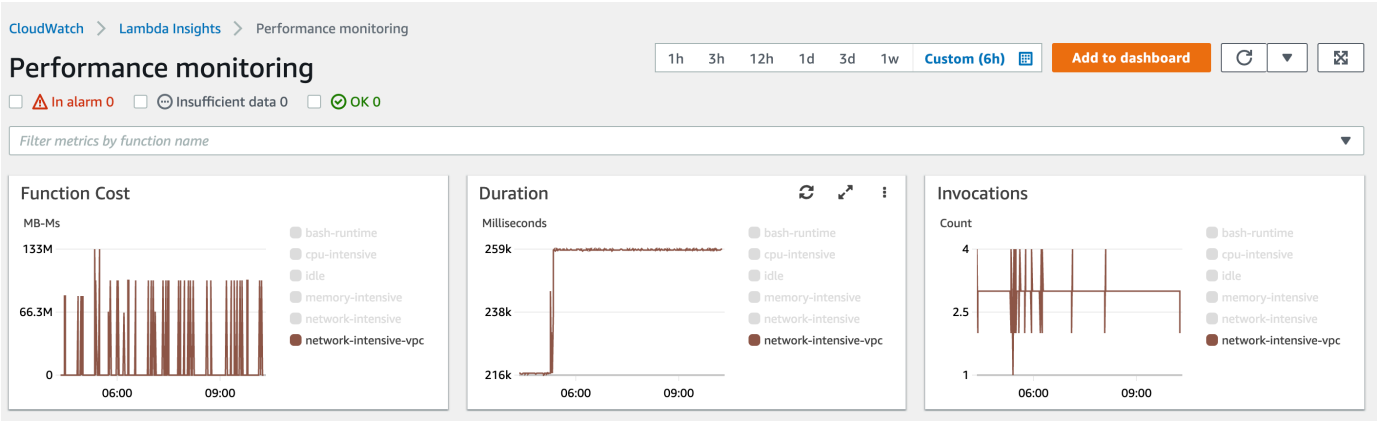
詳細については、CloudWatch の異常検出の使用方法を参照してください。

クエリを使用して関数のトラブルシューティングを行うワークフローの例

Lambda Insights ダッシュボードの single-function view を使用して、関数の所要時間が急増する根本原因を特定できます。例えば、multi-function overview で関数の所要時間が大きく増加している場合は、[Duration] ペインで一時停止するか、各関数を選択して、どの関数が増加の原因になっているかを判断できます。次に、single-function view に移動し、[Application logs] を確認して、根本原因を特定できます。

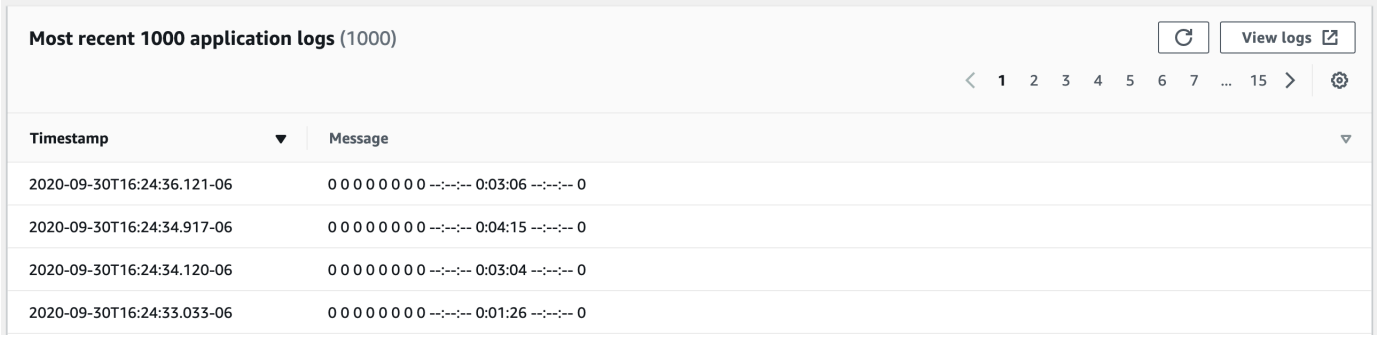
関数に対してクエリを実行するには

1. CloudWatch コンソールで [Multi-function (複数の関数)] ページを開きます。
2. [Duration] ペインで、所要時間メトリクスをフィルタリングする関数を選択します。

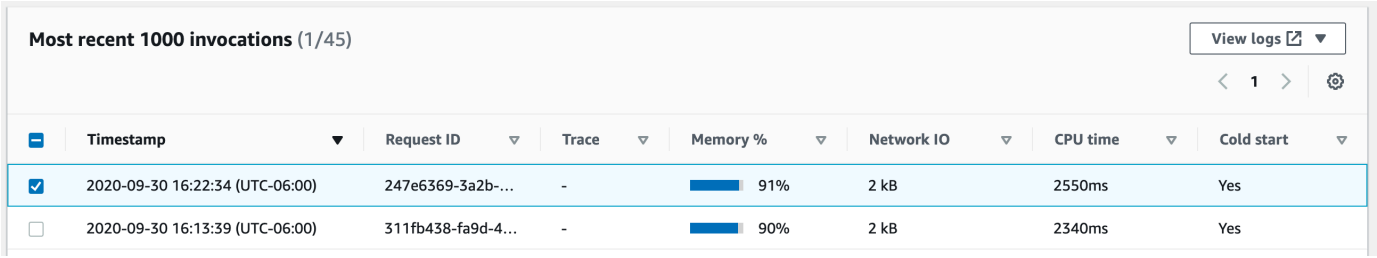


3. [Single-function (単独の関数)] ページを開きます。
4. [Filter metrics by function name] ドロップダウンリストを選択し、関数を選択します。
5. [Most recent 1000 application logs] を表示するには、[Application logs] タブを選択します。

6. [Timestamp] と [Message] を確認し、トラブルシューティングを行う呼び出しリクエストを特定します。



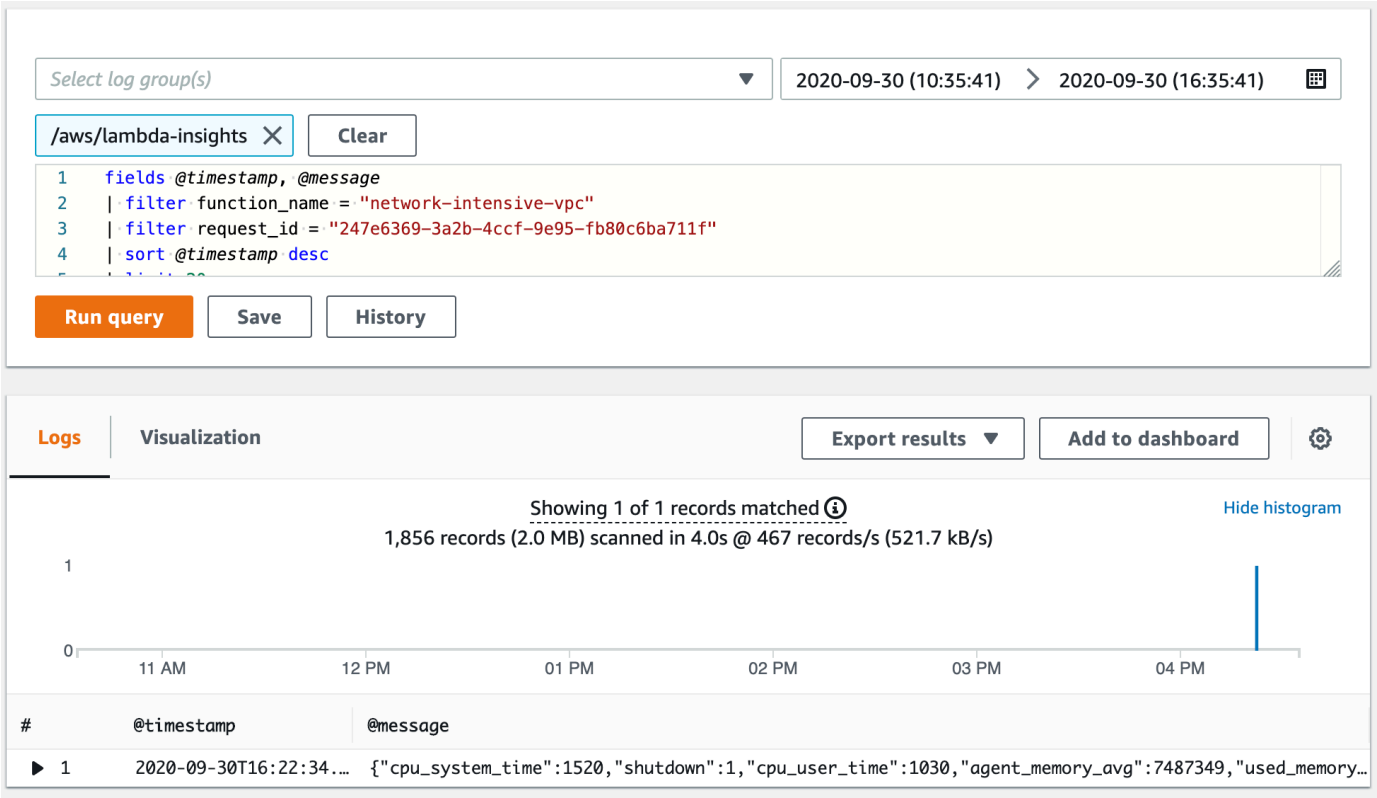
7. 最新の 1000 の呼び出し] を表示するには、[Invocations (呼び出し)] タブを選択します。
8. トラブルシューティングを行う呼び出しリクエストの [Timestamp] または [Message] を選択します。



9. [View logs] ドロップダウンリストを選択し、[View performance logs] を選択します。

   関数の自動生成されたクエリが [Logs Insights] ダッシュボードで開きます。

10. [Run query] を選択して、呼び出しリクエストの [Logs] メッセージを生成します。



次のステップ

- CloudWatch Logs ダッシュボードの作成方法については、Amazon CloudWatch ユーザーガイドの Create a Dashboard を参照してください。
- CloudWatch Logs ダッシュボードにクエリを追加する方法については、Amazon CloudWatch ユーザーガイドの Add Query to Dashboard or Export Query Results を参照してください。

Lambda 関数での CodeGuru Profiler の使用

Amazon CodeGuru Profiler を使用して、Lambda 関数のランタイムパフォーマンスに関するインサイトを得ることができます。このページでは、Lambda コンソールから CodeGuru Profiler を有効化する方法について説明します。

セクション

ランタイムのサポート

関数のランタイムが Python3.8、Python3.9、Amazon Linux 2 での Java 8、Java 11、または Java 17 の場合は、Lambda コンソールから CodeGuru Profiler を有効化できます。追加のランタイムバージョンについては、 CodeGuru Profiler を手動でアクティブ化できます。

- Java ランタイムについては、「AWS Lambda で実行される Java アプリケーションをプロファイリングする」を参照してください。
- Python ランタイムについては、「AWS Lambda で実行される Python アプリケーションをプロファイリングする」を参照してください。

> **Note**
>
> CodeGuru Profiler は現在、x86_64 アーキテクチャを使用する関数のみをサポートしています。

Lambda コンソールからの CodeGuru Profiler のアクティブ化

このセクションでは、Lambda コンソールから CodeGuru Profiler を有効化する方法について説明します。

Lambda コンソールから CodeGuru Profiler を有効化するには

1. Lambda コンソールの [関数ページ] を開きます。

2. 関数を選択します。
3. [設定] タブを選択します。
4. [Monitoring and operations tools] (モニタリングおよびオペレーションツール) ペインで、[編集] を選択します。
5. Amazon CodeGuru Profiler で、コードプロファイリング を有効にします。
6. [保存] を選択します。

アクティベーション後、 は という名前のプロファイラーグループ CodeGuru を自動的に作成しますaws-lambda-<your-function-name>。 CodeGuru コンソールから名前を変更できます。

Lambda コンソールから CodeGuru Profiler をアクティブ化するとどうなりますか？

コンソールから CodeGuru Profiler を有効にすると、Lambda はユーザーに代わって自動的に次の処理を行います。

- Lambda は CodeGuru 、Profiler レイヤーを関数に追加します。詳細については、「Amazon Profiler ユーザーガイド」のAWS Lambda「レイヤーの使用」を参照してください。 CodeGuru
- Lambda は、環境変数を関数に追加します。正確な値は、ランタイムによって異なります。

  環境変数

ランタイム	キー	値
java8.al2、java11	JAVA_TOOL_OPTIONS	-javaagent:/opt/codeguru-profiler-java-agent-standalone.jar
python3.8、python3.9	AWS_LAMBDA_EXEC_WRAPPER	/opt/codeguru_profiler_lambda_exec

- Lambda は、AmazonCodeGuruProfilerAgentAccess ポリシーを関数の実行ロールに追加します。

> **Note**
>
> コンソールから CodeGuru Profiler を非アクティブ化すると、Lambda は関数から CodeGuru Profiler レイヤーを自動的に削除します。ただし、Lambda は環境変数または `AmazonCodeGuruProfilerAgentAccess` ポリシーを実行ロールから削除しません。

次のステップ

- プロファイラーグループによって収集されたデータの詳細については、「Amazon CodeGuru Profiler ユーザーガイド」の「視覚化の使用」を参照してください。

他の AWS サービスを使用するワークフローの例

AWS Lambda は他の AWS のサービスと統合させることで、Lambda 関数のモニタリング、トレース、デバッグ、トラブルシューティングに使用できます。このページでは、AWS X-Ray、AWS Trusted Advisor で使用できる、Lambda 関数をトレースおよびトラブルシューティングするワークフローについて説明します。

セクション

前提条件

次のセクションでは、AWS X-Ray と Trusted Advisor を使って Lambda 関数をトラブルシューティングする手順について説明します。

AWS X-Ray を使用する

このページの AWS X-Ray ワークフローを完成させるには、AWS X-Ray が Lambda コンソールで有効になっている必要があります。実行ロールに必要なアクセス権限がない場合、Lambda コンソールは、そのアクセス権限の当該実行ロールへの追加を試みます。

Lambda コンソールで AWS X-Ray を有効にするには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. [設定] タブを選択します。
4. [Monitoring tools] ペインで、[Edit] を選択します。
5. AWS X-Ray で、[Active tracing] (アクティブトレース) をオンにします。
6. [Save] を選択します。

AWS Trusted Advisor を使用する

AWS Trusted Advisor は、AWS 環境を検査し、コストを節約し、システムの可用性とパフォーマンスを改善し、セキュリティギャップを埋めるのに役立つ方法についての推奨事項を提示します。Trusted Advisor チェックを使用して、AWS アカウントの Lambda 関数とアプリケーションを評価します。チェックは、実行する推奨手順と詳細情報のリソースを提供します。

- AWS チェックのための Trusted Advisor サポートプランの詳細については、「サポートプラン」をご参照ください。
- Lambda のチェックの詳細については、AWS Trusted Advisor のベストプラクティスのチェックリストを参照してください。
- Trusted Advisor コンソールの使用方法の詳細については、「AWS Trusted Advisor の開始方法」をご参照ください。
- コンソールアクセスを Trusted Advisor に許可および拒否する方法については、「IAM ポリシーの例」をご参照ください。

料金

- 記録、取得、およびスキャンされたトレースの数に基づいて、AWS X-Ray を使用した分のみ料金が発生します。詳細については、AWS X-Ray 料金を参照してください。
- Trusted Advisor Business および Enterprise サポートサブスクリプションには、AWS コスト最適化チェックが含まれています。詳細については、AWS Trusted Advisor 料金を参照してください。

トレースマップを表示する AWS X-Ray ワークフローの例

を有効にしている場合はAWS X-Ray、 CloudWatch コンソールでトレースマップを表示できます。トレースマップには、サービスエンドポイントとリソースがノードとして表示され、各ノードとその接続のトラフィック、レイテンシー、およびエラーがハイライトされます。

ノードを選択すると、サービスのその部分に関連付けられた相関メトリクス、ログ、およびトレースに関する詳細なインサイトを表示できます。これにより、問題とそのアプリケーションへの影響を調査できます。

CloudWatch コンソールを使用してトレースマップとトレースを表示するには

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。

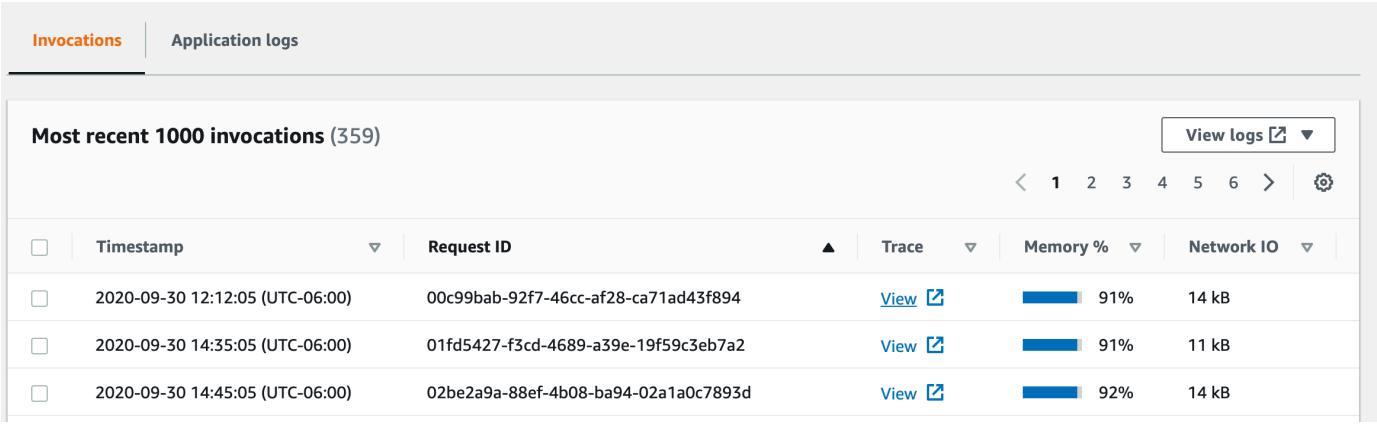
3. [モニタリング] を選択します。
4. [X-Ray トレースを表示] を選択します。
5. 左側のナビゲーションペインの [X-Ray トレース] で [トレースマップ] を選択します。
6. 定義済みの時間範囲から選択するか、カスタムの時間範囲を選択します。
7. リクエストのトラブルシューティングを行うには、フィルターを選択します。

トレースの詳細を表示する AWS X-Ray ワークフローの例

を有効にしている場合はAWS X-Ray、 CloudWatch Lambda Insights ダッシュボードの single-function view を使用して、関数呼び出しエラーの分散トレースデータを表示できます。例えば、アプリケーションログメッセージにエラーが表示された場合は、トレースマップを開くと、分散トレースデータとトランザクションを処理するその他のサービスを表示できます。

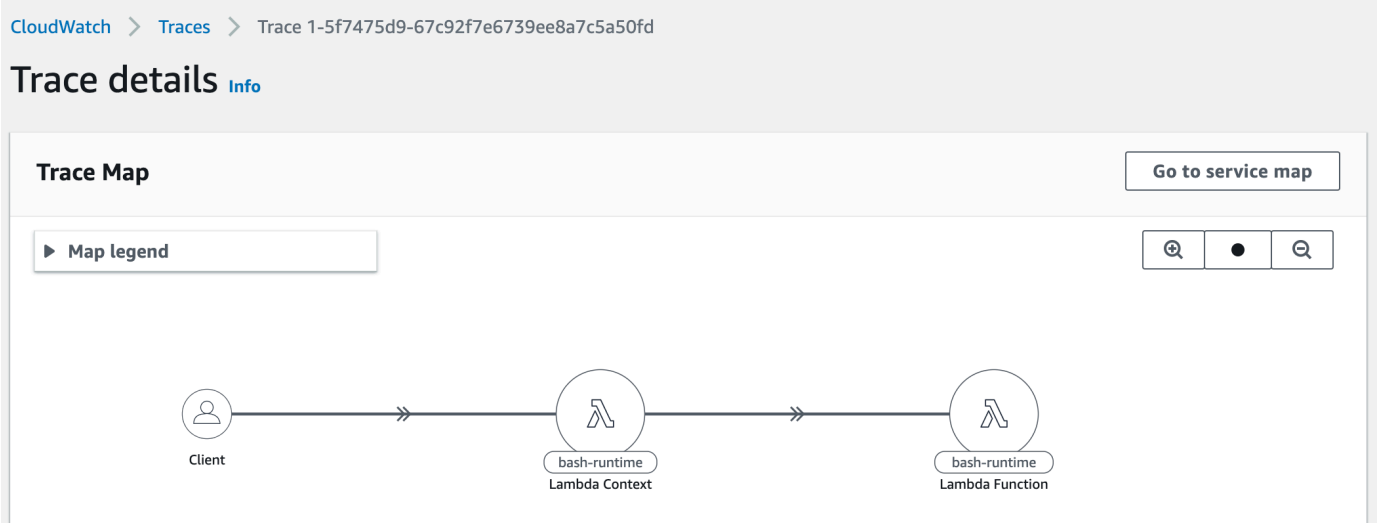
関数のトレース詳細を表示するには

1. CloudWatch コンソールで単一関数ビューを開きます。
2. [Application logs] (アプリケーションログ) タブを選択します。
3. [Timestamp] (タイムスタンプ) または [Message] (メッセージ) を使用し、トラブルシューティングを行う呼び出しリクエストを特定します。
4. [Most recent 1000 invocations] を表示するには、[Invocations] タブを選択します。



5. [Request ID] (リクエスト ID) 列を選択して、エントリをアルファベットの昇順でソートします。
6. [Trace] (トレース) 列で、[View] (表示) を選択します。

   トレースマップビューで、[トレースの詳細] ページが開きます。



推奨事項を表示するための AWS Trusted Advisor ワークフローの例

Trusted Advisor はすべての AWS リージョンの Lambda 関数をチェックして、潜在的に最も多くコストを節約できる関数を特定し、最適化のための実用的な推奨事項を提供します。関数の実行時間、請求期間、使用済みメモリ、設定済みメモリ、タイムアウト設定、エラーなどの Lambda の使用状況データを分析します。

例えば、エラー率が高い Lambda 関数チェックでは、 AWS X-Rayまたは を使用して Lambda 関数のエラー CloudWatch を検出することを推奨しています。

エラー率の高い関数をチェックするには

1. Trusted Advisor コンソールを開きます。
2. [Cost Optimization] (コスト最適化) カテゴリを選択します。
3. [AWS Lambda Functions with High Error Rates] (エラー率の高い AWS Lambda 関数) まで下にスクロールします。セクションを展開すると、結果と推奨されるアクションが表示されます。

次のステップ

- トレース、メトリクス、ログ、アラームを統合する方法の詳細については、「Using the X-Ray trace map」をご参照ください。
- 「ウェブサービスとして Trusted Advisor を使用する」で Trusted Advisor チェックのリストを取得する方法の詳細をご覧ください。

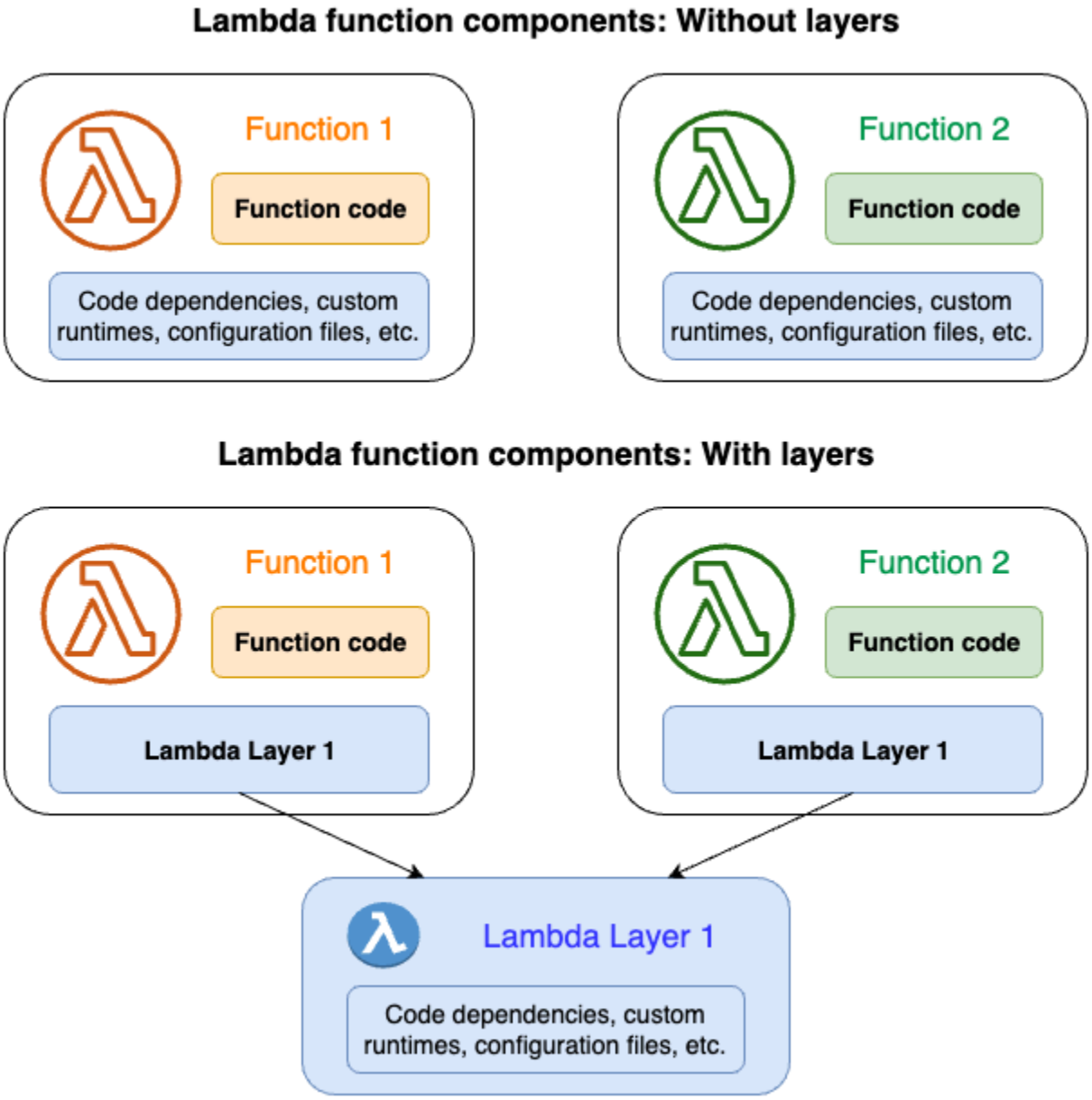
レイヤーによる Lambda 依存関係の管理

Lambda レイヤーは、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。

レイヤーの使用を検討する理由は複数あります。

- デプロイパッケージのサイズを小さくするため。関数コードとともにすべての関数依存関係をデプロイパッケージに含める代わりに、レイヤーに配置します。これにより、デプロイパッケージは小さく整理された状態に保たれます。
- コア関数ロジックを依存関係から分離するため。レイヤーを使用すると、関数コードと独立して関数の依存関係を更新でき、その逆も可能となります。これにより、関心事の分離が促進され、関数ロジックに集中することができます。
- 複数の関数間で依存関係を共有するため。レイヤーを作成したら、それをアカウント内の任意の数の関数に適用できます。レイヤーがない場合、個々のデプロイパッケージに同じ依存関係を含める必要があります。
- Lambda コンソールのコードエディターを使用するため。コードエディターは、関数コードの軽微な更新をすばやくテストするのに便利なツールです。ただし、デプロイパッケージのサイズが大きすぎる場合は、エディターを使用できません。レイヤーを使用すると、パッケージのサイズが小さくなり、コードエディターの使用制限を解除できます。

次の図は、依存関係を共有する 2 つの関数間の大まかなのアーキテクチャの違いを示しています。一方は Lambda レイヤーを使用し、もう一方は使用しません。



関数にレイヤーを追加すると、Lambda は関数の実行環境の /opt ディレクトリにレイヤーのコンテンツを抽出します。ネイティブにサポートされているすべての Lambda ランタイムには、/opt ディレクトリ内の特定のディレクトリへのパスが含まれています。これにより、関数はレイヤーコンテンツにアクセスできるようになります。これらの特定のパスの詳細とレイヤーを適切にパッケージ化する方法については、the section called “レイヤーのパッケージング” を参照してください。

各関数につき最大 5 つのレイヤーを含めることができます。また、レイヤーは、.zip ファイルアーカイブとしてデプロイされた Lambda 関数でのみ使用できます。コンテナイメージとして定義された関数では、コンテナイメージの作成時に、優先ランタイムとすべてのコード依存関係をパッケージ

化します。詳細については、AWS コンピューティングブログの「コンテナイメージの Lambda レイヤーと拡張機能を使用する」を参照してください。

トピック

- レイヤーの使用方法
- レイヤーとレイヤーバージョン
- レイヤーコンテンツのパッケージング
- Lambda でのレイヤーの作成と削除
- 関数へのレイヤーの追加
- AWS CloudFormation を使用したレイヤー
- AWS SAM を使用したレイヤー

レイヤーの使用方法

レイヤーを作成するには、通常のデプロイパッケージの作成と同様に、依存関係を .zip ファイルにパッケージ化します。具体的には、レイヤーの作成と使用の一般的なプロセスには、次の 3 つのステップが含まれます。

- まず、レイヤーコンテンツをパッケージ化します。これは .zip ファイルアーカイブを作成することを意味します。詳細については、「the section called “レイヤーのパッケージング”」を参照してください。
- 次に、Lambda でレイヤーを作成します。詳細については、「the section called “レイヤーの作成と削除”」を参照してください。
- レイヤーを関数に追加します。詳細については、「the section called “レイヤーの追加”」を参照してください。

レイヤーとレイヤーバージョン

レイヤーのバージョンは、レイヤーの特定のバージョンの不変のスナップショットです。新しいレイヤーを作成すると、Lambda はバージョン番号 1 の新しいレイヤーバージョンを作成します。レイヤーの更新を発行するたびに、Lambda はバージョン番号を増やし、新しいレイヤーバージョンを作成します。

すべてのレイヤーバージョンは、一意の Amazon リソースネーム (ARN) により識別されます。関数にレイヤーを追加する場合、使用したいレイヤーバージョンを正確に指定する必要があります。

レイヤーコンテンツのパッケージング

Lambda レイヤーは、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。

このセクションでは、レイヤーコンテンツを適切にパッケージングする方法について説明します。レイヤーの概念的な情報とその使用を検討する理由の詳細については、Lambda レイヤー を参照してください。

レイヤーを作成する最初のステップは、すべてのレイヤーコンテンツを .zip ファイルアーカイブにバンドルすることです。Lambda 関数は Amazon Linux 上で実行されるため、レイヤーコンテンツは Linux 環境でコンパイルおよびビルドできる必要があります。

レイヤーコンテンツが Linux 環境で適切に動作することを確認するには、「Docker」または「AWS Cloud9」のようなツールを使用してレイヤーコンテンツを作成することをお勧めします。AWS Cloud9 は、コードを実行およびテストするために Linux サーバーへのビルトインアクセスを提供するクラウドベースの統合開発環境 (IDE) です。詳細については、AWS Compute Blog の「Lambda レイヤーを使用して開発プロセスを簡素化する」を参照してください。

トピック

- 各 Lambda ランタイムのレイヤーパス

各 Lambda ランタイムのレイヤーパス

関数にレイヤーを追加すると、Lambda はレイヤーのコンテンツをその実行環境の `/opt` ディレクトリに読み込みます。Lambda ランタイムごとに、PATH 変数には `/opt` ディレクトリ内の特定のフォルダパスがあらかじめ含まれます。PATH 変数がレイヤーコンテンツを取得できるようにするには、レイヤーの .zip ファイルの依存関係が次のフォルダーパスにある必要があります。

各 Lambda ランタイムのレイヤーパス

実行時間	パス
Node.js	`nodejs/node_modules`
	`nodejs/node14/node_modules` (NODE_PATH)
	`nodejs/node16/node_modules` (NODE_PATH)

実行時間	パス
	nodejs/node18/node_modules (NODE_PATH)
Python	python
	python/lib/ *python3.x* /site-packages (サイトディレクトリ)
Java	java/lib (CLASSPATH)
Ruby	ruby/gems/3.2.0 (GEM_PATH)
	ruby/lib (RUBYLIB)
すべてのランタイム	bin (PATH)
	lib (LD_LIBRARY_PATH)

次の例は、レイヤーの .zip アーカイブでフォルダを設定する方法を示しています。

Node.js

Example Node.js 用の AWS X-Ray SDK のファイル構造

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example Requests ライブラリのファイル構造

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

Ruby

Example JSON gem のファイル構造

```
json.zip
# ruby/gems/2.7.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example Jackson JAR ファイルのファイル構造

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example JQ ライブラリのファイル構造

```
jq.zip
# bin/jq
```

レイヤーのパッケージ化、作成、追加に関する言語固有の手順については、次のページを参照してください。

- Python – the section called “レイヤー”
- Java – the section called “レイヤー”

Lambda でのレイヤーの作成と削除

Lambda レイヤーは、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。

このセクションでは、Lambda でレイヤーを作成および削除する方法を説明します。レイヤーの概念的な情報とその使用を検討する理由の詳細については、Lambda レイヤー を参照してください。

あなたがレイヤーコンテンツをパッケージ化したら、次のステップは Lambda でのレイヤーの作成です。このセクションでは、Lambda コンソールまたは Lambda API のみを使用して、レイヤーを作成および削除する方法を示します。AWS CloudFormation を使用してレイヤーを作成するには、the section called “AWS CloudFormation を使用したレイヤー” を参照してください。AWS Serverless Application Model (AWS SAM) を使用してレイヤーを作成するには、the section called “AWS SAM を使用したレイヤー” を参照してください。

トピック

- レイヤー作成
- レイヤーバージョンの削除

レイヤー作成

レイヤーを作成するには、ローカルマシンから、または Amazon Simple Storage Service (Amazon S3) から .zip ファイルのアーカイブをアップロードします。Lambda は、関数の実行環境を設定する際に、レイヤーの内容を `/opt` ディレクトリに抽出します。

レイヤーには、1 つまたは複数のレイヤーバージョンを含めることができます。レイヤーを作成すると、Lambda はレイヤーのバージョンをバージョン 1 に設定します。既存のレイヤーバージョンの権限はいつでも変更することができます。ただし、コードを更新したり、その他の構成を変更したりするには、新しいバージョンのレイヤーを作成する必要があります。

レイヤーを作成するには (コンソール)

1. Lambda コンソールの [Layers (レイヤー)] ページを開きます。
2. [Create layer] (レイヤーの作成) を選択します。
3. [レイヤー設定] の [名前] に、レイヤーの名前を入力します。
4. (オプション) [Description] (説明) で、レイヤーの説明を入力します。
5. レイヤーコードをアップロードするには、次のいずれかを実行します。

- コンピューターから .zip ファイルをアップロードするには、[.zip ファイルをアップロード] を選択します。[Upload] (アップロード) を選択して、ローカルの .zip ファイルを選択します。
- Amazon S3 からファイルをアップロードするには、[Upload a file from Amazon S3] (Amazon S3 からファイルをアップロードする) を選択します。その後、[Amazon S3 link URL] (Amazon S3 のリンク URL) で、ファイルへのリンクを入力します。

6. (オプション) [互換性のあるアーキテクチャ] では、1 つまたは両方の値を選択します。詳細については、「the section called “命令セット (ARM/x86)”」を参照してください。
7. (オプション) [互換性のあるランタイム] では、お使いのレイヤーと互換性のあるランタイムを選択します。
8. (オプション) [License] (ライセンス) で、必要なライセンス情報を入力します。
9. [作成] を選択します。

または、 PublishLayerVersion API を使用してレイヤーを作成することもできます。たとえば、名前、説明、および .zip ファイルアーカイブを指定して、`publish-layer-version` AWS Command Line Interface (CLI) コマンドを使用できます。ライセンス情報、互換性のあるランタイム、および互換性のあるアーキテクチャパラメータはオプションです。

```
aws lambda publish-layer-version --layer-name my-layer \
    --description "My layer" \
    --license-info "MIT" \
    --zip-file fileb://layer.zip \
    --compatible-runtimes python3.10 python3.11 \
    --compatible-architectures "arm64" "x86_64"
```

次のような出力が表示されます:

```
{
    "Content": {
        "Location": "https://awslambda-us-east-2-layers.s3.us-east-2.amazonaws.com/
snapshots/123456789012/my-layer-4aaa2fbb-ff77-4b0a-ad92-5b78a716a96a?
versionId=27iWyA73cCAYqyH...",
        "CodeSha256": "tv9jJO+rPbXUUXuRKi7CwHzKtLDkDRJLB3cC3Z/ouXo=",
        "CodeSize": 169
    },
    "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
    "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer:1",
    "Description": "My layer",
```

```
    "CreatedDate": "2023-11-14T23:03:52.894+0000",
    "Version": 1,
    "CompatibleArchitectures": [
        "arm64",
        "x86_64"
     ],
    "LicenseInfo": "MIT",
    "CompatibleRuntimes": [
        "python3.10",
        "python3.11"
    ]
}
```

publish-layer-version を呼び出すたびに、新しいバージョンのレイヤーを作成します。

レイヤーバージョンの削除

レイヤーバージョンを削除するには、 DeleteLayerVersion API を使用します。たとえば、レイヤー名とレイヤーバージョンを指定して delete-layer-version CLI コマンドを使用することができます。

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

レイヤーバージョンを削除すると、そのバージョンを使用する Lambda 関数を設定できなくなります。ただし、該当バージョンをすでに使用している関数は引き続きそのバージョンにアクセスできます。また、Lambda はレイヤー名にバージョン番号を再利用することはありません。

関数へのレイヤーの追加

Lambda レイヤーは、補助的なコードやデータを含む .zip ファイルアーカイブです。レイヤーには通常、ライブラリの依存関係、カスタムランタイム、または設定ファイルが含まれています。

このセクションでは、Lambda 関数にレイヤーを追加する方法を説明します。レイヤーの概念的な情報とその使用を検討する理由の詳細については、Lambda レイヤー を参照してください。

レイヤーを使用するよう Lambda 関数を設定する前に、以下を実行する必要があります。

- レイヤーコンテンツのパッケージ化
- Lambda でレイヤーを作成
- レイヤーバージョンで GetLayerVersion API を呼び出すアクセス許可があることを確認します。AWS アカウント の関数については、ユーザーポリシーでこの権限を持っている必要があります。別のアカウントでレイヤーを使用するには、そのアカウントの所有者は、リソースベースのポリシーで自分のアカウントにアクセス許可を付与する必要があります。例については、「the section called “他のアカウントへのアクセス権をレイヤーに付与する”」を参照してください。

Lambda 関数には最大 5 つのレイヤーを追加できます。関数とすべてのレイヤーの解凍後の合計サイズは、解凍後のデプロイパッケージのサイズクォータである 250 MB を超えることはできません。詳細については、「Lambda クォータ」を参照してください。

関数は、すでに追加したレイヤーバージョンであれば、そのレイヤーバージョンが削除された後でも、またはレイヤーへのアクセス権限が取り消された後でも、そのレイヤーバージョンを引き続き使用できます。しかし、削除されたレイヤーバージョンを使用して新しい関数を作成することはできません。

> **Note**
>
> 関数に追加するレイヤーが、関数のランタイムおよび命令セットアーキテクチャと互換性があることを確認します。

関数へレイヤーを追加するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 設定する関数を選択します。

3. [Layers] (レイヤー) で [Add a layer] (レイヤーの追加) をクリックします。
4. [レイヤーを選択する] で、レイヤーソースを選択します。

   a. [AWS レイヤー] または [カスタムレイヤー] のレイヤーソースの場合、プルダウンメニューからレイヤーを選択します。[Version (バージョン)]のプルダウンメニューからレイヤーバージョンを選択します。

   b. [ARN を指定する] レイヤーソースの場合、テキストボックスに ARN を入力して [確認] を選択します。次に、[Add] (追加) を選択します。

レイヤーを追加する順序は、Lambda がレイヤーのコンテンツを実行環境にマージする順序です。レイヤーのマージ順序はコンソールを使用して変更できます。

関数のレイヤーのマージ順序を更新するには (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 設定する関数を選択します。
3. [Layers] (レイヤー) で [Edit] (編集) をクリックします。
4. レイヤーの 1 つを選択します。
5. [Merge earlier (前にマージする)] または [Merge later (後にマージする)] を選択して、レイヤーの順序を調整します。
6. [保存] をクリックします。

レイヤーはバージョン管理されています。各レイヤーバージョンのコンテンツは変更できません。レイヤーの所有者は更新されたコンテンツを提供するため、新しいレイヤーバージョンをリリースすることができます。関数に添付されているレイヤーバージョンを更新するには、コンソールを使用します。

関数のレイヤーバージョンを更新するには (コンソール)

1. Lambda コンソールの [Layers (レイヤー)] ページを開きます。
2. バージョンを更新したいレイヤーを選択します。
3. [Functions using this version] (このバージョンを使用する関数) タブを選択します。
4. 変更したい関数を選択してから、[Edit] (編集) を選択します。
5. [Layer version] (レイヤーバージョン) では、変更するレイヤーのバージョンを選択します。
6. [Update functions] (関数を更新) をクリックします。

AWS アカウント間で、関数のレイヤーバージョン更新することはできません。

トピック

- 関数からレイヤーコンテンツにアクセスする
- レイヤー情報の確認

関数からレイヤーコンテンツにアクセスする

Lambda 関数にレイヤーを含めると、Lambda は関数実行環境で/optディレクトリにレイヤーコンテンツを抽出します。Lambda は、関数によって一覧表示された順序 (低から高) でレイヤーを抽出します。Lambda は同じ名前のフォルダをマージします。複数のレイヤーに同じファイルが表示された場合は、関数は最後に抽出されたレイヤーのバージョンを使用します。

各 Lambda ランタイムは、PATH 変数に特定の /opt ディレクトリフォルダを追加します。関数コードはパスを指定しなくても、レイヤーコンテンツにアクセスできます。Lambda 実行環境のパス設定の詳細については、「the section called “定義されたランタイム環境変数”」を参照してください。

レイヤー作成時にライブラリをどこに含めるかに関しては、the section called “各 Lambda ランタイムのレイヤーパス” を参照してください。

Node.js または Python ランタイムを使用している場合は、Lambda コンソールの組み込みコードエディターを使用することができます。レイヤーとして追加した任意のライブラリを、現在の関数にインポートできるはずです。

レイヤー情報の確認

関数のランタイムと互換性のあるレイヤーをアカウント内で検索するには、 ListLayers API を使用します。たとえば、次の `list-layers` AWS Command Line Interface (CLI) コマンドを使用することができます。

```
aws lambda list-layers --compatible-runtime python3.9
```

次のような出力が表示されます:

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
```

```
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2023-11-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.9",
                    "python3.10",
                    "python3.11",
                ]
            }
        }
    ]
}
```

アカウント内のすべてのレイヤーをリスト化するには、--compatible-runtime オプションを省略します。レスポンスの詳細には、各レイヤーの最新バージョンが表示されます。

ListLayerVersions API を使用してレイヤーの最新バージョンを取得することもできます。たとえば、次の list-layer-versions CLI コマンドを使用することができます。

```
aws lambda list-layer-versions --layer-name my-layer
```

次のような出力が表示されます:

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
```

```
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

AWS CloudFormation を使用したレイヤー

AWS CloudFormation を使用してレイヤーを作成し、そのレイヤーを Lambda 関数に関連付けることができます。次のテンプレートの例では、my-lambda-layer という名前のレイヤーを作成し、そのレイヤーを Layers プロパティを使用して Lambda 関数にアタッチします。

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: DOC-EXAMPLE-BUCKET
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Layers:
        - !Ref MyLambdaLayer
```

AWS SAM を使用したレイヤー

AWS Serverless Application Model (AWS SAM) を使用して、アプリケーションでレイヤーの作成を自動化できます。AWS::Serverless::LayerVersion リソースタイプによって、Lambda 関数設定から参照できるレイヤーバージョンが作成されます。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://DOC-EXAMPLE-BUCKET/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://DOC-EXAMPLE-BUCKET/my-function
      Layers:
        - !Ref MyLambdaLayer
```

Lambda 拡張機能を使用して Lambda 関数を補強する

Lambda 拡張機能を使用して Lambda 関数を補強できます。例えば、Lambda 拡張機能を使用して、任意のモニタリングツール、オブザーバビリティツール、セキュリティツール、およびガバナンスツールに関数を統合できます。AWS Lambda パートナーが提供する幅広いツールセットから選択することも、独自の Lambda 拡張機能を作成することもできます。

Lambda は、外部拡張機能と内部拡張機能をサポートしています。外部拡張機能は、実行環境で独立したプロセスとして実行され、関数の呼び出しが完全に処理された後も引き続き実行されます。拡張機能は別々のプロセスとして実行されるため、関数とは異なる言語で記述できます。すべての Lambda ランタイム は拡張機能をサポートします。

内部拡張機能は、ランタイムプロセスの一部として実行されます。関数は、ラッパースクリプトまたは `JAVA_TOOL_OPTIONS` などのインプロセスメカニズムを使用して、内部拡張機能にアクセスします。詳細については、「ランタイム環境の変更」を参照してください。

Lambda コンソール、AWS Command Line Interface (AWS CLI)、または Infrastructure as Code (IAC) サービス、およびツール (AWS CloudFormation、AWS Serverless Application Model (AWS SAM)、Terraform など) を使用して、関数に拡張機能を追加できます。

料金は、拡張機能の実行時間 (1 ms 単位) に対して課金されます。独自の拡張機能をインストールする場合、料金は発生しません。拡張機能の料金の詳細については、AWS Lambda 料金表を参照してください。パートナー拡張機能の料金については、パートナーのウェブサイトを参照してください。公式パートナーの拡張機能のリストについては、「the section called “拡張機能パートナー”」を参照してください。

拡張機能と Lambda 関数でそれらを使用する方法に関するチュートリアルについては、「AWS Lambda Extensions Workshop」を参照してください。

トピック

- 実行環境
- パフォーマンスとリソースへの影響
- アクセス許可
- Lambda 拡張機能の設定
- AWS Lambda 拡張機能パートナー
- Lambda 拡張機能 API を使用した拡張機能の作成

- Lambda Telemetry API

実行環境

Lambda は、実行環境で関数を呼び出します。これにより、安全で分離されたランタイム環境が提供されます。実行環境は、関数の実行に必要なリソースを管理し、関数のランタイムおよび拡張機能のライフサイクルサポートを提供します。

実行環境のライフサイクルには、以下のフェーズが含まれています。

- **Init**: このフェーズ中、Lambda は、設定したリソースを使用して実行環境を作成またはフリーズ解除し、関数コードとすべてのレイヤーをダウンロードして、拡張機能とランタイムを初期化し、関数の初期化コード (メインハンドラーの外部にあるコード) を実行します。Init フェーズは、最初の呼び出し中か、プロビジョニング済みの同時実行を有効にしている場合は関数呼び出しの前に、発生します。

  Init フェーズは、Extension init、Runtime init、Function init の 3 つのサブフェーズに分割されます。これらのサブフェーズでは、関数コードが実行される前に、すべての拡張機能とランタイムがそのセットアップタスクを完了します。

  Lambda SnapStart がアクティブ化されると、関数バージョンの発行時に Init フェーズが開始されます。Lambda は、初期化された実行環境のメモリとディスク状態のスナップショットを保存し、暗号化されたスナップショットを永続化して、低レイテンシーアクセスのためにスナップショットをキャッシュします。beforeCheckpoint ランタイムフックがある場合、コードは Init フェーズの最後に実行されます。
- **Restore** (SnapStart のみ): SnapStart 関数を初めて呼び出し、その関数がスケールアップすると、Lambda は関数をゼロから初期化するのではなく、永続化されたスナップショットから新しい実行環境を再開します。afterRestore() ランタイムフックがある場合、コードは Restore フェーズの最後に実行されます。ユーザーには、afterRestore() ランタイムフックの所要時間分の料金が請求されます。タイムアウト制限 (10 秒) 内にランタイム (JVM) がロードされ、afterRestore() ランタイムフックが完了される必要があります。その時間を超えると、SnapStartTimeoutException が発生します。Restore フェーズが完了すると、Lambda が関数ハンドラーを呼び出します (呼び出しフェーズ)。
- **Invoke**: このフェーズでは、Lambda は関数ハンドラーを呼び出します。関数が実行され、完了すると、Lambda は、別の関数呼び出しを処理する準備をします。
- **Shutdown**: このフェーズは、Lambda 関数が一定期間呼び出しを受け取らなかった場合にトリガーされます。Shutdown フェーズでは、Lambda はランタイムをシャットダウンし、拡張機能

にアラートを出してそれらを完全に停止させた後、環境を削除します。Lambda は、各拡張機能に Shutdown イベントを送信します。これにより、環境がシャットダウンされようとしていることを各拡張機能に伝えます。

Init フェーズでは、Lambda は拡張機能を含むレイヤーを実行環境の /opt ディレクトリに抽出します。Lambda は、/opt/extensions/ ディレクトリで拡張機能を検索し、各ファイルを拡張機能を起動するための実行可能なブートストラップであると解釈し、すべての拡張機能を並行して開始します。

パフォーマンスとリソースへの影響

関数の拡張機能のサイズは、デプロイパッケージのサイズ制限に対してカウントされます。アーカイブの Zip ファイルについては、関数とすべての拡張機能を解凍した後の合計サイズは、解凍後のデプロイパッケージのサイズ制限 250 MB を超えることはできません。

拡張機能は、CPU、メモリ、ストレージなどの関数リソースを共有するため、関数のパフォーマンスに影響を与える可能性があります。例えば、拡張機能で計算負荷の高いオペレーションを実行する場合、関数の実行時間が長くなることがあります。

Lambda が関数を呼び出す前に、各拡張機能の初期化を完了する必要があります。したがって、初期化にかなり時間がかかる拡張機能は、関数呼び出しのレイテンシーを増やす可能性があります。

関数の実行後に拡張機能によって発生する追加時間を測定するには、PostRuntimeExtensionsDuration 関数メトリクスを使用できます。使用されるメモリの増加を測定するには、MaxMemoryUsed メトリクスを使用できます。特定の拡張機能の影響を理解するために、異なるバージョンの関数を並行して実行できます。

アクセス許可

拡張機能は、関数と同じリソースにアクセスできます。拡張機能は関数と同じ環境内で実行されるため、アクセス許可は関数と拡張機能の間で共有されます。

アーカイブの Zip ファイルの場合、AWS CloudFormation テンプレートを作成することで、複数の関数に AWS Identity and Access Management (IAM) アクセス許可を含む、同じ拡張機能の設定をアタッチする作業を簡略化できます。

Lambda 拡張機能の設定

拡張子の設定 (.zip ファイルアーカイブ)

関数に、Lambda レイヤーとして拡張機能を追加できます。レイヤーを使用すれば、組織全体または Lambda デベロッパーのコミュニティ全体で拡張機能を共有できます。1 つ以上の拡張機能をレイヤーに追加できます。1 つの関数に最大 10 個の拡張機能を登録できます。

レイヤーの場合と同じメソッドを使用して、関数に拡張機能を追加します。詳細については、「Lambda レイヤー」を参照してください。

関数に拡張機能を追加する (コンソール)

1. Lambda コンソールの [関数ページ] を開きます。
2. 関数を選択します。
3. 選択されていない場合は、[Code (コード)] タブを選択します。
4. [レイヤー] で、[Edit (編集)] を選択します。
5. [Choose a layer] の [Specify an ARN] を選択します。
6. [Specify an ARN] に、拡張機能レイヤーの Amazon リソースネーム (ARN) を入力します。
7. [追加] を選択します。

コンテナイメージでの拡張機能の使用

コンテナイメージに拡張機能を追加できます 。ENTRYPOINT コンテナイメージ設定では、関数のメインプロセスを指定します。Dockerfile で ENTRYPOINT 設定を行うか、関数設定のオーバーライドとして設定します。

コンテナ内で複数のプロセスを実行できます。Lambda は、メインプロセスと任意の追加プロセスのライフサイクルを管理します。Lambda は、拡張機能 API を使用して、拡張機能のライフサイクルを管理します。

外部拡張機能の追加の例

外部拡張機能は、Lambda 関数とは別のプロセスで実行されます。Lambda は、`/opt/extensions/` ディレクトリで各拡張モジュールのプロセスを開始します。Lambda は、拡張機能 API を使用して、拡張機能のライフサイクルを管理します。関数が実行され完了すると、Lambda はそれぞれの外部拡張機能に `Shutdown` イベントを送信します。

Example Python ベースイメージに外部拡張機能を追加する

```
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt
ADD my-extension.zip /opt
CMD python ./my-function.py
```

次のステップ

拡張機能の詳細を参照するには、次のリソースをお勧めします。

- 基本的な使用例については、AWS Lambda コンピューティングブログの Building Extensions for AWS を参照してください。
- AWS Lambda パートナーが提供する拡張機能の詳細については、AWS Lambda コンピューティングブログの AWS 拡張機能の紹介を参照してください。
- 使用可能な拡張機能とラッパースクリプトの例を表示するには、AWSサンプル GitHub リポジトリのAWS Lambda「拡張機能」を参照してください。

AWS Lambda 拡張機能パートナー

AWS Lambda では、Lambda 関数と統合する拡張機能を、複数のサードパーティの企業から提供を受けています。次のリストに、現在利用が可能なサードパーティ拡張機能の詳細を示します。

- AppDynamics – Node.js または Python Lambda 関数の自動計測、および関数のパフォーマンスに関する可視性とアラートを提供します。
- Check Point CloudGuard – サーバーレスアプリケーションのライフサイクルを通じ完全なセキュリティを提供する、拡張ベースのランタイムソリューションです。
- Datadog – メトリクス、トレース、ログを使用して、サーバーレスアプリケーションのための、包括的かつリアルタイムの可視性を提供します。
- Dynatrace – トレースとメトリクスを可視化するとともに、AIを活用してアプリケーションスタック全体のエラーを自動検出し、その根本原因の分析を行います。
- Elastic — アプリケーションパフォーマンスモニタリング (APM) 機能を提供し、相関関係のあるトレース、メトリック、ログを使用して根本原因の問題を特定して解決します。
- Epsagon – 呼び出しイベントをリッスンしながらトレースを保存し、それらを同時に、実行される Lambda 関数に送信します。
- Fastly— インジェクション型攻撃、認証情報スタッフィングによるアカウント乗っ取り、悪意のあるボット、API の悪用などの疑わしいアクティビティから Lambda 関数を保護します。
- HashiCorp Vault – シークレットを管理し、デベロッパーが (関数の Vault を設定することなく) それらを関数コード内で使用できるようにします。
- Honeycomb – アプリケーションスタックをデバッグするための可観測性を提供するツールです。
- Lumigo – Lambda 関数の呼び出しのプロファイリングを行い、サーバーレス環境およびマイクロサービス環境における問題のトラブルシューティングのためのメトリクスを収集します。
- New Relic – Lambda 関数とともに実行することで、テレメトリを自動的に収集および拡張し、さらに New Relic の統合可観測性プラットフォームに転送します。
- 「Sedai」 — AI/ML を搭載した自律型クラウド管理プラットフォームで、クラウド運用チームに継続的に最適化を行い、クラウドのコスト削減、パフォーマンス、可用性を大規模に最大化します。
- Sentry – Lambda 関数のパフォーマンスを診断、修正、最適化します。
- Site24x7 – Lambda 環境でリアルタイムの可観測性を実現します。
- Splunk – 高分解能、低レイテンシーのメトリクスを収集して、Lambda 関数を効率的かつ効果的にモニタリングします。
- Sumo Logic – サーバーレスアプリケーションの動作状況とパフォーマンスを可視化します。
- Thundra – トレース、メトリクス、ログなど非同期のテレメトリレポートを提供します。

- Salt Security — さまざまなランタイムの自動セットアップとサポートを通して、Lambda 関数の API 体制のガバナンスと API セキュリティを簡素化します。

AWS 管理の拡張機能

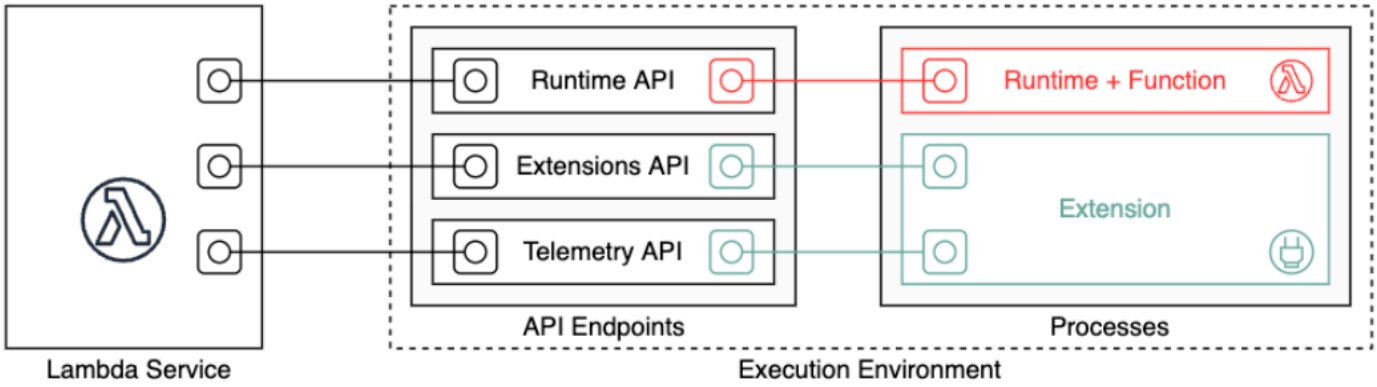
AWS では、次のような独自のマネージド型拡張機能を用意しています。

- AWS AppConfig – 機能フラグと動的データを使用して Lambda 関数を更新します。この拡張機能を使用すると、Ops のスロットリングやチューニングなど、他の動的設定を更新することもできます。
- Amazon CodeGuru Profiler – アプリケーション内で最もコストがかかっているコードの行番号を特定し、コードを改善するための推奨事項を提供することで、アプリケーションのパフォーマンスを向上させ、コストを削減します。
- CloudWatch Lambda Insights – 自動化されたダッシュボードにより、Lambda 関数のパフォーマンスに関するモニタリング、トラブルシューティング、最適化を行います。
- AWS Distro for OpenTelemetry (ADOT) – 関数が、AWS X-Ray などの AWS モニタリングサービス、および Honeycomb や Lightstep などの OpenTelemetry をサポートする宛先にトレースデータを送信できるようにします。
- AWS パラメータとシークレット – お客様が、セキュアな方法でパラメータを AWS Systems Manager パラメータストアから取得し、シークレットを AWS Secrets Manager から取得できるようにします。

追加の拡張機能のサンプルとデモプロジェクトについては、「AWS Lambda Extensions」(拡張機能) を参照してください。

Lambda 拡張機能 API を使用した拡張機能の作成

Lambda 関数の作成者は、拡張機能を使用して、Lambda を、モニタリング、可観測性、セキュリティ、およびガバナンスのための任意のツールと統合します。関数作成者は、AWS、AWS パートナー、およびオープンソースプロジェクトの拡張機能を使用できます。拡張機能の使用の詳細については、AWS Lambda コンピューティングブログの AWS 拡張機能の紹介を参照してください。このセクションでは、Lambda Extensions API、Lambda 実行環境ライフサイクル、および Lambda Extensions API リファレンスを使用する方法を説明します。



拡張機能の作成者は、Lambda Extensions API を使用して Lambda 実行環境に深く統合することができます。拡張機能は、関数および実行環境のライフサイクルイベントに登録できます。これらのイベントに対応して、新しいプロセスを開始し、ロジックを実行し、Lambda ライフサイクルのすべてのフェーズ (初期化、呼び出し、およびシャットダウン) を制御し、これらに参加することができます。さらに、Runtime Logs API を使用して、ログのストリームを受信できます。

拡張機能は、実行環境で独立したプロセスとして実行され、関数の呼び出しが完全に処理された後も引き続き実行できます。拡張機能はプロセスとして実行されるため、関数とは異なる言語で記述できます。コンパイルされた言語を使用して拡張機能を実装することをお勧めします。この場合、拡張機能は、サポートされているランタイムと互換性のある自己完結型のバイナリです。すべての Lambda ランタイム は拡張機能をサポートします。コンパイルされていない言語を使用する場合は、必ず互換性のあるランタイムを拡張機能に含めてください。

Lambda は内部拡張機能もサポートしています。内部拡張機能は、ランタイムプロセスで別のスレッドとして実行されます。ランタイムは、内部拡張を開始および停止します。Lambda 環境と統合する別の方法は、言語固有の環境変数とラッパースクリプトを使用することです。これらを使用して、ランタイム環境を設定し、ランタイムプロセスの起動動作を変更できます。

関数に拡張機能を追加するには、次の 2 つの方法があります。.zip ファイルアーカイブとしてデプロイされた関数では、拡張機能をレイヤーとしてデプロイします。コンテナイメージとして定義された関数の場合は、コンテナイメージに拡張機能を追加します。

> **Note**
>
> 拡張機能とラッパースクリプトの例については、AWS Lambda サンプル GitHub リポジトリの 「AWS拡張」を参照してください。

トピック

- Lambda 実行環境のライフサイクル
- 拡張機能 API リファレンス

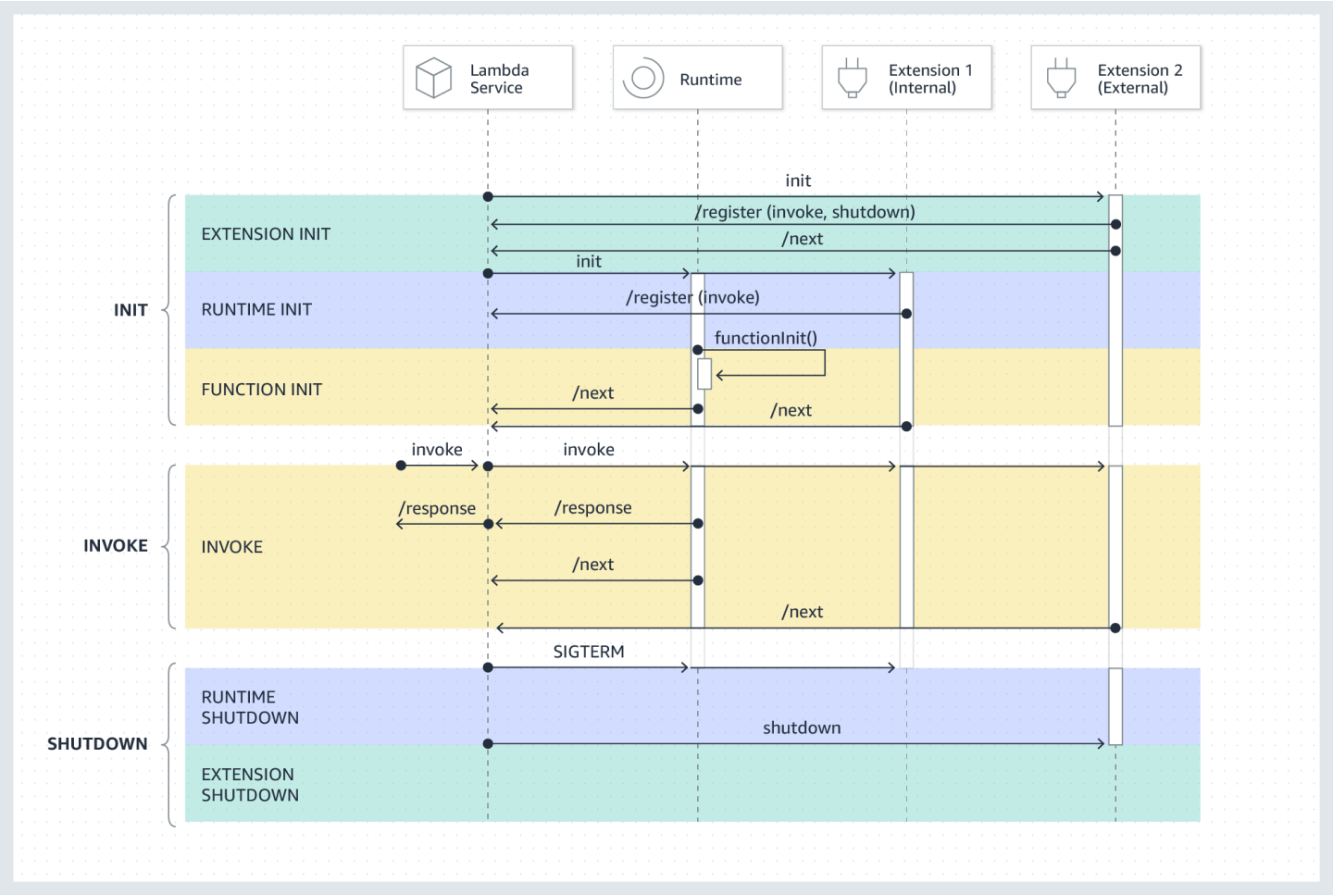
Lambda 実行環境のライフサイクル

実行環境のライフサイクルには、以下のフェーズが含まれています。

- **Init**: このフェーズ中、Lambda は、設定したリソースを使用して実行環境を作成またはフリーズ解除し、関数コードとすべてのレイヤーをダウンロードして、拡張機能とランタイムを初期化し、関数の初期化コード (メインハンドラーの外部にあるコード) を実行します。Init フェーズは、最初の呼び出し中か、プロビジョニング済みの同時実行を有効にしている場合は関数呼び出しの前に、発生します。

  Init フェーズは、Extension init、Runtime init、Function init の 3 つのサブフェーズに分割されます。これらのサブフェーズでは、関数コードが実行される前に、すべての拡張機能とランタイムがそのセットアップタスクを完了します。
- **Invoke**: このフェーズでは、Lambda は関数ハンドラーを呼び出します。関数が実行され、完了すると、Lambda は、別の関数呼び出しを処理する準備をします。
- **Shutdown**: このフェーズは、Lambda 関数が一定期間呼び出しを受け取らなかった場合にトリガーされます。Shutdown フェーズでは、Lambda はランタイムをシャットダウンし、拡張機能にアラートを出してそれらを完全に停止させた後、環境を削除します。Lambda は、各拡張機能に Shutdown イベントを送信します。これにより、環境がシャットダウンされようとしていることを各拡張機能に伝えます。

各フェーズは、Lambda からランタイム、および登録されたすべての拡張機能へのイベントから始まります。ランタイムと各拡張機能は、Next API リクエストを送信することで完了を示します。Lambda は、各プロセスが完了して保留中のイベントがなくなると、実行環境をフリーズします。



トピック

- 初期化フェーズ
- 呼び出しフェーズ
- シャットダウンフェーズ
- アクセス許可と設定
- 障害処理
- 拡張機能のトラブルシューティング

初期化フェーズ

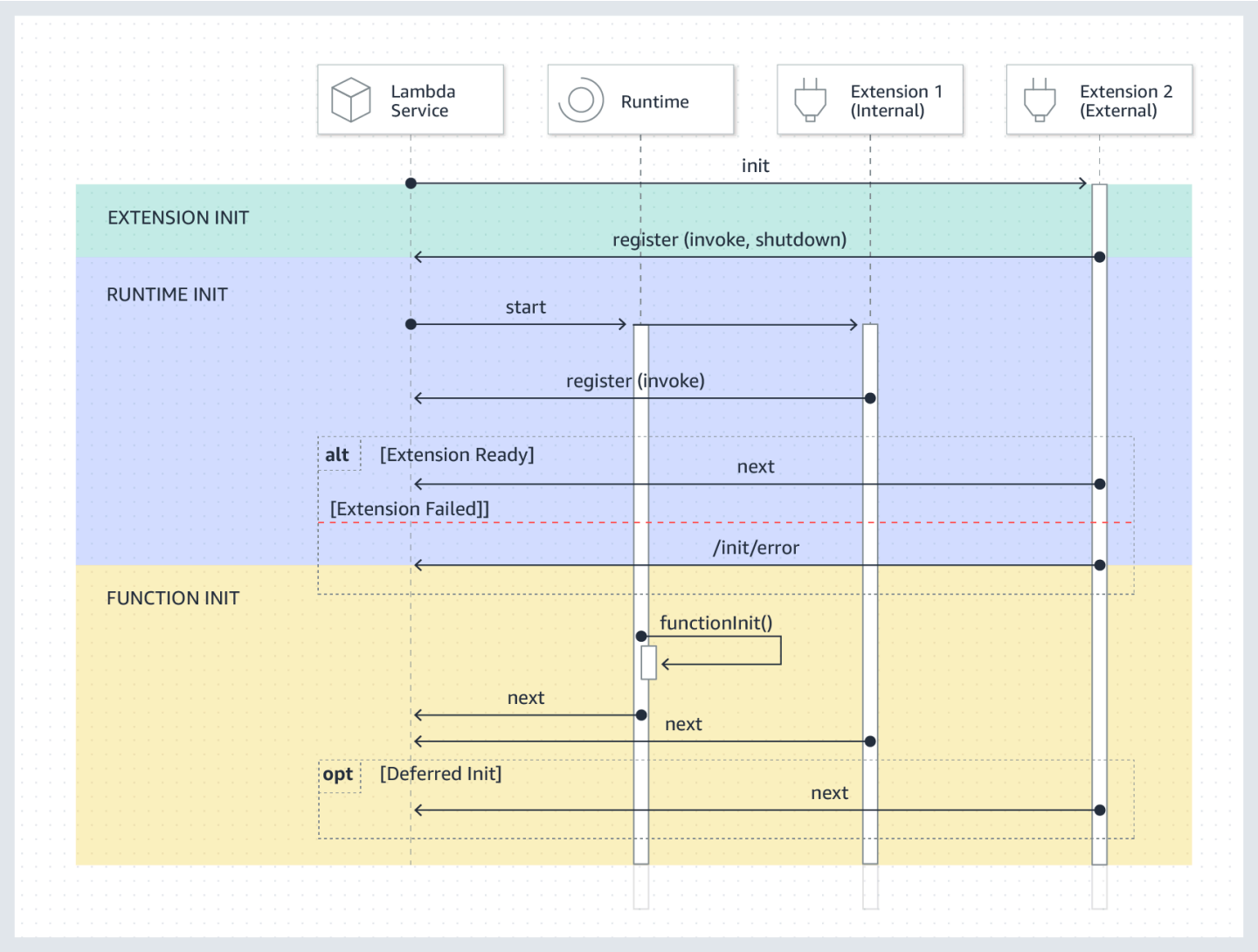
Extension init フェーズの間、各拡張機能モジュールは、イベントを受信するために Lambda に登録されている必要があります。Lambda は、拡張機能がブートストラップシーケンスを完了していることを検証するために、拡張機能の完全なファイル名を使用します。したがって、各 Register API コールには、拡張機能の完全なファイル名を持つ Lambda-Extension-Name ヘッダーを含める必要があります。

1 つの関数に最大 10 個の拡張機能を登録できます。この制限は、Register API コールを通じて適用されます。

各拡張機能が登録されると、Lambda は Runtime init フェーズを開始します。ランタイムプロセスは functionInit を呼び出して Function init フェーズを開始します。

Init フェーズはランタイム後に完了します。登録された各拡張機能は、Next API リクエストを送信することで完了を示します。

> **Note**
>
> 拡張機能は、Init フェーズの任意の時点で初期化を完了できます。



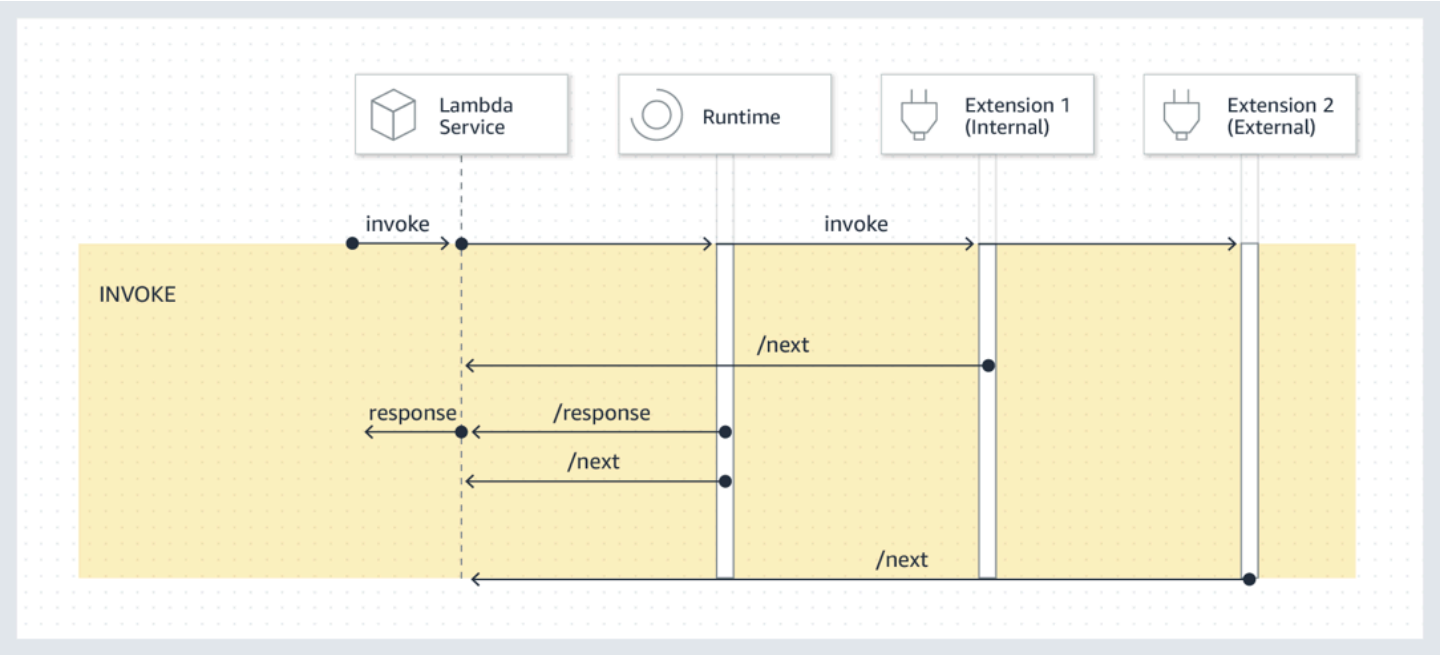
呼び出しフェーズ

Next API リクエストに応答して Lambda 関数が呼び出されると、Lambda は、ランタイム、および Invoke イベントに登録されている各拡張機能に、Invoke イベントを送信します。

呼び出し中、外部拡張機能は関数と並行して実行されます。また、関数が完了した後も、引き続き実行されます。これにより、診断情報をキャプチャしたり、ログ、メトリクス、トレースを任意の場所に送信したりできます。

ランタイムから関数応答を受信した後、拡張機能がまだ実行中であっても、Lambda はクライアントに応答を返します。

Invoke フェーズはランタイム後に終了します。すべての拡張機能は、Next API リクエストを送信することによって完了を示します。



イベントペイロード：ランタイムに送信されるイベント (および Lambda　関数) は、リクエスト、ヘッダー (RequestId など) およびペイロード全体を保持します。各拡張機能に送信されるイベントには、イベントの内容を説明するメタデータが含まれます。このライフサイクルイベントには、イベントのタイプ、関数のタイムアウト時間 (deadlineMs) 、requestId、呼び出された関数の Amazon リソースネーム (ARN) 、およびトレースヘッダーが含まれます。

関数イベント本体にアクセスする拡張機能は、その拡張機能と通信するインランタイム SDK を使用できます。関数のデベロッパーは、関数が呼び出されたときにインランタイム SDK を使用して、拡張機能にペイロードを送信します。

ペイロードの例を次に示します。

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

所要時間の制限: 関数のタイムアウト設定では、Invoke フェーズ全体の所要時間を制限します。例えば、関数のタイムアウトを 360 秒に設定した場合、関数とすべての拡張機能は 360 秒以内に完了する必要があります。独立した呼び出し後フェーズはないことに注意してください。所要時間はランタイムおよびすべての拡張機能の呼び出しが完了するまでの合計時間であり、関数およびすべての拡張機能の実行が終了するまで計算されません。

パフォーマンスの影響と拡張機能のオーバーヘッド: 拡張機能は、関数のパフォーマンスに影響を与える可能性があります。拡張機能の作成者は、拡張機能のパフォーマンスへの影響を制御する必要があります。例えば、拡張機能でコンピューティング負荷の高い操作を実行した場合、拡張機能と関数コードで同じ CPU リソースを共有するため、関数の実行時間が長くなります。さらに、関数呼び出しの完了後に拡張機能が広範な操作を実行する場合、すべての拡張機能が完了を示すまで Invoke フェーズが継続するため、関数の実行時間が長くなります。

> **Note**
>
> Lambda は、関数のメモリ設定に比例して CPU パワーを割り当てます。関数と拡張機能のプロセスが同じ CPU リソースで競合するため、メモリ設定が小さい場合、実行時間と初期化時間が長くなることがあります。実行時間と初期化時間を短縮するには、メモリ設定を引き上げてみてください。

Invoke フェーズで拡張機能によって発生したパフォーマンスへの影響を確認できるように、Lambda は PostRuntimeExtensionsDuration メトリクスを出力します。このメトリクスでは、ランタイム Next API リクエストから最後の拡張機能の Next API リクエストまでの累積時間が測定されます。使用されるメモリの増加を測定するには、MaxMemoryUsed メトリクスを使用します。関数のメトリクスの詳細については、「Lambda 関数のメトリクスの使用」を参照してください。

関数のデベロッパーは、異なるバージョンの関数を並行して実行して、特定の拡張機能の影響を把握することができます。拡張機能の作成者は、関数のデベロッパーが適切な拡張機能を選択しやすくするために、予想されるリソース消費を公開することをお勧めします。

シャットダウンフェーズ

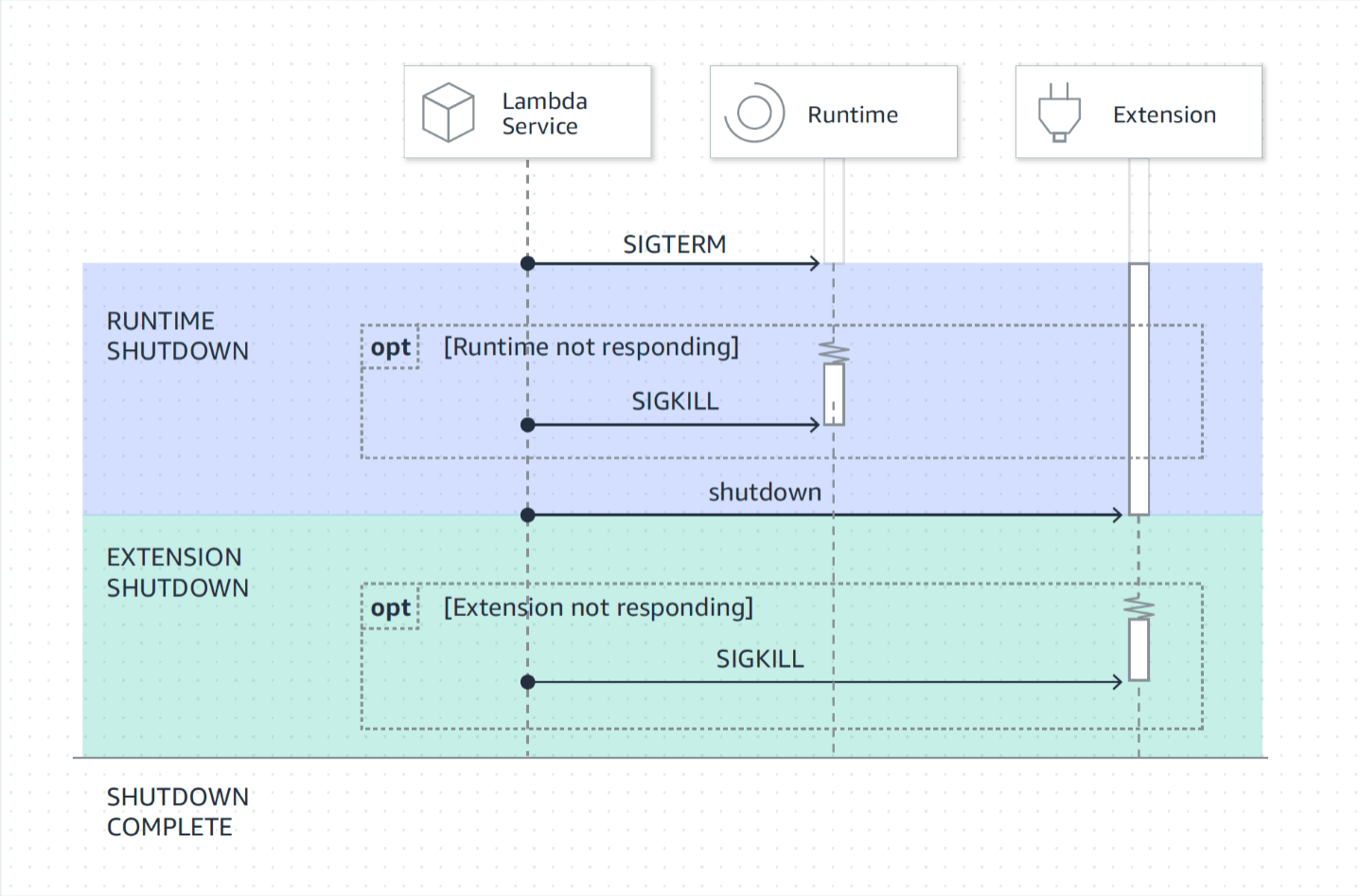
Lambda は、ランタイムをシャットダウンしようとする際に、登録された各外部拡張機能に Shutdown を送信します。拡張機能は、この時間を最終的なクリーンアップタスクに使用できます。Shutdown イベントは Next API リクエストに応答して送信されます。

所要時間の制限: Shutdown フェーズの最大所要時間は、登録された拡張機能の設定によって異なります。

- 0 ms – 登録された拡張機能を持たない関数
- 500 ms - 登録された内部拡張機能を持つ関数
- 2000 ms - 登録された外部拡張機能を 1 つ以上持つ関数

外部拡張機能を持つ関数の場合、Lambda はランタイムプロセスがグレースフルシャットダウンを実行するために、最大 300 ms (内部拡張機能を持つランタイムでは 500 ms) を予約します。Lambda は、外部拡張機能をシャットダウンするために 2,000 ミリ秒の制限の残りを割り当てます。

ランタイムまたは拡張機能が制限内で Shutdown イベントに応答しない場合、Lambda は SIGKILL の通知を使用してプロセスを終了します。



イベントペイロード: Shutdown イベントには、シャットダウンの理由と残り時間 (ミリ秒単位) が含まれます。

shutdownReason には次の値が含まれています。

- SPINDOWN - 正常なシャットダウン
- TIMEOUT - 所要時間の制限がタイムアウトしました
- FAILURE - エラー状態 (out-of-memory イベントなど)

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

アクセス許可と設定

拡張機能は、Lambda 関数と同じ実行環境で実行されます。拡張機能は、CPU、メモリ、/tmp ディスクストレージなどのリソースも関数と共有します。さらに、関数と同じ AWS Identity and Access Management (IAM) ロールとセキュリティコンテキストが使用されます。

ファイルシステムとネットワークアクセス許可: 拡張機能は、関数ランタイムと同じファイルシステムおよびネットワーク名の名前空間で実行されます。つまり、拡張機能は関連するオペレーティングシステムと互換性がある必要があります。拡張機能で追加の 送信ネットワークトラフィック ルールが必要な場合は、これらのルールを関数の設定に適用する必要があります。

> **Note**
>
> 関数コードディレクトリは読み取り専用であるため、拡張機能は関数コードを変更できません。

環境変数: 拡張機能は、関数の環境変数にアクセスできます。ただし、ランタイムプロセスに固有の次の変数は除きます。

- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME

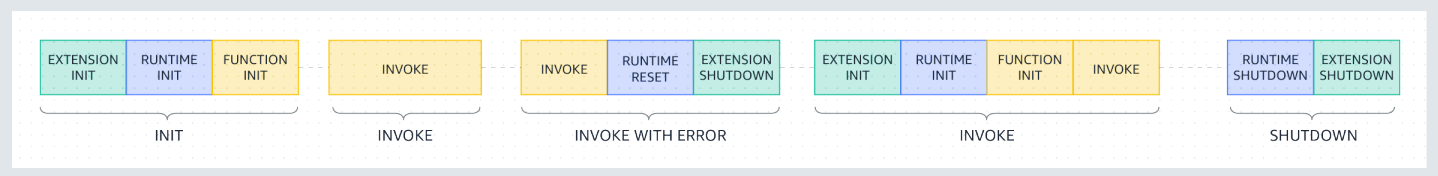
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR
- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

障害処理

初期化の失敗: 拡張機能が失敗した場合、Lambda は実行環境を再起動して一貫した動作を適用し、拡張機能のフェイルファストを推奨します。また、お客様によっては、拡張機能が、ログ記録、セキュリティ、ガバナンス、テレメトリ収集などのミッションクリティカルなニーズを満たす必要があります。

呼び出しの失敗 (メモリ不足、関数のタイムアウトなど) : 拡張機能はランタイムとリソースを共有するため、メモリが消耗した場合に影響を受けます。ランタイムが失敗すると、すべての拡張機能とランタイム自体が Shutdown フェーズに参加します。さらにランタイムは、現在の呼び出しの一部として、または遅延された再初期化メカニズムを通じて、自動的に再起動されます。

Invoke 中に障害が発生した場合 (関数のタイムアウトやランタイムエラーなど)、 Lambda サービスはリセットを実行します。リセットは Shutdown イベントのように動作します。まず、Lambda はランタイムをシャットダウンし、登録された各外部拡張機能に Shutdown イベントを送信します。イベントには、シャットダウンの理由が含まれます。この環境が新しい呼び出しに使用される場合、拡張機能とランタイムは次の呼び出しの一部として再初期化されます。



前示の図の詳細については、「呼び出しフェーズ中の失敗」を参照してください。

拡張機能のログ: Lambda は、拡張機能のログ出力を CloudWatch Logs に送信します。また Lambda は、Init 中に各拡張機能に対して追加のログイベントも生成します。ログイベントは、成功時には名前と登録設定 (event、config) を記録し、失敗時には失敗理由を記録します。

拡張機能のトラブルシューティング

- Register リクエストが失敗した場合は、Lambda-Extension-Name API コールの Register ヘッダーに拡張機能の完全なファイル名が含まれていることを確認します。
- 内部拡張機能に対する Register リクエストが失敗した場合は、そのリクエストが Shutdown イベントに登録されていないことを確認します。

拡張機能 API リファレンス

拡張機能 API バージョン 2020-01-01 の OpenAPI 仕様は、こちらから入手できます: extensions-api.zip

API エンドポイントの値は、AWS_LAMBDA_RUNTIME_API 環境変数から取得できます。Register リクエストを送信するには、各 API パスの前にプレフィックス 2020-01-01/ を使用します。以下に例を示します。

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API メソッド

- 登録
- 次へ
- 初期化エラー
- 終了エラー

登録

Extension init 中、すべての拡張機能は、イベントを受信するために Lambda に登録される必要があります。Lambda は、拡張機能がブートストラップシーケンスを完了していることを検証するために、拡張機能の完全なファイル名を使用します。したがって、各 Register API コールには、拡張機能の完全なファイル名を持つ Lambda-Extension-Name ヘッダーを含める必要があります。

内部拡張機能はランタイムプロセスによって開始および停止されるため、Shutdown イベントへの登録は許可されません。

パス – /extension/register

メソッド - POST

リクエストヘッダー

- Lambda-Extension-Name - 拡張機能の完全なファイル名。必須: はい。タイプ: 文字列。
- Lambda-Extension-Accept-Feature – これを使用して、登録時にオプションの拡張機能を指定します。必須: いいえ。型: カンマで区切られた文字列。この設定を使用して指定できる機能:
  - accountId – これを指定した場合は、拡張機能登録レスポンスに、拡張機能を登録している Lambda 関数に関連付けられたアカウント ID が含まれます。

リクエストボディのパラメータ

- events - 登録するイベントの配列。必須: いいえ。型: 文字列の配列 有効な文字列: INVOKE、SHUTDOWN。

レスポンスヘッダー

- Lambda-Extension-Identifier - それ以降のすべてのリクエストに必要な、生成された一意のエージェント識別子 (UUID 文字列)。

レスポンスコード

- 200 - レスポンス本文には、関数名、関数バージョン、およびハンドラー名が含まれます。
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error 回復不能な状態。拡張機能はすぐに終了する必要があります。

Example リクエストボディの例

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example レスポンスの例

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
```

```
}
```

Example オプションの accountId 機能が含まれたレスポンスボディの例

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

次へ

拡張機能は Next API リクエストを送信して、次のイベント (Invokeイベントまたは Shutdown イベント) を受信します。レスポンス本文にはペイロードが含まれます。ペイロードは、イベントデータを含む JSON ドキュメントです。

拡張機能は、新しいイベントを受信する準備ができていることを示す Next API リクエストを送信します。これはブロック呼び出しです。

拡張機能は一定期間中断される可能性があるため、返されるイベントが発生するまで、GET 呼び出しにタイムアウトを設定しないでください。

パス – /extension/event/next

メソッド - GET

リクエストヘッダー

- Lambda-Extension-Identifier - 拡張機能の一意の識別子 (UUID 文字列)。必須: はい。型: UUID 文字列。

レスポンスヘッダー

- Lambda-Extension-Event-Identifier – イベントの一意の識別子 (UUID 文字列)。

レスポンスコード

- 200 - レスポンスには、次のイベント (EventInvoke または EventShutdown) に関する情報が含まれます。

- 403 – Forbidden
- 500 – Container error 回復不能な状態。拡張機能はすぐに終了する必要があります。

初期化エラー

拡張機能はこのメソッドを使用して、初期化エラーを Lambda に報告します。拡張機能が登録後に初期化に失敗した場合に呼び出します。Lambda がエラーを受信すると、それ以降の API コールは成功しません。拡張機能は、Lambda からの応答を受信した後に終了する必要があります。

パス – `/extension/init/error`

メソッド - POST

リクエストヘッダー

- `Lambda-Extension-Identifier` - 拡張機能の一意の識別子。必須: はい。型: UUID 文字列。
- `Lambda-Extension-Function-Error-Type` - 拡張機能が検出したエラータイプ。必須: はい。このヘッダーは、文字列値で構成されています。Lambda はどのような文字列でも受け入れますが、形式は <category.reason> にすることが推奨されます。例:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

リクエストボディのパラメータ

- `ErrorRequest` - エラーに関する情報。必須: いいえ。

このフィールドは、次の構造を持つ JSON オブジェクトです。

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Lambda は、`errorType` として任意の値を受け入れることに注意してください。

次の例は、呼び出しで指定されたイベントデータを関数で解析できなかった Lambda 関数のエラーメッセージを示しています。

Example 関数エラー

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

レスポンスコード

- 202 - Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error 回復不能な状態。拡張機能はすぐに終了する必要があります。

終了エラー

拡張機能は、このメソッドを使用して、終了する前に Lambda にエラーを報告します。予期しない障害が発生したときに呼び出します。Lambda がエラーを受信すると、それ以降の API コールは成功しません。拡張機能は、Lambda からの応答を受信した後に終了する必要があります。

パス – `/extension/exit/error`

メソッド - POST

リクエストヘッダー

- `Lambda-Extension-Identifier` - 拡張機能の一意の識別子。必須: はい。型: UUID 文字列。
- `Lambda-Extension-Function-Error-Type` - 拡張機能が検出したエラータイプ。必須: はい。このヘッダーは、文字列値で構成されています。Lambda はどのような文字列でも受け入れますが、形式は <category.reason> にすることが推奨されます。例:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

リクエストボディのパラメータ

- `ErrorRequest` - エラーに関する情報。必須: いいえ。

このフィールドは、次の構造を持つ JSON オブジェクトです。

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Lambda は、`errorType` として任意の値を受け入れることに注意してください。

次の例は、呼び出しで指定されたイベントデータを関数で解析できなかった Lambda 関数のエラーメッセージを示しています。

Example 関数エラー

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

レスポンスコード

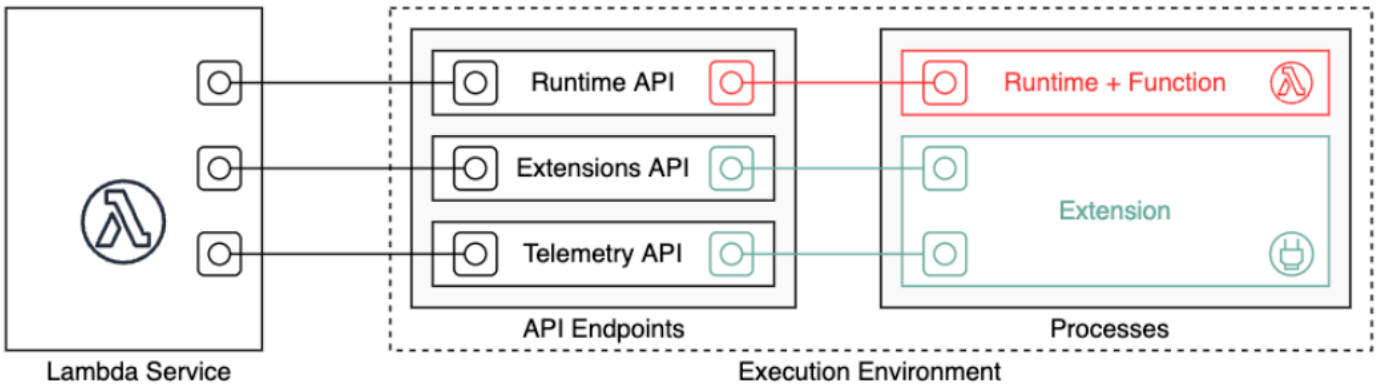
- 202 - Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error 回復不能な状態。拡張機能はすぐに終了する必要があります。

Lambda Telemetry API

Telemetry API を使用することで、拡張機能はテレメトリデータを Lambda から直接受信できます。Lambda は、関数の初期化および呼び出し中に、ログ、プラットフォームメトリクス、およびプラットフォームトレースなどのテレメトリを自動的に取得します。Telemetry API により、拡張機能はこのテレメトリデータを Lambda からほぼリアルタイムで直接取得できます。

Lambda 実行環境内で、Lambda 拡張機能をテレメトリストリームにサブスクライブできます。サブスクライブ後、Lambda は自動的にすべてのテレメトリデータを拡張機能に送信します。そのデータを処理、フィルタリング、送信し、Amazon Simple Storage Service (Amazon S3) バケットや、サードパーティのオブザーバビリティツールプロバイダーなどの目的の宛先に配信することができます。

以下の図は、Extensions API と Telemetry API が、実行環境内から拡張機能を Lambda にリンクする方法を示しています。さらに Runtime API も、ランタイムと関数を Lambda に接続します。



> **Important**
>
> Lambda Telemetry API は、Lambda Log API に取って代わる API です。Logs API は引き続き完全に機能しますが、今後は Telemetry API のみを使用することをお勧めします。拡張機能は、Telemetry API または Logs API のいずれかを使用して、テレメトリストリームにサブスクライブできます。これらの API のいずれかを使用してサブスクライブした後で、もう一方の API を使用してサブスクライブしようとすると、エラーが返されます。

拡張機能は、Telemetry API を使用して 3 つの異なるテレメトリストリームにサブスクライブできます。

- プラットフォームテレメトリ – 実行環境ランタイムライフサイクル、拡張機能ライフサイクル、および関数の呼び出しに関連するイベントとエラーを説明するログ、メトリクス、およびトレース。
- 関数ログ – Lambda 関数コードが生成するカスタムログ。
- 拡張機能ログ – Lambda 拡張機能コードが生成するカスタムログ。

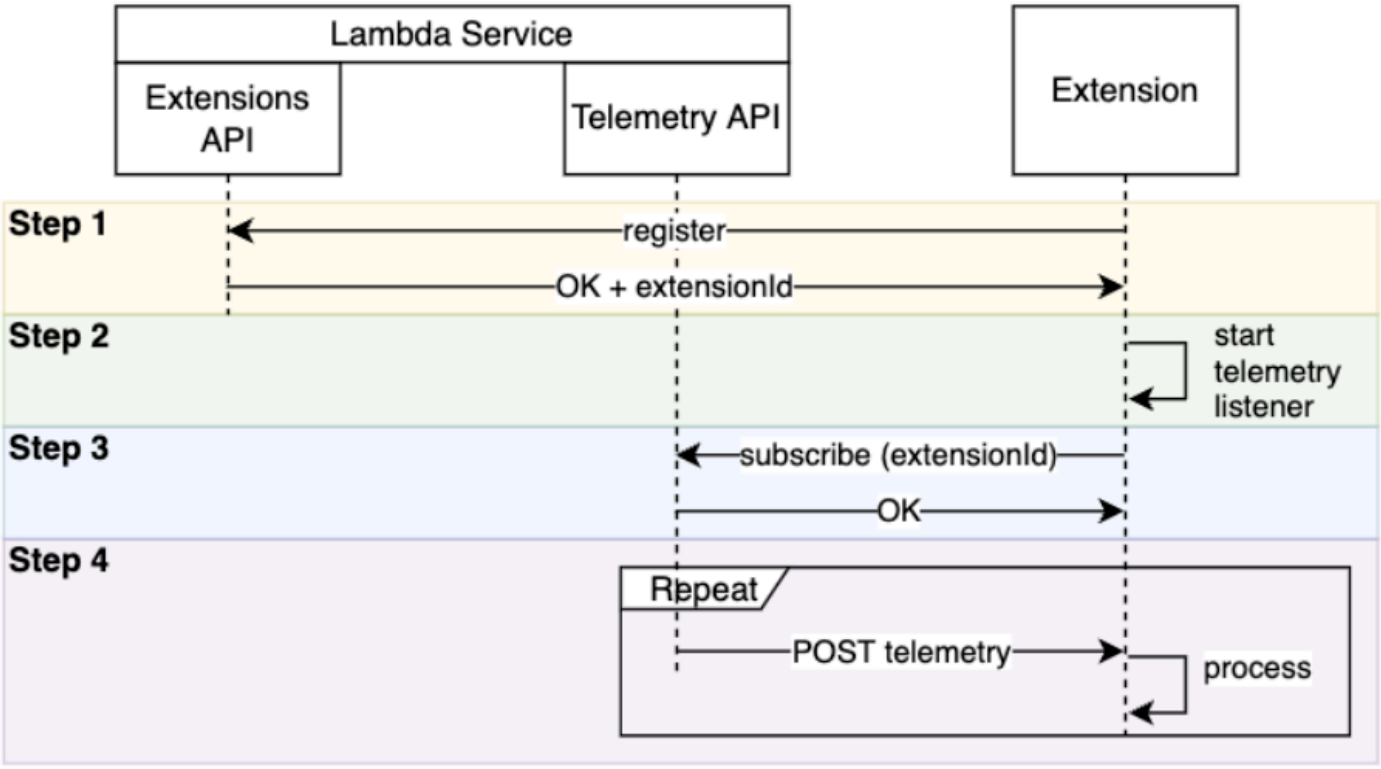
> **Note**
>
> Lambda は、拡張機能がテレメトリストリームにサブスクライブしている場合でも CloudWatch、ログとメトリクスを に送信し、トレースを X-Ray (トレースを有効にしている場合) に送信します。

セクション

Telemetry API を使用した拡張機能の作成

Lambda 拡張機能は、実行環境で独立したプロセスとして実行されます。拡張機能は、関数の呼び出しが完了した後も引き続き実行できます。拡張機能は別個のプロセスであるため、関数コードとは異なる言語で記述することができます。拡張機能は、Golang や Rust などのコンパイルされた言語を使用して記述することが推奨されます。そうすることで、拡張機能は、サポートされているランタイムとの互換性がある自己完結型のバイナリになります。

以下の図は、Telemetry API を使用してテレメトリデータを受信し、処理する拡張機能を作成するための 4 ステッププロセスを説明しています。



以下は、各ステップの詳しい説明です。

1. the section called “拡張機能 API” を使用して拡張機能を登録します。これによって、この後のステップで必要になる `Lambda-Extension-Identifier` が提供されます。拡張機能の登録方法に関する詳細については、「the section called “拡張機能の登録”」を参照してください。
2. テレメトリリスナーを作成します。これは、基本的な HTTP または TCP サーバーにすることができます。Lambda は、テレメトリリスナーの URI を使用してテレメトリデータを拡張機能に送信します。詳細については、「the section called “テレメトリリスナーの作成”」を参照してください。
3. Telemetry API 内の Subscribe API を使用して、拡張機能を目的のテレメトリストリームにサブスクライブします。このステップには、テレメトリリスナーの URI が必要になります。詳細については、「the section called “Telemetry API へのサブスクリプションリクエストの送信”」を参照してください。
4. テレメトリリスナー経由で Lambda からテレメトリデータを取得します。このデータには、Amazon S3、または外部のオブザーバビリティサービスへのデータのディスパッチなど、あらゆるカスタム処理を実行できます。

> **Note**
>
> Lambda 関数の実行環境は、そのライフサイクルの一環として、複数回起動および停止できます。一般的に、拡張機能コードは関数の呼び出し中に実行されるとともに、シャットダウンフェーズ中にも最大 2 秒間実行されます。テレメトリがリスナーに届いた時点でバッチ処理することをお勧めします。次に、Invoke および Shutdown ライフサイクルイベントを使用して、各バッチを目的の送信先に送信します。

拡張機能の登録

テレメトリデータにサブスクライブする前に、Lambda 拡張機能を登録する必要があります。登録は、拡張機能の初期化フェーズ中に行われます。以下は、拡張機能を登録するための HTTP リクエストの例です。

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

リクエストが成功すると、サブスクライバーは HTTP 200 成功レスポンスを受信します。レスポンスヘッダーには、Lambda-Extension-Identifier が含まれています。レスポンスボディには、関数のその他のプロパティが含まれています。

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

詳細については、「the section called “拡張機能 API リファレンス”」を参照してください。

テレメトリリスナーの作成

Lambda 拡張機能には、Telemetry API からの受信リクエストを処理するリスナーが必要です。以下のコードは、Golang でのテレメトリリスナーの実装例です。

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }
```

```
 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

宛先プロトコルの指定

Telemetry API を使用してテレメトリを受信するためにサブスクライブするときは、宛先 URI だけでなく、宛先プロトコルも指定できます。

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda は、テレメトリの受信のために 2 つのプロトコルを受け入れます。

- HTTP (推奨) – Lambda は、テレメトリを JSON 形式のレコードの配列としてローカル HTTP エンドポイント (`http://sandbox.localdomain:${PORT}/${PATH}`) に配信します。$PATH パラメータはオプションです。Lambda は HTTP のみをサポートし、HTTPS はサポートしません。Lambda は、POST リクエストを通じてテレメトリを配信します。
- TCP – Lambda は、テレメトリを改行区切りの JSON (NDJSON) 形式で TCP ポートに配信します。

> **Note**
>
> TCP ではなく、HTTP を使用することが強く推奨されます。TCP を使用する場合、Lambda プラットフォームはテレメトリをアプリケーションレイヤーにいつ配信するかを確認できません。このため、拡張機能がクラッシュすると、ログが失われる可能性があります。HTTP にこの制限はありません。

サブスクライブしてテレメトリを受信する前に、ローカル HTTP リスナーまたは TCP ポートを確立します。セットアップ中は、以下の点に注意してください。

- Lambda がログを送信するのは実行環境内の宛先のみです。

- Lambda は、リスナーがない場合、または POST リクエストにエラーが発生した場合は、テレメトリの送信を再試行します (バックオフも実施)。テレメトリリスナーがクラッシュした場合は、Lambda が実行環境を再起動した後でテレメトリの受信を再開します。
- Lambda はポート 9001 を予約しています。他のポート番号の制限や推奨事項はありません。

メモリの使用量とバッファリングの設定

実行環境でのメモリ使用量は、サブスクライバーの数に比例して直線的に増加します。各サブスクリプションがテレメトリデータを格納するために新しいメモリバッファを開始するため、サブスクリプションはメモリリソースを消費します。バッファメモリ使用量は、実行環境の全体的なメモリ消費量の一部としてカウントされます。

Telemetry API を使用してテレメトリを受信するようにサブスクライブするときは、テレメトリデータをバッファして、バッチ形式でサブスクライバーに配信できます。メモリの使用量を最適化できるように、バッファリング設定を指定することが可能です。

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

バッファリング設定

パラメータ	説明	デフォルトと制限
`maxBytes`	メモリでバッファするテレメトリの最大量 (バイト単位)。	デフォルト: 262,144 最小: 262,144 最大: 1,048,576
`maxItems`	メモリでバッファするイベントの最大数。	デフォルト: 10,000 最小: 1,000 最大: 10,000

パラメータ	説明	デフォルトと制限
timeoutMs	バッチをバッファする最大時間 (ミリ秒単位)。	デフォルト: 1,000 最小: 25 最大: 30,000

バッファーを設定するときは、次の点に注意してください。

- 入力ストリームのいずれかが閉じられている場合、Lambda はログをフラッシュします。これは、ランタイムがクラッシュした場合などに発生する可能性があります。
- 各サブスクライバーは、サブスクリプションリクエストで独自のバッファリング設定をカスタマイズできます。
- バッファリング設定のコンポーネントが maxBytes の場合、データを読み取るためのバッファーサイズを決定する際には、受信ペイロードのサイズを 2 * maxBytes + metadataBytes と想定してください。考慮すべき metadataBytes 量を判断するには、以下のメタデータを確認してください。Lambda は、以下のようなメタデータを各レコードに追加します。

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- サブスクライバーが着信テレメトリを十分な速さで処理できない場合、あるいは関数コードが大量のログボリュームを生成する場合、Lambda は、メモリ使用率を制限内に収めるためにレコードをドロップする可能性があります。この状況が発生すると、Lambda は platform.logsDropped イベントを送信します。

Telemetry API へのサブスクリプションリクエストの送信

Lambda 拡張機能は、Telemetry API にサブスクリプションリクエストを送信することで、テレメトリデータを受信するためにサブスクライブできます。サブスクリプションリクエストには、拡張機能がサブスクライブするイベントのタイプに関する情報が含まれている必要があります。これに加えて、リクエストには配信先情報とバッファリング設定を含めることができます。

サブスクリプションリクエストを送信する前に、拡張機能 ID (Lambda-Extension-Identifier) が必要になります。Extensions API を使用して拡張機能を登録すると、API レスポンスから拡張機能 ID を取得できます。

サブスクリプションは、拡張機能の初期化フェーズ中に行われます。以下は、プラットフォームテレメトリ、関数ログ、および拡張機能ログの 3 つのテレメトリストリームすべてにサブスクライブするための HTTP リクエストの例です。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

リクエストが成功すると、サブスクライバーが HTTP 200 成功レスポンスを受信します。

```
HTTP/1.1 200 OK
"OK"
```

インバウンド Telemetry API メッセージ

Telemetry API を使用してサブスクライブすると、拡張機能は POST リクエストで Lambda からのテレメトリ受信を自動的に開始します。各 POST リクエスト本文には Event オブジェクトの配列が含まれています。それぞれの Event には以下のスキーマがあります。

```
{
   time: String,
   type: String,
```

```
    record: Object
}
```

- time プロパティは、Lambda プラットフォームがイベントを生成した時刻を定義します。これは、イベントが実際に発生した時刻とは異なります。time の文字列値は、ISO 8601 形式のタイムスタンプです。
- type プロパティは、イベントタイプを定義します。以下の表には、可能な値のすべてが説明されています。
- record プロパティは、テレメトリデータが含まれる JSON オブジェクトを定義します。この JSON オブジェクトのスキーマは、type に応じて異なります。

以下の表は、すべての Event オブジェクトタイプと、各イベントタイプの Telemetry API Event スキーマリファレンスへのリンクをまとめたものです。

Telemetry API メッセージタイプ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
プラットフォームイベント	platform.initStart	関数の初期化が開始されました。	the section called “platform.initStart ” スキーマ
プラットフォームイベント	platform.initRuntimeDone	関数の初期化が完了しました。	the section called “platform.initRuntimeDone ” スキーマ
プラットフォームイベント	platform.initReport	関数の初期化のレポートです。	the section called “platform.initReport ” スキーマ
プラットフォームイベント	platform.start	関数の呼び出しが開始されました。	the section called “platform.start ” スキーマ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
プラットフォームイベント	platform.runtimeDone	ランタイムが、成功または失敗のいずれかでイベントの処理を終了しました。	the section called “platform.runtimeDone” スキーマ
プラットフォームイベント	platform.report	関数の呼び出しのレポートです。	the section called “platform.report” スキーマ
プラットフォームイベント	platform.restoreStart	ランタイムの復元が開始されました。	the section called “platform.restoreStart” スキーマ
プラットフォームイベント	platform.restoreRuntimeDone	ランタイムの復元が完了しました。	the section called “platform.restoreRuntimeDone” スキーマ
プラットフォームイベント	platform.restoreReport	ランタイムの復元のレポート。	the section called “platform.restoreReport” スキーマ
プラットフォームイベント	platform.telemetrySubscription	拡張機能が Telemetry API にサブスクライブしました。	the section called “platform.telemetrySubscription” スキーマ
プラットフォームイベント	platform.logsDropped	Lambda がログエントリをドロップしました。	the section called “platform.logsDropped” スキーマ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
関数ログ	`function`	関数コードからのログ行です。	the section called “`function`” スキーマ
拡張ログ	`extension`	拡張コードからのログ行です。	the section called “`extension `” スキーマ

Lambda Telemetry API リファレンス

Lambda Telemetry API エンドポイントを使用して、拡張機能をテレメトリーストリームにサブスクライブします。Telemetry API エンドポイントは、AWS_LAMBDA_RUNTIME_API 環境変数から取得できます。API リクエストを送信するには、API バージョン (2022-07-01/) と telemetry/ を付加します。例:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

サブスクリプションレスポンスバージョン 2022-12-13 の OpenAPI Specification (OAS) 定義については、以下を参照してください。

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

API 操作

- Subscribe

Subscribe

Lambda 拡張機能は、テレメトリーストリームにサブスクライブするために Subscribe API リクエストを送信できます。

- パス – /telemetry
- メソッド – PUT
- ヘッダー
  - Content-Type: application/json
- リクエストボディパラメータ
  - schemaVersion
    - 必須: はい
    - 型: 文字列
    - 有効な値: "2022-12-13" または "2022-07-01"
  - destination – テレメトリイベントの宛先とイベント配信のプロトコルを定義する設定。
    - 必須: はい
    - 型: オブジェクト

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

- protocol – Lambda がテレメトリデータを送信するために使用するプロトコル。
  - 必須: はい
  - 型: 文字列
  - 有効な値: "HTTP"|"TCP"
- URI – テレメトリデータの送信先になる URI。
  - 必須: はい
  - 型: 文字列
- 詳細については、「the section called “宛先プロトコルの指定”」を参照してください。

- types – 拡張機能がサブスクライブするテレメトリのタイプ。
  - 必須: はい
  - タイプ : 文字列の配列
  - 有効な値: "platform"|"function"|"extension"
- buffering – イベントバッファリングの設定。
  - 必須: いいえ
  - 型: オブジェクト

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - MaxItems – メモリでバッファするイベントの最大数。
    - 必須: いいえ
    - 型: 整数
    - デフォルト: 1,000
    - 最小: 1,000

    - 最大: 10,000
- maxBytes – メモリでバッファするテレメトリの最大量 (バイト単位)。
    - 必須: いいえ
    - 型: 整数
    - デフォルト: 262,144
    - 最小: 262,144
    - 最大: 1,048,576
- timeoutMs - バッチをバッファーする最大時間 (ミリ秒単位) 。
    - 必須: いいえ
    - 型: 整数
    - デフォルト: 1,000
    - 最小: 25
    - 最大: 30,000
- 詳細については、「the section called “メモリの使用量とバッファリングの設定”」を参照してください。

Subscribe API リクエストの例

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2022-12-13",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

Subscribe リクエストが正常に実行されると、拡張機能が HTTP 200 成功レスポンスを受信します。

```
HTTP/1.1 200 OK
"OK"
```

Subscribe リクエストが失敗すると、拡張機能がエラーレスポンスを受信します。例:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

以下は、拡張機能が受信できる追加のレスポンスコードです。

- 200 – リクエストが正常に完了しました
- 202 – リクエストが受理されました。ローカルテスト環境でのサブスクリプションリクエストレスポンス
- 400 – 不正なリクエスト
- 500 – サービスエラー

Lambda Telemetry API **Event** スキーマリファレンス

Lambda Telemetry API エンドポイントを使用して、拡張機能をテレメトリーストリームにサブスクライブします。Telemetry API エンドポイントは、AWS_LAMBDA_RUNTIME_API 環境変数から取得できます。API リクエストを送信するには、API バージョン (2022-07-01/) と telemetry/ を付加します。例:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

サブスクリプションレスポンスバージョン 2022-12-13 の OpenAPI Specification (OAS) 定義については、以下を参照してください。

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

以下の表は、Telemetry API がサポートするすべての Event オブジェクトタイプをまとめたものです。

Telemetry API メッセージタイプ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
プラットフォームイベント	platform.initStart	関数の初期化が開始されました。	the section called “platform.initStart” スキーマ
プラットフォームイベント	platform.initRuntimeDone	関数の初期化が完了しました。	the section called “platform.initRuntimeDone” スキーマ
プラットフォームイベント	platform.initReport	関数の初期化のレポートです。	the section called “platform.initReport” スキーマ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
プラットフォームイベント	platform.start	関数の呼び出しが開始されました。	the section called “platform.start ” スキーマ
プラットフォームイベント	platform.runtimeDone	ランタイムが、成功または失敗のいずれかでイベントの処理を終了しました。	the section called “platform.runtimeDone ” スキーマ
プラットフォームイベント	platform.report	関数の呼び出しのレポートです。	the section called “platform.report ” スキーマ
プラットフォームイベント	platform.restoreStart	ランタイムの復元が開始されました。	the section called “platform.restoreStart ” スキーマ
プラットフォームイベント	platform.restoreRuntimeDone	ランタイムの復元が完了しました。	the section called “platform.restoreRuntimeDone ” スキーマ
プラットフォームイベント	platform.restoreReport	ランタイムの復元のレポート。	the section called “platform.restoreReport ” スキーマ
プラットフォームイベント	platform.telemetrySubscription	拡張機能が Telemetry API にサブスクライブしました。	the section called “platform.telemetrySubscription ” スキーマ

カテゴリ	イベントタイプ	説明	イベントレコードスキーマ
プラットフォームイベント	`platform.logsDropped`	Lambda がログエントリをドロップしました。	the section called “`platform.logsDropped`” スキーマ
関数ログ	`function`	関数コードからのログ行です。	the section called “`function`” スキーマ
拡張ログ	`extension`	拡張コードからのログ行です。	the section called “`extension`” スキーマ

目次

- InitReportMetrics
- InitType
- ReportMetrics
- RestoreReportMetrics
- RuntimeDoneMetrics
- Span
- Status
- TraceContext
- TracingType

Telemetry API **Event** オブジェクトタイプ

このセクションでは、Lambda Telemetry API がサポートする Event オブジェクトのタイプについて詳しく説明します。イベントの説明にある疑問符 (?) は、その属性がオブジェクト内に存在しない可能性があることを示します。

platform.initStart

platform.initStart イベントは、関数の初期化フェーズが開始されたことを示します。platform.initStart Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

PlatformInitStart オブジェクトには以下の属性があります。

- functionName – String
- functionVersion – String
- initializationType – the section called “InitType” オブジェクト
- instanceId? – String
- instanceMaxMemory? – Integer
- phase – the section called “InitPhase” オブジェクト
- runtimeVersion? – String

- runtimeVersionArn? – String

以下は、タイプ platform.initStart の Event の例です。

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.initRuntimeDone

platform.initRuntimeDone イベントは、関数の初期化フェーズが完了したことを示します。platform.initRuntimeDone Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

PlatformInitRuntimeDone オブジェクトには以下の属性があります。

- initializationType – the section called “InitType” オブジェクト
- phase – the section called “InitPhase” オブジェクト
- status – the section called “Status” オブジェクト
- spans? – the section called “Span” オブジェクトのリスト

以下は、タイプ platform.initRuntimeDone の Event の例です。

```
{
```

```
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

platform.initReport

platform.initReport イベントには、関数の初期化フェーズに関する全体的なレポートが含まれています。platform.initReport Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

PlatformInitReport オブジェクトには以下の属性があります。

- errorType? – 文字列
- initializationType – the section called “InitType” オブジェクト
- phase – the section called “InitPhase” オブジェクト
- metrics – the section called “InitReportMetrics” オブジェクト
- spans? – the section called “Span” オブジェクトのリスト
- status – the section called “Status” オブジェクト

以下は、タイプ platform.initReport の Event の例です。

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
```

```
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

platform.start

platform.start イベントは、関数の呼び出しフェーズが開始されたことを示します。platform.start Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

PlatformStart オブジェクトには以下の属性があります。

- requestId – String
- version? – String
- tracing? – the section called “TraceContext”

以下は、タイプ platform.start の Event の例です。

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
```

```
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

platform.runtimeDone

platform.runtimeDone イベントは、関数の呼び出しフェーズが完了したことを示します。platform.runtimeDone Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

PlatformRuntimeDone オブジェクトには以下の属性があります。

- errorType? – String
- metrics? – the section called “RuntimeDoneMetrics” オブジェクト
- requestId – String
- status – the section called “Status” オブジェクト
- spans? – the section called “Span” オブジェクトのリスト
- tracing? – the section called “TraceContext” オブジェクト

以下は、タイプ platform.runtimeDone の Event の例です。

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
```

```
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

platform.report

platform.report イベントには、関数の初期化フェーズに関する全体的なレポートが含まれています。platform.report Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

PlatformReport オブジェクトには以下の属性があります。

- metrics – the section called “ReportMetrics” オブジェクト
- requestId – String
- spans? – the section called “Span” オブジェクトのリスト
- status – the section called “Status” オブジェクト
- tracing? – the section called “TraceContext” オブジェクト

以下は、タイプ platform.report の Event の例です。

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
```

```
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

platform.restoreStart

platform.restoreStart イベントは、関数の環境復元イベントが開始されたことを示します。環境復元イベントでは、Lambda は環境をゼロから初期化するのではなく、キャッシュされたスナップショットから環境を作成します。詳細については、「Lambda SnapStart」を参照してください。platform.restoreStart Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

PlatformRestoreStart オブジェクトには以下の属性があります。

- functionName – String
- functionVersion – String
- instanceId? – String
- instanceMaxMemory? – String
- runtimeVersion? – String
- runtimeVersionArn? – String

以下は、タイプ platform.restoreStart の Event の例です。

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
```

```
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.restoreRuntimeDone

platform.restoreRuntimeDone イベントは、関数の環境復元イベントが完了したことを示します。環境復元イベントでは、Lambda は環境をゼロから初期化するのではなく、キャッシュされたスナップショットから環境を作成します。詳細については、「Lambda SnapStart」を参照してください。platform.restoreRuntimeDone Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

PlatformRestoreRuntimeDone オブジェクトには以下の属性があります。

- errorType? – String
- spans? – the section called “Span” オブジェクトのリスト
- status – the section called “Status” オブジェクト

以下は、タイプ platform.restoreRuntimeDone の Event の例です。

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
```

```
        ]
    }
}
```

platform.restoreReport

platform.restoreReport イベントには、関数の復元イベントに関する全体的なレポートが含まれています。platform.restoreReport Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

PlatformRestoreReport オブジェクトには以下の属性があります。

- errorType? – 文字列
- metrics? – the section called “RestoreReportMetrics” オブジェクト
- spans? – the section called “Span” オブジェクトのリスト
- status – the section called “Status” オブジェクト

以下は、タイプ platform.restoreReport の Event の例です。

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
```

```
}
```

platform.extension

extension イベントには、拡張機能コードからのログが含まれています。extension Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

PlatformExtension オブジェクトには以下の属性があります。

- events – String のリスト
- name – String
- state – String

以下は、タイプ platform.extension の Event の例です。

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

platform.telemetrySubscription イベントには、拡張機能サブスクリプションに関する情報が含まれています。platform.telemetrySubscription Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
```

```
- record: PlatformTelemetrySubscription
```

PlatformTelemetrySubscription オブジェクトには以下の属性があります。

- name – String
- state – String
- types – String のリスト

以下は、タイプ platform.telemetrySubscription の Event の例です。

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

platform.logsDropped イベントには、ドロップされたイベントに関する情報が含まれています。関数が短時間に大量のログを出力して Lambda が処理できない場合、Lambda は platform.logsDropped イベントを発行します。関数が生成するレートで CloudWatch または Telemetry API にサブスクライブされている拡張機能にログを送信できない場合、Lambda は関数の実行速度が低下するのを防ぐためにログを削除します。platform.logsDropped Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

PlatformLogsDropped オブジェクトには以下の属性があります。

- droppedBytes – Integer
- droppedRecords – Integer
- reason – String

以下は、タイプ platform.logsDropped の Event の例です。

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

function

function イベントには、関数コードからのログが含まれています。function Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = function
- record: {}
```

record フィールドの形式は、関数のログがプレーンテキスト形式か JSON 形式かによって異なります。ログ形式設定オプションの詳細については、「the section called “JSON とプレーンテキストのログフォーマットの設定”」を参照してください。

以下は、ログ形式がプレーンテキストであるタイプ function の Event 例です。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

以下は、ログ形式が JSON であるタイプ function の Event 例です。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
```

```
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> 使用しているスキーマのバージョンが 2022-12-13 バージョンよりも古い場合、関数のログ記録形式が JSON として設定されていても、"record" は常に文字列としてレンダリングされます。

extension

extension イベントには、拡張機能コードからのログが含まれています。extension Event オブジェクトは以下のような形状になっています。

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

record フィールドの形式は、関数のログがプレーンテキスト形式か JSON 形式かによって異なります。ログ形式設定オプションの詳細については、「the section called “JSON とプレーンテキストのログフォーマットの設定”」を参照してください。

以下は、ログ形式がプレーンテキストであるタイプ extension の Event 例です。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

以下は、ログ形式が JSON であるタイプ extension の Event 例です。

```
{
```

```
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> 使用しているスキーマのバージョンが 2022-12-13 バージョンよりも古い場合、関数のログ記録形式が JSON として設定されていても、"record" は常に文字列としてレンダリングされます。

共有オブジェクトタイプ

このセクションでは、Lambda Telemetry API がサポートする共有オブジェクトのタイプについて詳しく説明します。

InitPhase

初期化ステップが行われるフェーズを記述する文字列列挙です。ほとんどの場合、Lambda は `init` フェーズ中に関数の初期化コードを実行しますが、一部のエラーケースでは、Lambda が `invoke` フェーズ中に関数の初期化コードを再実行する場合があります。(これは、抑制された初期化と呼ばれます。)

- 型 – `String`
- 有効な値 – `init|invoke|snap-start`

InitReportMetrics

初期化フェーズに関するメトリクスが含まれたオブジェクトです。

- 型 – `Object`

`InitReportMetrics` オブジェクトは以下のように形成されています。

```
InitReportMetrics: Object
- durationMs: Double
```

以下は、InitReportMetrics オブジェクトの例です。

```
{
    "durationMs": 247.88
}
```

InitType

Lambda が環境を初期化した方法を記述する文字列列挙です。

- 型 – String
- 有効な値 – on-demand|provisioned-concurrency

ReportMetrics

完了したフェーズに関するメトリクスが含まれるオブジェクトです。

- 型 – Object

ReportMetrics オブジェクトは以下のように形成されています。

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

以下は、ReportMetrics オブジェクトの例です。

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
```

```
    "memorySizeMB": 128
}
```

RestoreReportMetrics

完了した復元フェーズに関するメトリクスが含まれるオブジェクトです。

- 型 – Object

RestoreReportMetrics オブジェクトは以下のように形成されています。

```
RestoreReportMetrics: Object
- durationMs: Double
```

以下は、RestoreReportMetrics オブジェクトの例です。

```
{
    "durationMs": 15.19
}
```

RuntimeDoneMetrics

呼び出しフェーズに関するメトリクスが含まれるオブジェクトです。

- 型 – Object

RuntimeDoneMetrics オブジェクトは以下のように形成されています。

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

以下は、RuntimeDoneMetrics オブジェクトの例です。

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

スパンに関する詳細情報が含まれるオブジェクトです。スパンは、トレース内の作業または操作の単位を表します。スパンの詳細については、OpenTelemetry Docs ウェブサイトの「Tracing API」ページにある「Span」(スパン) を参照してください。

Lambda は、platform.RuntimeDone イベントに対して次のスパンをサポートしています。

- responseLatency スパンは、Lambda 関数がレスポンスの送信を開始するまでにかかった時間を表します。
- responseDuration スパンは、Lambda 関数がレスポンス全体の送信を完了するまでにかかった時間を表します。
- runtimeOverhead スパンは、Lambda ランタイムが次の関数呼び出しを処理する準備ができたことを通知するまでにかかった時間を表します。これは、関数のレスポンスを返した後に、次のイベントを取得するための次の呼び出し API をランタイムが呼び出すまでにかかった時間です。

以下は、responseLatency スパンオブジェクトの例です。

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

初期化または呼び出しフェーズのステータスを記述するオブジェクトです。ステータスが failure または error の場合、Status オブジェクトにはエラーを記述する errorType フィールドも含まれています。

- 型 – Object
- 有効なステータス値 – success|failure|error|timeout

TraceContext

トレースのプロパティを記述するオブジェクトです。

- 型 – Object

TraceContext オブジェクトは以下のように形成されています。

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

以下は、TraceContext オブジェクトの例です。

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

the section called “TraceContext” オブジェクト内のトレーシングのタイプを記述する文字列列挙です。

- 型 – String
- 有効な値 – X-Amzn-Trace-Id

Lambda Telemetry API **Event** オブジェクトを OpenTelemetryスパンに変換する

AWS Lambda Telemetry API スキーマは OpenTelemetry 、 (OTel) と意味的に互換性があります。つまり、AWS LambdaTelemetry API Event オブジェクトを OpenTelemetry (OTel) スパンに変換できます。変換するときは、単一の Event オブジェクトを単一の OTel スパンにマップしないようにしてください。その代わりに、ライフサイクルフェーズに関連する 3 つのイベントすべてを 単一の OTel スパンで提示する必要があります。例えば、start、runtimeDone、および runtimeReport イベントは、単一の関数呼び出しを表します。これら 3 つのイベントのすべてを単一の OTel スパンとして提示します。

イベントは、スパンイベントまたは子 (ネストされた) スパンを使用して変換できます。このページの表には、両方のアプローチに関する Telemetry API スキーマプロパティと OTel スパンプロパティ間のマッピングが説明されています。OTel スパンの詳細については、 OpenTelemetry Docs ウェブサイトのトレース API ページの「スパン」を参照してください。

セクション

スパンイベントを使用して OTel スパンにマップする

以下の表にある e は、テレメトリソースからのイベントを表しています。

***Start** イベントのマッピング

OpenTelemetry	Lambda Telemetry API スキーマ
Span.Name	拡張機能は、type フィールドに基づいてこの値を生成します。
Span.StartTime	e.time を使用します。
Span.EndTime	イベントがまだ完了していないことから、該当するものはありません。
Span.Kind	Server に設定します。
Span.Status	Unset に設定します。

OpenTelemetry	Lambda Telemetry API スキーマ
Span.TraceId	e.tracing.value にある AWS X-Ray ヘッダーを解析してから、TraceId 値を使用します。
Span.ParentId	e.tracing.value にある X-Ray ヘッダーを解析してから、Parent 値を使用します。
Span.SpanId	利用できる場合は、e.tracing.spanId を使用します。利用できない場合は、新しい SpanId を生成します。
Span.SpanContext.TraceState	X-Ray トレースコンテキストについては、該当するものはありません。
Span.SpanContext.TraceFlags	e.tracing.value にある X-Ray ヘッダーを解析してから、Sampled 値を使用します。
Span.Attributes	拡張機能は、ここに任意のカスタム値を追加できます。

***RuntimeDone** イベントのマッピング

OpenTelemetry	Lambda Telemetry API スキーマ
Span.Name	拡張機能は、type フィールドに基づいて値を生成します。
Span.StartTime	一致する *Start イベントからの e.time を使用します。 または、e.time - e.metrics.durationMs を使用します。
Span.EndTime	イベントがまだ完了していないことから、該当するものはありません。
Span.Kind	Server に設定します。

OpenTelemetry	Lambda Telemetry API スキーマ
Span.Status	e.status が success ではない場合は、Error に設定します。 それ以外の場合は、Ok に設定します。
Span.Events[]	e.spans[] を使用します。
Span.Events[i].Name	e.spans[i].name を使用します。
Span.Events[i].Time	e.spans[i].start を使用します。
Span.TraceId	e.tracing.value にある AWS X-Ray ヘッダーを解析してから、TraceId 値を使用します。
Span.ParentId	e.tracing.value にある X-Ray ヘッダーを解析してから、Parent 値を使用します。
Span.SpanId	*Start イベントからのものと同じ SpanId を使用します。利用できない場合は、e.tracing.spanId を使用するか、新しい SpanId を生成します。
Span.SpanContext.TraceState	X-Ray トレースコンテキストについては、該当するものはありません。
Span.SpanContext.TraceFlags	e.tracing.value にある X-Ray ヘッダーを解析してから、Sampled 値を使用します。
Span.Attributes	拡張機能は、ここに任意のカスタム値を追加できます。

***Report** イベントのマッピング

OpenTelemetry	Lambda Telemetry API スキーマ
Span.Name	拡張機能は、type フィールドに基づいて値を生成します。
Span.StartTime	一致する *Start イベントからの e.time を使用します。 または、e.time - e.metrics.durationMs を使用します。
Span.EndTime	e.time を使用します。
Span.Kind	Server に設定します。
Span.Status	*RuntimeDone イベントと同じ値を使用します。
Span.TraceId	e.tracing.value にある AWS X-Ray ヘッダーを解析してから、TraceId 値を使用します。
Span.ParentId	e.tracing.value にある X-Ray ヘッダーを解析してから、Parent 値を使用します。
Span.SpanId	*Start イベントからのものと同じ SpanId を使用します。利用できない場合は、e.tracing.spanId を使用するか、新しい SpanId を生成します。
Span.SpanContext.TraceState	X-Ray トレースコンテキストについては、該当するものはありません。
Span.SpanContext.TraceFlags	e.tracing.value にある X-Ray ヘッダーを解析してから、Sampled 値を使用します。
Span.Attributes	拡張機能は、ここに任意のカスタム値を追加できます。

子スパンを使用して OTel スパンにマップする

以下の表には、*RuntimeDone スパンの子 (ネストされた) スパンを使用して、Lambda Telemetry API イベントを OTel スパンに変換する方法が説明されています。*Start および *Report マッピングについては、これらは子スパンでも同様であるため、the section called “スパンイベントを使用して OTel スパンにマップする” の表を参照してください。この表の e は、テレメトリソースからのイベントを表しています。

***RuntimeDone** イベントのマッピング

OpenTelemetry	Lambda Telemetry API スキーマ
Span.Name	拡張機能は、type フィールドに基づいて値を生成します。
Span.StartTime	一致する *Start イベントからの e.time を使用します。 または、e.time - e.metrics.durationMs を使用します。
Span.EndTime	イベントがまだ完了していないことから、該当するものはありません。
Span.Kind	Server に設定します。
Span.Status	e.status が success ではない場合は、Error に設定します。 それ以外の場合は、Ok に設定します。
Span.TraceId	e.tracing.value にある AWS X-Ray ヘッダーを解析してから、TraceId 値を使用します。
Span.ParentId	e.tracing.value にある X-Ray ヘッダーを解析してから、Parent 値を使用します。
Span.SpanId	*Start イベントからのものと同じ SpanId を使用します。利用できない場合

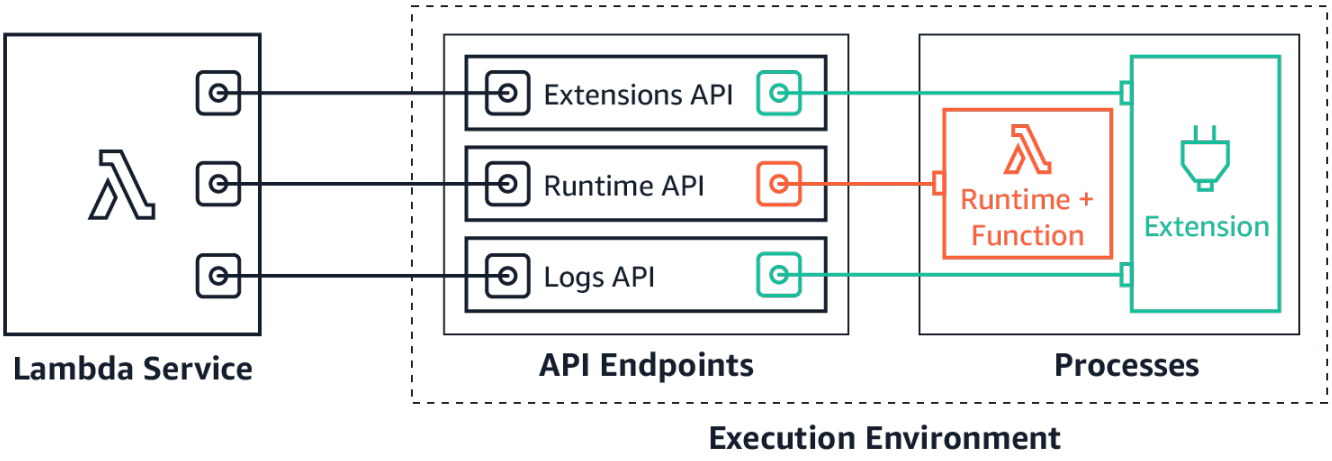
OpenTelemetry	Lambda Telemetry API スキーマ
	は、e.tracing.spanId を使用するか、新しい SpanId を生成します。
Span.SpanContext.TraceState	X-Ray トレースコンテキストについては、該当するものはありません。
Span.SpanContext.TraceFlags	e.tracing.value にある X-Ray ヘッダーを解析してから、Sampled 値を使用します。
Span.Attributes	拡張機能は、ここに任意のカスタム値を追加できます。
ChildSpan[i].Name	e.spans[i].name を使用します。
ChildSpan[i].StartTime	e.spans[i].start を使用します。
ChildSpan[i].EndTime	e.spans[i].start + e.spans[i].durations を使用します。
ChildSpan[i].Kind	親 Span.Kind と同じ。
ChildSpan[i].Status	親 Span.Status と同じ。
ChildSpan[i].TraceId	親 Span.TraceId と同じ。
ChildSpan[i].ParentId	親 Span.SpanId を使用します。
ChildSpan[i].SpanId	新しい SpanId を生成します。
ChildSpan[i].SpanContext.TraceState	X-Ray トレースコンテキストについては、該当するものはありません。
ChildSpan[i].SpanContext.TraceFlags	親 Span.SpanContext.TraceFlags と同じ。

Lambda ログ API

> **Important**
>
> Lambda Telemetry API は、Lambda Log API に取って代わる API です。Logs API は引き続き完全に機能しますが、今後は Telemetry API のみを使用することをお勧めします。拡張機能は、Telemetry API または Logs API のいずれかを使用して、テレメトリストリームにサブスクライブできます。これらの API のいずれかを使用してサブスクライブした後で、もう一方の API を使用してサブスクライブしようとすると、エラーが返されます。

Lambda はランタイムログを自動的にキャプチャし、Amazon にストリーミングします CloudWatch。このログストリームには、 関数コードと拡張機能が生成するログと、Lambda が関数呼び出しの一部として生成するログが含まれます。

Lambda の拡張機能では、Lambda ランタイムログ API を使用して、Lambda 実行環境で直接ログストリームをサブスクライブできます。Lambda は、ログを拡張機能にストリームし、拡張機能は、ログを処理、フィルタリングして、指定された宛先に送信します。



Logs API を使用すると、拡張機能は 3 つの異なるログストリームにサブスクライブできます。

- Lambda 関数が生成して、`stdout` または `stderr` に書き込む関数ログ。
- 拡張コードが生成する拡張ログ。
- Lambda プラットフォームログ。呼び出しと拡張機能に関連するイベントとエラーを記録します。

> **Note**
>
> Lambda は、拡張機能が 1 つ以上のログストリームにサブスクライブしている場合でも CloudWatch、すべてのログを に送信します。

トピック

サブスクライブしてログを受信する

Lambda 拡張機能は、Logs API にサブスクリプションリクエストを送信することで、ログを受信するようサブスクライブできます。

ログを受信するためにサブスクライブするには、内線識別子 (`Lambda-Extension-Identifier`) が必要です。最初に 内線番号を登録 し、内線番号を受け取ります。そして 初期化中 に Logs API にサブスクライブします。初期化フェーズの完了後は、Lambda はサブスクリプションリクエストを処理しません。

> **Note**
>
> Logs API サブスクリプションは冪等です。サブスクリプションリクエストが重複しても、サブスクリプションが重複することはありません。

メモリ使用量

サブスクライバの数が増加すると、メモリ使用量は直線的に増加します。サブスクリプションは、各サブスクリプションが新しいメモリバッファを開き、ログを保存するため、メモリリソースを消費し

ます。メモリ使用量を最適化するために、バッファリング設定 を調整できます。バッファメモリ使用量は、実行環境の全体的なメモリ消費量の一部としてカウントされます。

送信先プロトコル

ログを受信するには、次のいずれかのプロトコルを選択できます。

1. HTTP (推奨) - Lambda は JSON 形式の記録の配列として、ローカル HTTP エンドポイント (`http://sandbox.localdomain:${PORT}/${PATH}`) にログを配信します。$PATH パラメータはオプションです。HTTP のみがサポートされ、HTTPS はサポートされません。ログを受信するには、PUT または POST のいずれかを選択できます。
2. TCP - Lambda は改行区切りの JSON (NDJSON) 形式で TCPポートにログを配信します。

TCP ではなく HTTP を使用することをお勧めします。TCP では、Lambda プラットフォームはログをアプリケーション層に配信するときに確認できません。したがって、拡張機能がクラッシュすると、ログが失われる可能性があります。HTTP はこの制限を共有しません。

また、サブスクライブしてログを受信する前に、ローカル HTTP リスナーまたは TCP ポートを設定することをお勧めします。セットアップ中に、次の点に注意してください。

- Lambda は、実行環境内の送信先にのみログを送信します。
- Lambda は、リスナーがない場合、または POST や PUT リクエストの結果でエラーが発生した場合は、ログ送信の試行を再試行します (バックオフあり)。ログサブスクライバーがクラッシュした場合、Lambda が実行環境を再起動した後もログを受信し続けます。
- Lambda がポート 9001 を予約しています。他のポート番号の制限や推奨事項はありません。

バッファリング構成

Lambda はログをバッファリングし、サブスクライバに配信できます。サブスクリプションリクエストでこの動作を設定するには、次のオプションフィールドを指定します。Lambda では、指定しないフィールドにはデフォルト値が使用されます。

- timeoutMs - バッチをバッファーする最大時間 (ミリ秒単位) 。デフォルト: 1,000。最小: 25 最大: 30,000。
- maxBytes - メモリにバッファするログの最大サイズ (バイト単位) 。デフォルト: 262,144。最小: 262,144。最大: 1,048,576。

- MaxItems - メモリにバッファするイベントの最大数。デフォルト: 10,000。最小: 1,000。最大: 10,000。

バッファリングの設定時には、次の点に注意してください。

- Lambda は、ランタイムがクラッシュした場合など、入力ストリームが閉じられている場合、ログをフラッシュします。
- 各サブスクライバーは、サブスクリプションリクエストで異なるバッファリング設定を指定できます。
- データを読み取るために必要なバッファサイズを考慮してください。サブスクライブリクエストで設定されている 2*maxBytes+metadata と同じ大きさのペイロード maxBytes を受信することを想定します。例えば、Lambda は次のメタデータバイトを各レコードに追加します。

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- サブスクライバが着信ログを十分な速さで処理できない場合、Lambda はログを削除して、メモリ使用率を制限し続ける可能性があります。削除されたレコードの数を示すために、Lambda は platform.logsDropped ログを送信します。

サブスクリプションの例

次の例は、プラットフォームログと関数ログをサブスクライブするリクエストを示しています。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
```

```
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

リクエストが成功すると、サブスクライバーは HTTP 200 成功レスポンスを受信します。

```
HTTP/1.1 200 OK
"OK"
```

ログ API のサンプルコード

カスタム送信先にログを送信する方法を示すサンプルコードについては、AWS Lambda コンピューティングブログの AWS 拡張機能を使用してログをカスタム送信先に送信するを参照してください。

基本的な Lambda 拡張機能を開発し、Logs API をサブスクライブする方法を示す Python と Go のコード例については、AWSサンプル GitHub リポジトリのAWS Lambda「拡張機能」を参照してください。Lambda 拡張機能の構築 に関する詳細については、the section called “拡張機能 API” を参照してください。

Logs API リファレンス

Logs API エンドポイントの値は、AWS_LAMBDA_RUNTIME_API の環境変数から取得できます。API リクエストを送信するには、API パスの前にプレフィックス 2020-08-15/ を使用します。例:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

Logs API バージョン の OpenAPI 仕様は、.logs-api-requestzip から入手できます。 2020-08-15

Subscribe

Lambda 実行環境で使用できる 1 つ以上のログストリームをサブスクライブするために、拡張機能が Subscribe API リクエストを送信します。

パス – /logs

メソッド – PUT

Body パラメータ

destination「」を参照してください。the section called “送信先プロトコル”必須: はい。タイプ: 文字列。

buffering「」を参照してください。the section called “バッファリング構成”必須: いいえ。タイプ: 文字列。

types - 受信するログのタイプの配列。必須: はい。タイプ: 文字列の配列 有効な値 : 「プラットフォーム」、「関数」、「拡張」。

schemaVersion - 必須: いいえ。デフォルト値: "2020-08-15"。拡張機能が platform.runtimeDone メッセージを受信するには、「2021-03-18」に設定します。

レスポンスパラメータ

サブスクリプションレスポンスの OpenAPI 仕様 (バージョン 2020-08-15) は、HTTP および TCP プロトコルで使用できます。

- HTTP: logs-api-http-response.zip
- TCP: logs-api-tcp-response.zip

レスポンスコード

- 200 - リクエストは正常に完了しました
- 202 - リクエストは承認されました ローカルテスト中のサブスクリプションリクエストへのレスポンス。
- 4XX - 無効なリクエスト
- 500 - サービスエラー

リクエストが成功すると、サブスクライバーは HTTP 200 成功レスポンスを受信します。

```
HTTP/1.1 200 OK
"OK"
```

リクエストが失敗した場合、サブスクライバはエラーレスポンスを受信します。以下に例を示します。

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

ログメッセージ

Logs API を使用すると、拡張機能は 3 つの異なるログストリームにサブスクライブできます。

- 関数 - Lambda 関数が生成し、`stdout` または `stderr` に書き出すログ。
- 拡張機能 - 拡張コードが生成するログ。
- プラットフォーム - ランタイムプラットフォームが生成するログ。呼び出しと拡張機能に関連するイベントおよびエラーを記録します。

トピック

関数ログ

Lambda 関数と内部拡張機能は、関数ログを生成し、`stdout` または `stderr` に書き出します。

次の例は、関数ログメッセージの形式を示しています。 { "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered。Stack trace:\n\my-function (line 10)\n" }

拡張ログ

拡張機能は、拡張ログを生成できます。ログ形式は、関数ログの場合と同じです。

プラットフォームログ

Lambda は、`platform.start`、`platform.end`、`platform.fault` などのプラットフォームイベントのログメッセージを生成します。

必要に応じて、`platform.runtimeDone` ログメッセージを含む Logs API スキーマの 2021-03-18 バージョンをサブスクライブできます。

プラットフォームログメッセージ例

次の例は、プラットフォームの開始ログと終了ログを示しています。これらのログは、requestId で指定された呼び出しの開始時刻と呼び出しの終了時刻を示します。

```
{
```

```
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

プラットフォーム。initRuntimeDone ログメッセージには、初期化ライフサイクルフェーズ の一部であるRuntime initサブフェーズのステータスが表示されます。 ???Runtime init が正常に実行されると、ランタイムが /next Runtime API リクエスト (on-demand および provisioned-concurrency 初期化タイプの場合)、または restore/next (snap-start 初期化タイプの場合) を送信します。次の例は、正常に実行されたプラットフォームを示しています。initRuntimeDone snap-start 初期化タイプの ログメッセージ。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

platform.initReport ログメッセージには、Init フェーズの継続時間と、このフェーズ中に料金が請求されたミリ秒数が表示されます。初期化タイプが provisioned-concurrency の場合、Lambda は呼び出し中にこのメッセージを送信します。初期化タイプが snap-start の場合、Lambda はスナップショットの復元後にこのメッセージを送信します。以下は、snap-start 初期化タイプに関する platform.initReport ログメッセージの例です。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
```

```
  }
}
```

プラットフォームレポートログには、requestId で指定された呼び出しに関するメトリックが含まれます。呼び出しにコールドスタートが含まれている場合にのみ、initDurationMs フィールドがログに含まれます。AWS X-Ray トレースがアクティブである場合、ログには X-Ray メタデータが含まれます。次の例は、コールドスタートを含む呼び出しのプラットフォームレポートログを示しています。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

プラットフォーム障害ログは、ランタイムまたは実行環境エラーをキャプチャします。次の例は、プラットフォーム障害ログメッセージを示しています。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

Lambda は、拡張機能が拡張 API に登録されると、プラットフォーム拡張ログを生成します。次の例は、プラットフォーム拡張メッセージを示しています。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
```

```
     }
}
```

Lambda は、拡張機能がログ API をサブスクライブすると、プラットフォームログサブスクリプションログを生成します。次の例は、ログサブスクリプションメッセージを示しています。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda は、拡張機能が受信しているログの数を処理できない場合に、プラットフォームログをドロップしたログを生成します。次の例は、platform.logsDropped ログメッセージの例を示しています。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

platform.restoreStart ログメッセージには、Restore フェーズが開始された時刻が表示されます (snap-start 初期化タイプのみ)。例：

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

platform.restoreReport ログメッセージには、Restore フェーズの継続時間と、このフェーズ中に料金が請求されたミリ秒数が表示されます (snap-start 初期化タイプのみ)。例：

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

プラットフォーム **runtimeDone** メッセージ

サブスクライブリクエストでスキーマバージョンを「2021-03-18」に設定した場合、Lambda は関数の呼び出しが正常に完了するか、エラーで終了した後に platform.runtimeDone メッセージを送信します。拡張機能は、このメッセージを使用して、この関数呼び出しのすべてのテレメトリ収集を停止できます。

スキーマバージョン 2021-03-18 での Log イベントタイプの OpenAPI 仕様は、schema-2021-03-18.zip から入手できます。

Lambda は、ランタイムが Next または Error ランタイム API リクエストを送信したときに platform.runtimeDone ログメッセージを生成します。platform.runtimeDone ログは、関数の呼び出しが完了したことを Logs API のコンシューマーに通知します。拡張機能は、この情報を使用して、呼び出し中に収集されたすべてのテレメトリをいつ送信するかを決定できます。

例

Lambda は、関数の呼び出しが完了したときに、ランタイムが NEXT リクエストを送信した後に platform.runtimeDone メッセージを送信します。次の例は、各ステータス値 (成功、失敗、タイムアウト) のメッセージを示しています。

Example 成功した場合のメッセージ例

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
```

```
}
```

Example 失敗した場合のメッセージ例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example タイムアウトした場合のメッセージ例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example platform.restoreRuntimeDone message の例 (**`snap-start`** 初期化タイプのみ)

プラットフォーム。restoreRuntimeDone ログメッセージには、 `Restore`フェーズが成功したかどうかが表示されます。Lambda は、ランタイムが `restore/next` Runtime API リクエストを送信するときに、このメッセージを送信します。可能なステータスには、success (成功)、failure (失敗)、および timeout (タイムアウト) の 3 つがあります。次の例は、正常なプラットフォームを示しています。restoreRuntimeDone ログメッセージ。

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Lambda における問題のトラブルシューティング

以下のトピックでは、Lambda API、コンソール、ツールの使用時に発生する可能性のあるエラーや問題のトラブルシューティングに関するアドバイスを提供します。ここに記載されていない問題が見つかった場合は、このページの [Feedback] ボタンを使用して報告することができます。

トラブルシューティングに関するアドバイス、およびサポートへの一般的な質問に対する回答については、AWS ナレッジセンターにアクセスしてください。

Lambda アプリケーションのデバッグとトラブルシューティングの詳細については、Serverless Land の「デバッグ」を参照してください。

トピック

- Lambda におけるデプロイメントに関する問題のトラブルシューティング
- Lambda での呼び出しに関する問題のトラブルシューティング
- Lambda における実行に関する問題点のトラブルシューティング
- Lambda でのネットワークに関する問題のトラブルシューティング

Lambda におけるデプロイメントに関する問題のトラブルシューティング

関数を更新すると、Lambda は、関数の新しいインスタンスを起動してコードや設定の更新を反映することで、その変更をデプロイします。デプロイエラーが発生すると、新しいバージョンは使用できなくなります。デプロイエラーは、デプロイパッケージ、コード、アクセス許可、またはツールに関する問題が原因で発生する場合があります。

Lambda API を使用するか、AWS CLI などのクライアントを使用して関数に更新を直接デプロイした場合、エラーは Lambda の出力に直接表示されます。AWS CloudFormation、AWS CodeDeploy、AWS CodePipeline などのサービスを使用した場合は、該当サービスのログまたはイベントストリームで Lambda からのレスポンスを探します。

以下のトピックでは、Lambda API、コンソール、ツールの使用時に発生する可能性のあるエラーや問題のトラブルシューティングに関するアドバイスを提供します。ここに記載されていない問題が見つかった場合は、このページの [Feedback] ボタンを使用して報告することができます。

トラブルシューティングに関するアドバイス、およびサポートへの一般的な質問に対する回答については、AWS ナレッジセンターにアクセスしてください。

Lambda アプリケーションのデバッグとトラブルシューティングの詳細については、Serverless Land の「デバッグ」を参照してください。

トピック

一般: アクセス権限が拒否されました/該当のファイルをロードできません

エラー: EACCES: アクセス許可が拒否されました。'/var/task/index.js' を開きます。

エラー: そのようなファイルはロードできません -- 関数

エラー: [Errno 13] アクセス許可が拒否されました: '/var/task/function.py'

Lambda ランタイムには、デプロイパッケージ内のファイルを読み取るアクセス許可が必要です。Linux のアクセス権限の 8 進表記では、Lambda には非実行ファイル用に 644 のアクセス権限 (rw-r--r--) が必要であり、ディレクトリと実行可能ファイル用に 755 のアクセス権限 (rwxr-xr-x) が必要です。

Linux と MacOS で、デプロイパッケージ内のファイルやディレクトリのファイルアクセス権限を変更するには、chmod コマンドを使用します。例えば、実行可能ファイルに正しいアクセス許可を付与するには、次のコマンドを実行します。

```
chmod 755 <filepath>
```

Windows でファイルアクセス許可を変更するには、「Microsoft Windows ドキュメント」の「Set, View, Change, or Remove Permissions on an Object」を参照してください。

一般: UpdateFunctionCode を呼び出すときにエラーが発生しました

エラー: UpdateFunctionCode オペレーションを呼び出すときにエラーが発生しました (RequestEntityTooLargeException)

デプロイパッケージまたはレイヤーアーカイブを Lambda に直接アップロードする場合、ZIP ファイルのサイズは 50 MB に制限されます。これよりも大きなファイルをアップロードするには、Amazon S3 に保存し、S3Bucket と S3Key のパラメータを使用します。

> **Note**
>
> AWS CLI、AWS SDK などを使用してファイルを直接アップロードすると、バイナリ ZIP ファイルは base64 に変換され、サイズが約 30% 増加します。この点と、リクエスト内の他のパラメータのサイズを考慮して、Lambda で実際に適用されるリクエストサイズの制限はより大きくなります。このため、50 MB の制限は概算です。

Amazon S3: エラーコード PermanentRedirect。

エラー: GetObject 中にエラーが発生しました。S3 エラーコード: PermanentRedirect。S3 エラーメッセージ: バケットはリージョン us-east-2 にあります。このリージョンを使用してリクエストを再試行してください

Amazon S3 バケットから関数のデプロイパッケージをアップロードする場合、そのバケットは関数と同じリージョンに存在する必要があります。この問題は、UpdateFunctionCode の呼び出しで Amazon S3 オブジェクトを指定するか、AWS CLI または AWS SAM CLI でパッケージとデプロイコマンドを使用する場合に発生する可能性があります。アプリケーションを開発するリージョンごとにデプロイアーティファクトバケットを作成してください。

一般: 見つかりません、ロードできません、インポートできません、クラスが見つかりません、該当のファイルまたはディレクトリがありません

エラー: モジュール 'function' が見つかりません

エラー: そのようなファイルはロードできません -- 関数

エラー: モジュール 'function' をインポートできません

エラー: クラスが見つかりません: function.Handler

エラー: fork/exec /var/task/function: そのようなファイルやディレクトリはありません

エラー: アセンブリ 'Function' から型 'Function.Handler' をロードできません。

関数のハンドラー設定のファイルまたはクラスの名前がコードと一致しません。詳細については、次のセクションを参照してください。

一般: 未定義のメソッドハンドラー

エラー: index.handler が定義されていないか、エクスポートされていません

エラー: モジュール 'function' にハンドラー 'handler' がありません

エラー: #<LambdaHandler:0x000055b76ccebf98> の未定義のメソッド 'handler'

エラー: 適切なメソッドの署名の付いた handleRequest という名前のパブリックメソッドがクラス function.Handler で見つかりません

エラー: アセンブリ 'Function' に型 'Function.Handler' のメソッド 'handleRequest' が見つかりません

関数のハンドラー設定のハンドラーメソッドの名前がコードと一致しません。各ランタイムはハンドラーの命名規則を定義します (*filename.methodname* など)。ハンドラーは、関数を呼び出したときにランタイムで実行される、関数のコード内のメソッドです。

一部の言語の場合、Lambda は、ハンドラーメソッドに特定の名前があることを期待するインターフェイスを持つライブラリを提供します。言語別のハンドラーの命名の詳細については、以下のトピックを参照してください。

- Node.js による Lambda 関数の構築
- Python による Lambda 関数の構築
- Ruby による Lambda 関数の構築
- Java による Lambda 関数の構築
- Go による Lambda 関数の構築
- C# による Lambda 関数の構築
- PowerShell による Lambda 関数の構築

Lambda: レイヤー変換が失敗しました

エラー: Lambda レイヤー変換が失敗しました。この問題を解決するための情報については、Lambda ユーザーガイドの「Lambda でのデプロイ問題のトラブルシューティング」ページを参照してください。

Lambda 関数をレイヤーで設定すると、Lambda はそのレイヤーを関数コードでマージします。このプロセスが完了しない場合は、Lambda はこのエラーを返します。このエラーが発生した場合は、次の手順を実行します。

- レイヤーから未使用のファイルをすべて削除する
- レイヤー内のシンボリックリンクをすべて削除する
- 関数のいずれかのレイヤーにあるディレクトリと同じ名前のファイルの名前を変更する

Lambda: InvalidParameterValueException または RequestEntityTooLargeException

エラー: InvalidParameterValueException: 指定した環境変数が 4 KB の上限を超えているため、Lambda は環境変数を設定できませんでした。測定された文字列: {"A1":"uSFeY5cyPiPn7AtnX5BsM..。

エラー: RequestEntityTooLargeException: UpdateFunctionConfiguration オペレーションに対するリクエストは 5120 バイト未満にする必要があります。

関数の設定に保存される変数オブジェクトの最大サイズが 4096 バイトを超えないようにしてください。これには、キー名、値、引用符、カンマ、括弧が含まれます。HTTP リクエストボディの合計サイズも制限されます。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs20.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "DOC-EXAMPLE-BUCKET",
            "KEY": "file.txt"
        }
    },
```

```
    ...
}
```

この例で、オブジェクトは 39 文字です。これを文字列 {"BUCKET":"DOC-EXAMPLE-BUCKET","KEY":"file.txt"} として保存すると (空白を含まない)、39 バイトになります。環境変数の標準 ASCII 文字は、文字ごとに 1 バイトの値となります。拡張 ASCII 文字と Unicode 文字は、文字ごとに 2～4 バイトを使用する場合があります。

Lambda: InvalidParameterValueException

エラー: InvalidParameterValueException: 指定した環境変数に含まれている予約キーは、現在、変更がサポートされていないため、Lambda は環境変数を設定できませんでした。

Lambda は、内部使用のためにいくつかの環境変数キーを予約します。たとえば、AWS_REGION は現在のリージョンを確認するためにランタイムによって使用される変数で、オーバーライドすることはできません。PATH などの他の変数は、ランタイムによって使用されますが、関数設定で拡張できます。詳細なリストについては、「定義されたランタイム環境変数」を参照してください

Lambda: 同時実行とメモリのクォータ

エラー: 関数に指定された ConcurrentExecutions は、アカウントの UnreservedConcurrentExecution を最小値未満に減らします

エラー: 「メモリサイズ」 値は制約を満たすことに失敗しました: メンバーは 3008 以下の値を持つ必要があります

これらのエラーは、アカウントの同時実行数またはメモリのクォータを超えると発生します。新しい AWS アカウントでは、同時実行とメモリのクォータが少なくなっています。同時実行に関連するエラーを解決するには、クォータの引き上げをリクエストできます。メモリクォータの引き上げはリクエストできません。

- 同時実行数: 予約済みまたはプロビジョニングされた同時実行数を使用して関数を作成、あるいは関数ごとの同時実行の要求 (PutFunctionConcurrency) がアカウントの同時実行クォータを超えた場合、エラーが発生します。
- メモリ: 関数に割り当てられたメモリの量がアカウントのメモリクォータを超えた場合、エラーが発生します。

Lambda での呼び出しに関する問題のトラブルシューティング

Lambda 関数を呼び出すと、Lambda はリクエストを検証し、イベントを関数 (非同期呼び出しの場合はイベントキュー) に送信する前にスケーリングキャパシティーをチェックします。呼び出しエラーは、リクエストのパラメータ、イベント構造、関数の設定、ユーザーのアクセス許可、リソースに対するアクセス許可、または制限に関する問題が原因で発生する場合があります。

関数を直接呼び出した場合は、Lambda からのレスポンスに呼び出しエラーが表示されます。イベントソースマッピングまたは別のサービスを通じて関数を非同期的に呼び出した場合は、ログ、デッドレターキュー、または失敗イベントの送信先にエラーが表示されることがあります。エラー処理オプションと再試行の動作は、関数を呼び出した方法とエラーの種類によって異なります。

`Invoke` オペレーションが返す可能性のあるエラータイプのリストについては、呼び出し を参照してください。

IAM: lambda:InvokeFunction は許可されていません

エラー: ユーザー (arn:aws:iam::123456789012:user/developer) によるリソース (my-function) に対する lambda:InvokeFunction の実行は許可されていません

ユーザー、または引き受けるロールには、関数を呼び出すための許可が必要です。この要件は、Lambda 関数、および関数を呼び出す他のコンピューティングリソースにも適用されます。AWS マネージドポリシーである AWSLambdaRole をユーザーに追加するか、ターゲット関数での `lambda:InvokeFunction` アクションを許可するカスタムポリシーを追加します。

> **Note**
>
> IAM アクションの名前 (`lambda:InvokeFunction`) は、`Invoke` Lambda API オペレーションを示しています。

詳細については、「AWS Lambda アクセス許可の管理」を参照してください。

Lambda: 有効なブートストラップ (Runtime.InvalidEntrypoint) が見つかりませんでした

エラー: 有効なブートストラップが見つかりませんでした: [/var/task/bootstrap /opt/bootstrap]

このエラーは通常、デプロイパッケージのルートに `bootstrap` という名前の実行ファイルが含まれていない場合に発生します。例えば、.zip ファイルを使用して `provided.al2023` 関数をデプロイ

する場合、bootstrap ファイルは .zip ファイルのディレクトリ内ではなく、ルートにある必要があります。

Lambda: オペレーションは ResourceConflictException を実行できません

エラー: ResourceConflictException: 今回はオペレーションを実行できません。現在、この関数は次の状態にあります: 保留中

関数の作成時に Virtual Private Cloud (VPC) に接続すると、Lambda が Elastic Network Interface を作成するまでの間、関数は Pending 状態になります。この間は、関数を呼び出したり変更したりすることはできません。関数の作成後に VPC に接続すると、更新が保留中である間に関数を呼び出すことができます。ただし、そのコードや設定を変更することはできません。

詳細については、「Lambda 関数の状態」を参照してください。

Lambda: 関数が Pending のままとなっています

エラー: 関数が数分間 *Pending* 状態で止まっている。

関数が 6 分を超えて Pending 状態で止まっている場合は、以下のいずれかの API オペレーションを呼び出して、ブロックを解除します。

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda は保留中のオペレーションをキャンセルし、関数を Failed 状態に移します。その後、再度更新を試みることができます。

Lambda: 1 つの関数がすべての同時実行を使用しています

問題: 1 つの関数がすべての利用可能な同時実行を使用しているため、他の関数がスロットリングされる。

AWS リージョンで AWS アカウントで使用可能な同時実行をプールに分割するには、予約済みの同時実行を使用します。予約同時実行により、関数は常に割り当てられた同時実行にスケールでき、割り当てられた同時実行を超えることはありません。

一般: 他のアカウントまたはサービスで関数を呼び出すことはできません

問題: 関数を直接呼び出すことはできるが、別のサービスやアカウントから呼び出すと関数が実行されない。

他のサービスやアカウントから関数を呼び出すためのアクセス許可を、関数のリソースベースのポリシーで付与します。別のアカウントから呼び出す場合、そのアカウントのユーザーにも関数を呼び出すためのアクセス権限が必要です。

一般: 関数の呼び出しはループしています

問題: 関数がループ内で連続して呼び出される。

これは、通常、関数が管理するリソースが、この関数をトリガーするのと同じ AWS のサービス内にある場合に発生します。例えば、Amazon Simple Storage Service (Amazon S3)バケットにオブジェクトを保存する関数を作成した場合、このバケットにこの関数を再度呼び出す通知が設定されていることがあります。関数の実行を停止するには、使用可能な同時実行数を 0 に設定することで、以降の呼び出しがすべてスロットリングされます。その後、再帰呼び出しの原因となったコードパスまたは設定エラーを特定します。Lambda は、一部の AWS サービスおよび SDK の再帰ループを自動的に検出して停止します。詳細については、「the section called “再帰ループ検出”」を参照してください。

Lambda: プロビジョニングされた同時実行によるエイリアスルーティング

問題: エイリアスのルーティング中の、プロビジョニングされた同時実行の Spillover Invocations。

Lambda は、単純な確率モデルを使用して 2 つの関数バージョン間でトラフィックを分散します。低いトラフィックレベルでは、各バージョンで設定されたトラフィックの割合と実際の割合の間に大きな差異が生じる場合があります。関数がプロビジョニングされた同時実行を使用する場合、エイリアスルーティングがアクティブである間に、プロビジョニングされた同時実行インスタンスの数を高く設定することで、過剰呼び出しを防ぐことができます。

Lambda: プロビジョニングされた同時実行によるコールドスタートします

問題: プロビジョニングされた同時実行を有効した後に、コールドスタートが発生します。

関数での同時実行の数が、プロビジョニングされた同時実行の設定済みレベル以下の場合、コールドスタートは発生しないはずです。プロビジョニングされた同時実行が正常に動作しているかどうかを確認するには、次の手順を実行します。

- 関数バージョンまたはエイリアスでプロビジョニングされた同時実行が有効になっていることを確認してください。

> **Note**
>
> プロビジョニング済み同時実行数は、未公開バージョンの関数 ($LATEST) では設定可能ではありません。

- トリガーで正しい関数バージョンまたはエイリアスが呼び出されることを確認します。例えば、Amazon API Gateway を使用している場合は、API Gateway が、$LATEST ではなく、プロビジョニングされた同時実行で関数バージョンまたはエイリアスを呼び出すことを確認します。プロビジョニングされた同時実行が使用されていることを確認するには、ProvisionedConcurrencyInvocations Amazon CloudWatch メトリクスを確認します。ゼロ以外の値は、関数が初期化された実行環境で呼び出しを処理していることを示します。
- ProvisionedConcurrencySpilloverInvocations CloudWatch メトリクスをチェックして、関数の同時実行がプロビジョニングされた同時実行の設定済みレベルを超えているかどうかを判別します。ゼロ以外の値は、プロビジョニングされたすべての同時実行が使用中であり、いくつかの呼び出しがコールドスタートで発生したことを示します。
- 呼び出し頻度 (1 秒あたりのリクエスト数) をご確認ください。プロビジョニングされた同時実行を持つ関数の最大レートは、プロビジョニングされた同時実行ごとに 1 秒あたり 10 件のリクエストです。例えば、100 のプロビジョニングされた同時実行で設定された関数は、1 秒あたり 1,000 件のリクエストを処理できます。呼び出し速度が 1 秒あたり 1,000 件のリクエストを超えると、コールドスタートがいくつか発生する可能性があります。

Lambda: 新しいバージョンによるコールドスタート

問題: 関数の新しいバージョンのデプロイ中にコールドスタートが発生します。

関数エイリアスを更新すると、Lambda は、エイリアスで設定された重みに基づいて、プロビジョニングされた同時実行を新しいバージョンに自動的にシフトします。

エラー: KMSDisabledException: 使用されている KMS キーが無効になっているため、Lambda は環境変数を復号できませんでした。関数の KMS キー設定を確認してください。

このエラーは、AWS Key Management Service (AWS KMS) キーが無効になっている場合、またはキーの使用を Lambda に許可する付与が取り消された場合に発生します。許可がない場合は、別のキーを使用するように関数を設定します。その後、カスタムキーを再割り当てして許可を再作成します。

EFS: 関数は EFS ファイルシステムをマウントできませんでした

エラー: EFSMountFailureException: この関数は、アクセスポイント arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd に EFS ファイルシステムをマウントできませんでした。

関数のファイルシステムへのマウントリクエストが拒否されました。関数のアクセス許可を確認し、そのファイルシステムとアクセスポイントが存在していて使用できる状態であることを確認します。

EFS: 関数は EFS ファイルシステムに接続できませんでした

エラー: EFSMountConnectivityException: この関数は、アクセスポイント arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd で Amazon EFS ファイルポイントに接続できませんでした。ネットワーク設定を確認して、もう一度試してください。

関数は、NFS プロトコル (TCP ポート 2049) を使用して関数のファイルシステムへの接続を確立できませんでした。VPC のサブネットのセキュリティグループとルーティング設定を確認します。

関数の VPC 設定を更新した後にこれらのエラーが発生した場合は、ファイルシステムのアンマウントと再マウントを試してください。

EFS: タイムアウトのため、関数が EFS ファイルシステムをマウントできませんでした

エラー: EFSMountTimeoutException: この関数は、マウントのタイムアウトのため、アクセスポイント {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} に EFS ファイルシステムをマウントできませんでした。

関数は、そのファイルシステムに接続できましたが、マウントオペレーションがタイムアウトしました。しばらくしてから再試行し、関数の同時実行数を制限して、ファイルシステムの負荷を軽減することを検討してください。

Lambda: Lambda は時間がかかり過ぎている IO プロセスを検出しました

EFSIOException: 時間がかかりすぎている IO プロセスを Lambda が検出したため、この関数インスタンスは停止されました。

以前の呼び出しがタイムアウトし、Lambda が関数ハンドラーを終了できませんでした。この問題は、接続されたファイルシステムがバーストクレジットを使い果たし、ベースラインスループットが

不十分な場合に発生することがあります。スループットを高めるには、ファイルシステムのサイズを増やすか、プロビジョニングされたスループットを使用します。詳細については、「スループット」を参照してください。

Lambda における実行に関する問題点のトラブルシューティング

Lambda ランタイムが関数コードを実行すると、このイベントは、すでに他のイベントを処理中の関数のインスタンスで処理されるか、必要に応じて新しいインスタンスが初期化されます。関数が初期化されている間に、ハンドラーコードでイベントが処理されているときに、または関数からレスポンスが返される (または返されない) ときに、エラーが発生する場合があります。

関数の実行エラーは、コード、関数の設定、ダウンストリームのリソース、またはアクセス許可の問題に起因する場合があります。関数を直接呼び出した場合は、Lambda からのレスポンスに関数エラーが表示されます。イベントソースマッピングまたは別のサービスを通じて関数を非同期的に呼び出した場合は、ログ、配信不能キュー、または障害発生時の送信先にエラーが表示されることがあります。エラー処理オプションと再試行の動作は、関数を呼び出した方法とエラーの種類によって異なります。

関数コードまたは Lambda ランタイムからエラーが返された場合、Lambda からのレスポンスのステータスコードは 200 OK です。レスポンス内にエラーが存在することは、`X-Amz-Function-Error` という名前のヘッダーで示されます。400 および 500 シリーズのステータスコードは、呼び出しエラー用に予約されています。

Lambda: 実行に時間がかかり過ぎています

問題: 関数の実行に時間がかかり過ぎる。

Lambda でコードを実行すると、ローカルマシンよりもはるかに長い時間がかかる場合は、その関数で使用できるメモリまたは処理能力が制限されている可能性があります。メモリと CPU の両方を増やすには、メモリを追加して関数を設定します。

Lambda: ログやトレースが表示されません

問題: ログが CloudWatch Logs に表示されない。

問題: トレースが AWS X-Ray に表示されない。

関数には、CloudWatch Logs および X-Ray を呼び出すためのアクセス許可が必要です。その実行ロールを更新してアクセス許可を付与します。ログと追跡を有効にするには、以下の管理ポリシーを追加します。

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

関数にアクセス許可を追加する場合は、そのコードや設定にも些細な更新を行います。これにより、(古い認証情報により実行中の) 関数のインスタンスが、強制的に停止され置き換えられます。

> **Note**
>
> 関数の呼び出し後にログが表示されるまで、5 ~ 10 分かかることがあります。

Lambda: 関数のログの一部が表示されない

問題: 適切な権限があっても、CloudWatch Logs に関数ログが見つからない

AWS アカウント が CloudWatch Logs のクォータ制限に達すると、CloudWatch は関数のロギングを抑制します。この場合、関数によって出力されたログの一部が CloudWatch Logs に表示されない場合があります。

関数がログを出力する頻度が高すぎて Lambda で処理できない場合、ログ出力が CloudWatch Logs に表示されない可能性もあります。Lambda は、関数がログを生成する速度でログを CloudWatch に送信できない場合、関数の実行が遅くなるのを防ぐためにログをドロップします。

関数が JSON 形式のログを使用するように設定されている場合、Lambda はログを削除するとき CloudWatch Logs に `logsDropped` イベントを送信しようとします。ただし、CloudWatch は、関数のログから受け入れることができるデータ量を調整していると、ログを受け取れない場合があります。そのため、Lambda がログを削除したときにレコードが常に表示されるとは限りません。

AWS アカウント が CloudWatch Logs のクォータ制限に達しているかどうかを確認するには、次の手順を実行します。

1. Service Quotas コンソール を開きます。
2. ナビゲーションペインで、[AWS services] (AWS のサービス) を選択します。
3. [AWS サービス] リストから、Amazon CloudWatch Logs を検索します。
4. [Service Quotas] リストで、`CreateLogGroup throttle limit in transactions per second`、`CreateLogStream throttle limit in transactions per second` および `PutLogEvents throttle limit in transactions per second` クォータを選択して使用状況を表示します。

また、アカウントの使用率がこれらのクォータに指定した制限を超えたときに警告するように CloudWatch アラームを設定することもできます。詳しくは、「Create a CloudWatch alarm based on a static threshold」をご覧ください。

CloudWatch Logs のデフォルトのクォータ制限がユースケースに十分ではない場合は、クォータの引き上げをリクエストできます。

Lambda: 関数は実行が終了する前に返します

問題: (Node.js) コードの実行が終了する前に関数が戻る

AWS SDK を含む多くのライブラリは、非同期的に動作します。レスポンスを待つ必要があるネットワーク呼び出しや別のオペレーションを実行すると、ライブラリは、オペレーションの進行状況をバックグラウンドで追跡するプロミスと呼ばれるオブジェクトを返します。

プロミスがレスポンスに解決されるまで待つには、`await` キーワードを使用します。これにより、プロミスがレスポンスを含むオブジェクトに解決されるまで、ハンドラーコードの実行がブロックされます。レスポンス内のデータをコードで使用する必要がない場合は、プロミスをランタイムに直接返すことができます。

一部のライブラリはプロミスを返しませんが、これらはプロミスを返すコードでラップできます。詳細については、「Node.js の Lambda 関数ハンドラーの定義」を参照してください。

AWS SDK: バージョンと更新

問題: ランタイムに含まれている AWS SDK が最新バージョンではない

問題: ランタイムに含まれている AWS SDK が自動的に更新される

スクリプト言語のランタイムには AWS SDK が含まれており、定期的に最新バージョンに更新されます。各ランタイムの現行バージョンは、ランタイムページに表示されます。より新しいバージョンの AWS SDK を使用したり、関数を特定のバージョンにロックしたりするには、ライブラリを関数コードでバンドルするか、Lambda レイヤーを作成します。依存関係を持つデプロイパッケージの作成の詳細については、以下のトピックを参照してください。

Node.js

- .zip ファイルアーカイブで Node.js Lambda 関数をデプロイする

Python

- Python Lambda 関数で .zip ファイルアーカイブを使用する

Ruby

- Ruby Lambda 関数で .zip ファイルアーカイブを使用する

Java

- .zip または JAR ファイルアーカイブで Java Lambda 関数をデプロイする

Go

- .zip ファイルアーカイブを使用して Go Lambda 関数をデプロイする

C#

- .zip ファイルアーカイブを使用して C# Lambda 関数を構築し、デプロイする

PowerShell

- .zip ファイルアーカイブを使用して PowerShell Lambda 関数をデプロイする

Python: ライブラリが正しくロードされません

問題: (Python) 一部のライブラリがデプロイパッケージから正しくロードされない

C または C++ で記述された拡張モジュールを持つライブラリは、Lambda (Amazon Linux) と同じプロセッサアーキテクチャの環境でコンパイルする必要があります。詳細については、「Python Lambda 関数で .zip ファイルアーカイブを使用する」を参照してください。

Lambda でのネットワークに関する問題のトラブルシューティング

デフォルトでは、Lambda は AWS のサービスとインターネットに接続された内部の Virtual Private Cloud (VPC) で関数を実行します。ローカルネットワークのリソースにアクセスするには、アカウントで VPC に接続するように関数を設定できます。この機能を使用するときは、ユーザーが関数のインターネットアクセスと Amazon Virtual Private Cloud (Amazon VPC) リソースとのネットワーク接続を管理します。

ネットワーク接続エラーは、VPC のルーティング設定、セキュリティグループルール、AWS Identity and Access Management (IAM) ロールの許可、NAT、または IP アドレスやネットワークインターフェイスなどのリソースの可用性に関する問題が原因で発生する場合があります。問題によっては、リクエストが送信先に到達できない場合に、特定のエラーやタイムアウトが表示されることがあります。

VPC: 関数がインターネットアクセスを失う、またはタイムアウトする

問題: Lambda 関数が VPC に接続した後でインターネットにアクセスできなくなります。

エラー: エラー: 接続 ETIMEDOUT 176.32.98.189:443

エラー: エラー: タスクが 10.00 秒後にタイムアウトしました

エラー:ReadTimeoutError: Read timed out. (read timeout=15) (ReadTimeoutError: 読み取りがタイムアウトしました。(読み取りタイムアウト=15))

関数を VPC に接続すると、すべてのアウトバウンドリクエストが VPC を通過します。インターネットに接続するには、関数のサブネットからパブリックサブネットの NAT ゲートウェイにアウトバウンドトラフィックを送信するように VPC を設定します。VPC 設定の詳細と例については、「the section called “VPC 関数のインターネットアクセス”」を参照してください。

一部の TCP 接続がタイムアウトしている場合は、パケットフラグメンテーションが原因である可能性があります。Lambda は TCP または ICMP の IP フラグメンテーションをサポートしないため、Lambda 関数は受信するフラグメント化された TCP リクエストを処理できません。

VPC: インターネットを使用せずに関数から AWS のサービスにアクセスする必要がある

問題: Lambda 関数がインターネットを使用せずに AWS のサービスにアクセスする必要があります。

インターネットアクセスがないプライベートサブネットから AWS のサービスに接続するには、VPC エンドポイントを使用します。

VPC: Elastic Network Interface の制限に到達した

エラー: ENILimitReachedException: 関数の VPC で Elastic Network Interface の上限数に達しました。

Lambda 関数を VPC に接続すると、Lambda が、その関数にアタッチされたサブネットとセキュリティグループの組み合わせごとに Elastic Network Interface を作成します。デフォルトのサービスクォータは、VPC あたり 250 個のネットワークインターフェイスです。クォータの引き上げをリクエストするには、Service Quotas コンソールを使用してください。

EC2:「lambda」タイプの Elastic Network Interface

エラーコード: Client.OperationNotPermitted

エラーメッセージ: このタイプのインターフェイスではセキュリティグループを変更できません

Lambda によって管理されている Elastic network interface (ENI) を変更しようとすると、このエラーが表示されます。`ModifyNetworkInterfaceAttribute` は、Lambda が作成した Elastic network interface で更新オペレーションを行うための Lambda API には含まれていません。

AWS Lambda アプリケーション

AWS Lambda アプリケーションは、Lambda 関数、イベントソース、その他のリソースを組み合わせたもので、協調して動作することによりタスクを実行します。AWS CloudFormation および他のツールを使用すると、アプリケーションのコンポーネントを単一パッケージに収集して、1 つのリソースとしてデプロイし管理できます。アプリケーションは Lambda プロジェクトを移植可能にし、AWS CodePipeline、AWS CodeBuild、および AWS Serverless Application Model コマンドラインインターフェイス (AWS SAM CLI) などの追加のデベロッパーツールと統合できるようにします。

AWS Serverless Application Repositoryでは、数クリックでアカウントにデプロイできる Lambda アプリケーションのコレクションを提供しています。リポジトリには、独自のプロジェクトの開始点として使用できるアプリケーションとサンプルのいずれも含まれています。また、独自のプロジェクトを含めるように送信することもできます。

AWS CloudFormation では、アプリケーションのリソースを定義するテンプレートを作成し、アプリケーションをスタックとして管理できるようにします。アプリケーションスタックのリソースをより安全に追加または変更できます。更新の一部が失敗した場合、AWS CloudFormation は自動的に前の設定にロールバックします。AWS CloudFormationパラメータを使用すると、同じテンプレートからアプリケーションの複数の環境を作成できます。AWS SAMは、Lambda アプリケーション開発に重点を置いたシンプルな構文で AWS CloudFormation を拡張します。

AWS CLI および AWS SAM CLI は、Lambda アプリケーションスタックを管理するためのコマンドラインツールです。AWS CloudFormation API でアプリケーションスタックを管理するコマンドに加え、AWS CLI はデプロイパッケージのアップロードやテンプレートの更新などのタスクを簡素化する高レベルのコマンドをサポートしています。AWS SAM CLI は、テンプレートの検証、ローカルテスト、CI/CD システムとの統合を含む追加の機能性を提供します。

アプリケーションを作成するときは、CodeCommit または GitHub への AWS CodeStar 接続を使用して、Git リポジトリを作成できます。CodeCommit を使用すると、IAM コンソールでユーザーの SSH キーと HTTP 認証情報を管理できます。CodeConnections は、GitHub アカウントに接続できるようにします。接続の詳細については、デベロッパーツールコンソールのユーザーガイドの接続とはをご参照ください。

Lambda アプリケーションの設計に関する詳細は、Serverless Land の「アプリケーション設計」を参照してください。

トピック

- AWS Lambda コンソールでのアプリケーションの管理

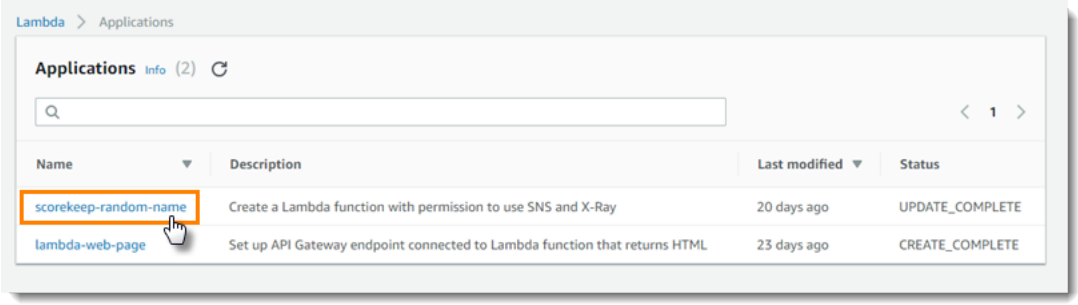
- Lambda 関数のローリングデプロイの作成
- Kubernetes で Lambda を使用する

AWS Lambda コンソールでのアプリケーションの管理

AWS Lambda コンソールは、Lambda アプリケーションのモニタリングと管理に便利です。[Applications] (アプリケーション) メニューには Lambda 関数を使用した AWS CloudFormation スタックが一覧表示されます。このメニューには、AWS CloudFormation コンソール、AWS CloudFormation、AWS Serverless Application Repository や AWS CLI CLI を使用して AWS SAM で起動したスタックが含まれています。

Lambda アプリケーションを表示するには

1. Lambda コンソールの [Applications (アプリケーション)] ページを開きます。
2. アプリケーションを選択します。



概要にはアプリケーションに関する以下の情報が表示されます。

- [AWS CloudFormation template] (CloudFormation テンプレート)または [SAM template] (SAM テンプレート) – アプリケーションを定義するテンプレート
- [Resources] (リソース) – アプリケーションのテンプレートで定義される AWS リソース アプリケーションの Lambda 関数を管理するには、リストから関数名を選択します。

アプリケーションのモニタリング

モニタリングタブには、アプリケーション内のリソースの集計メトリクスを含む Amazon CloudWatch ダッシュボードが表示されます。

Lambda アプリケーションをモニタリングするには

1. Lambda コンソールの [Applications (アプリケーション)] ページを開きます。
2. [モニタリング] を選択します。

デフォルトでは、Lambda コンソールにベーシックなダッシュボードが表示されます。アプリケーションテンプレートでカスタムダッシュボードを定義することで、このページをカスタマイズできます。テンプレートの 1 つ以上のダッシュボードが含まれている場合、このページにはデフォルトのダッシュボードではなく、使用するダッシュボードが表示されます。ページの右上のドロップダウンメニューから、ダッシュボードを切り替えることができます。

カスタムモニタリングダッシュボード

AWS::CloudWatch::Dashboard リソースタイプのアプリケーションテンプレートに 1 つ以上の Amazon CloudWatch ダッシュボードを追加して、アプリケーションモニタリングページをカスタマイズします。次の例では、my-function という名前の関数を呼び出す数をグラフ化する単一のウィジェットのダッシュボードを作成します。

Example 関数ダッシュボードのテンプレート

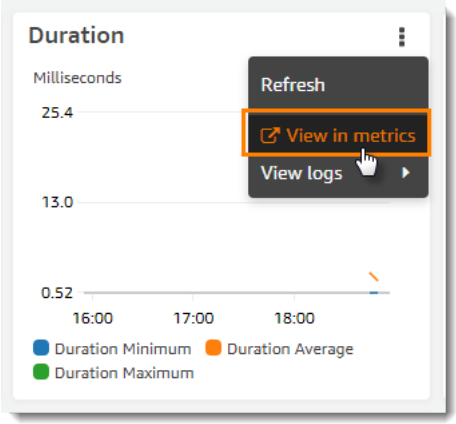
```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
```

```
                                "label": "Total Invocations"
                            }
                        ]
                    ],
                    "region": "us-east-1",
                    "title": "Invocations",
                    "view": "timeSeries",
                    "stacked": false
                }
            }
        ]
    }
```

CloudWatch コンソールのデフォルトのモニタリングダッシュボードから、ウィジェットの定義を取得できます。

ウィジェットの定義を表示するには

1. Lambda コンソールの [Applications (アプリケーション)] ページを開きます。
2. 標準ダッシュボードがあるアプリケーションを選択します。
3. [モニタリング] を選択します。
4. ウィジェットで、ドロップダウンメニューから [メトリクスを表示] を選択します。



5. [Source] を選択します。

CloudWatch ダッシュボードとウィジェットの作成の詳細については、「Amazon CloudWatch API リファレンス」の「ダッシュボード本文の構造と構文」を参照してください。

Lambda 関数のローリングデプロイの作成

ローリングデプロイを使用して、Lambda 関数の新しいバージョンの導入に伴うリスクを制御します。ローリングデプロイでは、システムは関数の新しいバージョンを自動的にデプロイし、徐々に増加するトラフィックを新しいバージョンに送信します。トラフィックの量と増加率は、設定可能なパラメータです。

AWS CodeDeploy および AWS SAM を使用して、ローリングデプロイの設定を行います。CodeDeploy は、Amazon EC2 や AWS Lambda などの Amazon コンピューティングプラットフォームへのアプリケーションデプロイを自動化するサービスです。詳細については、「CodeDeploy とは何ですか?」を参照してください。CodeDeploy を使用して Lambda 関数をデプロイすることにより、簡単にデプロイのステータスをモニタリングし、問題を検出した場合にロールバックを開始できます。

AWS SAM は、サーバーレスアプリケーションを構築するためのオープンソースのフレームワークです。AWS SAM テンプレート（YAML フォーマット）を作成して、ローリングデプロイに必要なコンポーネントの設定を指定できます。AWS SAM はこのテンプレートを使用して、コンポーネントを作成し設定します。詳細については、「AWS SAM とは?」を参照してください。

ローリングデプロイでは、AWS SAM は次のタスクを実行します。

- Lambda 関数を設定し、エイリアスを作成します。

  エイリアスのルーティング設定は、ローリングデプロイを実装する基本的な機能です。
- CodeDeploy アプリケーションとデプロイグループを作成します。

  デプロイグループは、ローリングデプロイとロールバック（必要な場合）を管理します。
- Lambda 関数の新しいバージョンを作成したとき、検出します。
- 新しいバージョンのデプロイを開始するよう CodeDeploy をトリガーします。

サンプル AWS SAM Lambda テンプレート

次の例では、単純なローリングデプロイの AWS SAM テンプレートを示しています。

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions.
```

```
Resources:
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
      Runtime: nodejs18.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

このテンプレートは、次のプロパティを持つ myDateTimeFunction という名前の Lambda 関数を定義します。

AutoPublishAlias

AutoPublishAlias プロパティは、live という名前のエイリアスを作成します。さらに、関数の新しいコードを保存するときに、AWS SAM フレームワークが自動的に検出します。その後、フレームワークは新しい関数バージョンを公開して、新しいバージョンを指すように live エイリアスを更新します。

DeploymentPreference

DeploymentPreference プロパティにより、CodeDeploy アプリケーションが Lambda 関数の元のバージョンから新しいバージョンにトラフィックを移行するレートが決まります。この値 Linear10PercentEvery2Minutes により、トラフィックの追加 10% が 2 分ごとに新しいバージョンにシフトされます。

事前設定されたデプロイ設定の一覧については、「デプロイ設定」を参照してください。

Lambda 関数で CodeDeploy を使用する方法の詳細については、「CodeDeploy を使用して更新された Lambda 関数をデプロイする」を参照してください。

Kubernetes で Lambda を使用する

AWS Controllers for Kubernetes (ACK) または Crossplane を使用して、Kubernetes API で Lambda 関数をデプロイおよび管理できます。

AWS Controllers for Kubernetes (ACK)

ACK を使用すると、Kubernetes API からの AWS リソースのデプロイおよび管理を行うことができます。ACK を通じて、 AWSは Lambda、Amazon Elastic Container Registry (Amazon ECR)、Amazon Simple Storage Service (Amazon S3)、Amazon などのAWSサービス用のオープンソースのカスタムコントローラーを提供します SageMaker。サポートされている各 AWS サービスには、独自のカスタムコントローラーがあります。Kubernetes クラスターに、使用する各 AWS サービス用のコントローラーをインストールします。次に、カスタムリソース定義 (CRD) を作成して、AWS リソースを定義します。

Helm 3.8 またはそれ以降を使用して ACK コントローラーをインストールすることをお勧めします。すべての ACK コントローラーには、コントローラー、CRD、および Kubernetes RBAC ルールをインストールする独自の Helm チャートが付属しています。詳細については、「ACK ドキュメント」の「Install an ACK Controller」を参照してください。

ACK カスタムリソースは、作成後、他の組み込み Kubernetes オブジェクトと同様に使用できます。例えば、kubectl などのお好みの Kubernetes ツールチェーンで Lambda 関数をデプロイして管理できます。

ACK を使用して Lambda 関数をプロビジョニングするユースケースの例を次に示します。

- 組織で ロールベースアクセス制御 (RBAC) およびサービスアカウントの IAM ロールを使用して、アクセス許可の境界を作成する。ACK を使用すると、新しいユーザーやポリシーを作成しなくても、このセキュリティモデルを Lambda に再利用できます。
- 組織には、Kubernetes マニフェストを使用して Amazon Elastic Kubernetes Service (Amazon EKS) クラスターにリソースをデプロイする DevOps プロセスがあります。ACK を使用すると、コードテンプレートとして個別のインフラストラクチャを作成しなくても、マニフェストを使用して Lambda 関数をプロビジョニングできます。

ACK の使用に関する詳細については、ACK ドキュメントの Lambda チュートリアルを参照してください。

Crossplane

Crossplane は、Kubernetes を使用してクラウドインフラストラクチャリソースを管理する、クラウドネイティブコンピューティング財団 (CNCF) のオープンソースのプロジェクトです。Crossplane を使用すると、デベロッパーはインフラストラクチャの複雑さを理解しなくてもインフラストラクチャをリクエストできます。プラットフォームチームは、インフラストラクチャのプロビジョニングおよび管理方法を制御できます。

Crossplane を使用すると、Kubernetes にマニフェストをデプロイできる任意の CI/CD パイプラインや kubectl のようなお好みの Kubernetes ツールチェーンで Lambda 関数をデプロイして管理できます。Crossplane を使用して Lambda 関数をプロビジョニングするユースケースの例を次に示します。

- 組織が、Lambda 関数に正しいタグがあることを確認することでコンプライアンスを強化したいと考えている。プラットフォームチームは、Crossplane Compositions を使用し、API 抽象化を通じてこのポリシーを定義できます。その後、デベロッパーはこれらの抽象化を使用してタグ付きの Lambda 関数をデプロイできます。
- プロジェクトでは、Kubernetes GitOps で を使用します。このモデルでは、Kubernetes は git リポジトリ (望ましい状態) とクラスター内で実行されているリソース (現在の状態) を継続的に照合します。違いがある場合、 GitOps プロセスは自動的にクラスターを変更します。Kubernetes GitOps で を使用すると、CRDs や Controllers などの使い慣れた Kubernetes ツールと概念を使用します。

Lambda で Crossplane を使用する方法の詳細については、次を参照してください。

- AWS Blueprints for Crossplane: このリポジトリには、Crossplane を使用して Lambda 関数などの AWS リソースをデプロイする方法の例が含まれています。

  > **Note**
  >
  > AWS Blueprints for Crossplane は現在開発中であり、本番環境では使用することはできません。

- Deploying Lambda with Amazon EKS and Crossplane: この動画では、Crossplane を使用した AWS サーバーレスアーキテクチャの高度なデプロイの例が紹介されており、デベロッパーとプラットフォームの両方の観点から設計が検討されています。

Lambda サンプルアプリケーション

このガイドの GitHub リポジトリには、さまざまな言語と AWS のサービスの使用を示すサンプルアプリケーションが含まれています。各サンプルアプリケーションには、簡易のデプロイとクリーンアップ用のスクリプト、AWS SAM テンプレート、サポートリソースが含まれています。

Node.js

Node.js のサンプル Lambda アプリケーション

- blank-nodejs – ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Node.js 関数。
- nodejs-apig – API Gateway からのイベントを処理し、HTTP レスポンスを返す公開 API エンドポイントを持つ関数。
- efs-nodejs – Amazon VPC で Amazon EFS ファイルシステムを使用する関数。このサンプルには、Lambda で使用するように設定された VPC、ファイルシステム、マウントターゲット、アクセスポイントが含まれます。

Python

Python のサンプル Lambda アプリケーション

- blank-python – ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Python 関数。

Ruby

Ruby のサンプル Lambda アプリケーション

- blank-ruby - ログ記録、環境変数、AWS X-Ray トレース、レイヤー、単位テスト、AWS SDK の使用を示す Ruby 関数。
- AWS Lambda の Ruby コードサンプル – AWS Lambda との対話方法を示す、Ruby で記述されたコードサンプル。

Java

Java のサンプル Lambda アプリケーション

- [java17-examples] — Java レコードを使用して入力イベントデータオブジェクトを表現する方法を示す Java 関数。
- java-basic - 単位テストと変数ログ記録設定を使用する、最小限の Java 関数のコレクション。
- java-events - Amazon API Gateway、Amazon SQS、Amazon Kinesis などのさまざまなサービスからのイベントを処理する方法のスケルトンコードを含む Java 関数のコレクション。これらの関数は、最新バージョンの aws-lambda-java-events ライブラリ (3.0.0 以降) を使用します。これらの例では、依存関係としての AWS SDK が不要です。
- s3-java - Amazon S3 からの通知イベントを処理し、Java Class Library (JCL) を使用して、アップロードされたイメージファイルからサムネイルを作成する Java 関数。
- API Gateway を使用して Lambda 関数を呼び出す - 従業員情報を含む Amazon DynamoDB テーブルをスキャンする Java 関数。次に、Amazon Simple Notification Service を使用して、仕事の記念日を祝うテキストメッセージを従業員に送信します。この例では、API Gateway ゲートウェイを使用して関数を呼び出します。

Lambda で一般的な Java フレームワークを実行する

- spring-cloud-function-samples — Spring クラウド関数フレームワークを使用して AWS Lambda 関数を作成する方法を示す Spring の例。
- サーバーレス Spring Boot アプリケーションのデモ — 一般的なSpring Bootアプリケーションを、SnapStart を使用または使用しないマネージド Java ランタイムでセットアップする方法、またはカスタムランタイムを使用した GraalVM ネイティブイメージとしてセットアップする方法を示す例。
- サーバーレス Micronaut アプリケーションのデモ — SnapStart を使用または使用しないマネージド Java ランタイムで Micronaut を使用する方法、またはカスタムランタイムを使用した GraalVM ネイティブイメージとして Micronaut を使用する方法を示す例。詳細については、「Micronaut/Lambda guides」を参照してください。
- サーバーレス Quarkus アプリケーションのデモ — SnapStart を使用または使用しないマネージド Java ランタイムで Quarkus を使用する方法、またはカスタムランタイムを使用した GraalVM ネイティブイメージとして Quarkus を使用する方法を示す例。詳細については、「Quarkus/Lambda guide」および「Quarkus/SnapStart guide」を参照してください。

Go

Lambda は、Go ランタイム用の次のサンプルアプリケーションを提供します。

Go のサンプル Lambda アプリケーション

- go-al2 — パブリック IP アドレスを返す Hello World 関数。このアプリは `provided.al2` カスタムランタイムを使用しています。
- blank-go – Lambda の Go ライブラリ、ログ記録、環境変数、AWS SDK の使用を示す Go 関数。このアプリは `go1.x` ランタイムを使用しています。

C#

C# のサンプル Lambda アプリケーション

- blank-csharp – Lambda の .NET ライブラリ、ログ記録、環境変数、AWS X-Ray トレース、単位テスト、AWS SDK の使用を示す C# 関数。
- blank-csharp-with-layer – C# 関数。.NET CLI を使用して、この関数自体の依存関係をパッケージ化するレイヤーを作成します。
- ec2-spot - Amazon EC2 でスポットインスタンスリクエストを管理する関数。

PowerShell

Lambda は、PowerShell 用の次のサンプルアプリケーションを提供します。

- blank-powershell – ログ記録、環境変数、AWS SDK の使用方法を示す PowerShell 関数。

サンプルアプリケーションをデプロイするには、README ファイルの手順に従います。アプリケーションのアーキテクチャとユースケースの詳細については、この章のトピックを参照してください。

トピック

- AWS Lambda の blank 関数サンプルアプリケーション

AWS Lambda の blank 関数サンプルアプリケーション

blank 関数のサンプルアプリケーションは、Lambda API を呼び出す関数を使用し、Lambda の一般的なオペレーションを示すスターター アプリケーションです。ロギング、環境変数、AWS X-Ray トレース、レイヤー、単体テスト、AWS SDK の使用を示します。このアプリケーションについて調べ、お使いのプログラミング言語で Lambda 関数を構築したり、自分のプロジェクトの出発点として使用したりする方法を学びます。

このサンプルアプリケーションのバリアントは、次の言語で使用できます。

バリアント型

- Node.js – blank-nodejs。
- Python – blank-python。
- Ruby – blank-ruby。
- Java – blank-java。
- Go – blank-go。
- C# – blank-csharp。
- PowerShell – blank-powershell。

このトピックの例では、Node.js バージョンのコードが強調されていますが、詳細は通常すべてのバリアントに適用できます。

サンプルは、AWS CLI および AWS CloudFormation を使用して数分でデプロイできます。README の指示に従い、ご自分のアカウントでダウンロード、設定、デプロイしてください。

セクション

- アーキテクチャとハンドラーコード
- AWS CloudFormation および AWS CLI を使用したデプロイの自動化
- AWS X-Ray での計測
- レイヤーを使用した依存関係管理

アーキテクチャとハンドラーコード

サンプルアプリケーションは、関数コード、AWS CloudFormation テンプレート、およびサポートリソースで構成されています。サンプルをデプロイするときは、次の AWS サービスを使用します。

- AWS Lambda - 関数コードの実行、CloudWatch Logs へのログの送信、X-Ray へのトレースデータの送信を行います。また、この関数は Lambda API を呼び出して、現在のリージョンでのアカウントのクォータと使用状況に関する詳細を取得します。
- AWS X-Ray - トレースデータの収集、トレースのインデックス作成 (検索用)、サービスマップの生成を行います。
- Amazon CloudWatch — ログとメトリックを格保存します。
- AWS Identity and Access Management(IAM) — アクセス許可を付与します。
- Amazon Simple Storage Service (Amazon S3) — デプロイ時に関数のデプロイパッケージを保存します。
- AWS CloudFormation - アプリケーションリソースを作成し、関数コードをデプロイします。

各サービスには標準料金が適用されます。詳細については、AWS 料金を参照してください。

関数コードは、イベントを処理するための基本的なワークフローを示しています。ハンドラーは、入力として Amazon Simple Queue Service (Amazon SQS) イベントを受け取り、そのイベントに含まれるレコードを反復処理して、各メッセージの内容をログ記録します。これは、イベントの内容、コンテキストオブジェクト、環境変数をログ記録します。次に、AWS SDK を呼び出し、レスポンスを Lambda ランタイムに渡します。

Example blank-nodejs/function/index.js - ハンドラーコード

```
// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
        console.log(record.body);
    });

    console.log('## ENVIRONMENT VARIABLES: ' + serialize(process.env));
    console.log('## CONTEXT: ' + serialize(context));
    console.log('## EVENT: ' + serialize(event));

    return getAccountSettings();
};

// Use SDK client
var getAccountSettings = function() {
    return lambda.send(new GetAccountSettingsCommand());
};
```

```
var serialize = function(object) {
    return JSON.stringify(object, null, 2);
};
```

サンプルアプリケーションには、イベントを送信するAmazon SQS キューは含まれていませんが、Amazon SQS (event.json) からのイベントを使用して、イベントの処理方法が示されています。アプリケーションに Amazon SQS キューを追加する方法については、Amazon SQS での Lambda の使用 を参照してください。

AWS CloudFormation および AWS CLI を使用したデプロイの自動化

サンプルアプリケーションのリソースは AWS CloudFormation テンプレートで定義され、AWS CLI を使用してデプロイされます。プロジェクトには、アプリケーションのセットアップ、デプロイ、呼び出し、および解体のプロセスを自動化する簡単なシェルスクリプトが含まれています。

アプリケーションテンプレートでは、AWS Serverless Application Model (AWS SAM) リソースタイプを使用してモデルを定義します。AWS SAM は、実行ロール、API、およびその他のリソースの定義を自動化することで、サーバーレスアプリケーションのテンプレート作成を簡素化します。

テンプレートは、アプリケーションスタック内のリソースを定義します。これには、関数およびその実行ロールのほか、関数のライブラリの依存関係を提供する Lambda レイヤーが含まれます。スタックには、AWS CLI がデプロイ中に使用するバケットまたは CloudWatch Logs グループは含まれません。

Example blank-nodejs/template.yml - サーバーレスリソース

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs20.x
      CodeUri: function/.
      Description: Call the AWS Lambda API
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
```

```
        - AWSXrayWriteOnlyAccess
      Tracing: Active
      Layers:
        - !Ref libs
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs20.x
```

アプリケーションをデプロイすると、AWS CloudFormation は AWS SAM 変換をテンプレートに適用し、AWS CloudFormation や `AWS::Lambda::Function` などの標準タイプの `AWS::IAM::Role` テンプレートを生成します。

Example 処理済みテンプレート

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "An AWS Lambda application that calls the Lambda API.",
  "Resources": {
    "function": {
      "Type": "AWS::Lambda::Function",
      "Properties": {
        "Layers": [
          {
            "Ref": "libs32xmpl61b2"
          }
        ],
        "TracingConfig": {
          "Mode": "Active"
        },
        "Code": {
          "S3Bucket": "lambda-artifacts-6b000xmpl1e9bf2a",
          "S3Key": "3d3axmpl473d249d039d2d7a37512db3"
        },
        "Description": "Call the AWS Lambda API",
        "Tags": [
          {
            "Value": "SAM",
            "Key": "lambda:createdBy"
```

```
        }
      ],
```

この例では、Code プロパティは Amazon S3 バケット内のオブジェクトを指定します。これは、プロジェクトテンプレートの CodeUri プロパティのローカルパスに対応します。

```
    CodeUri: function/.
```

プロジェクトファイルを Amazon S3 にアップロードするため、デプロイメントスクリプトは AWS CLI にあるコマンドを使用します。cloudformation package コマンドは、テンプレートを前処理して、アーティファクトをアップロードし、ローカルパスを Amazon S3 オブジェクトの場所に置き換えます。cloudformation deploy コマンドは、AWS CloudFormation 変更セット持つ処理済みテンプレートをデプロイします。

Example blank-nodejs/3-deploy.sh – パッケージングとデプロイ

```
#!/bin/bash
set -eo pipefail
ARTIFACT_BUCKET=$(cat bucket-name.txt)
aws cloudformation package --template-file template.yml --s3-bucket $ARTIFACT_BUCKET --
output-template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name blank-nodejs --
capabilities CAPABILITY_NAMED_IAM
```

このスクリプトを初めて実行すると、AWS CloudFormation という名前の blank-nodejs スタックが作成されます。関数コードまたはテンプレートを変更した場合は、関数コードまたはテンプレートを再度実行してスタックを更新できます。

クリーンアップスクリプト (blank-nodejs/5-cleanup.sh) はスタックを削除し、必要に応じてデプロイバケットと関数ログを削除します。

AWS X-Ray での計測

サンプル関数は、AWS X-Ray でトレースするように構成されています。トレースモードをアクティブに設定すると、Lambda は呼び出しのサブセットのタイミング情報を記録し、これを X-Ray に送信します。X-Ray は、このデータを処理してサービスマップを生成し、クライアントノードと 2 つのサービスノードを示します。

最初のサービスノード (AWS::Lambda) は、呼び出しリクエストを検証して関数に送信する Lambda サービスを表します。2 番目のノード AWS::Lambda::Function は、関数自体を表します。

追加の詳細を記録するために、サンプル関数は X-Ray SDK を使用します。関数コードを最小限変更するだけで、X-Ray SDK は AWS SDK を使用した AWS のサービスへの呼び出しに関する詳細を記録します。

Example blank-nodejs/function/index.js – インストルメンテーション

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());
```

AWS SDK クライアントを計測すると、サービスマップにノードが追加され、トレースにさらに詳細が追加されます。この例では、サービスマップに Lambda API を呼び出すサンプル関数が表示され、現在のリージョンでのストレージと同時実行の使用状況に関する詳細を取得しています。

トレースは、関数の初期化、呼び出し、オーバーヘッドのサブセグメントとともに、呼び出しのタイミングの詳細を表示します。呼び出しサブセグメントには、AWS API オペレーションに対する `GetAccountSettings` SDK 呼び出しのサブセグメントが含まれています。

X-Ray SDK やその他のライブラリを関数のデプロイパッケージに含めることも、Lambda レイヤーに個別にデプロイすることもできます。Node.js、Ruby、Python の場合、Lambda ランタイムは実行環境に AWS SDK を含めます。

レイヤーを使用した依存関係管理

ライブラリは、ローカルにインストールして、Lambda にアップロードするデプロイパッケージに含めることができますが、これにはデメリットがあります。ファイルサイズが大きくなるほどデプロイ時間が長くなり、Lambda コンソールで関数コードの変更をテストできなくなる可能性があります。デプロイパッケージを小さく保ち、変更されていない依存関係のアップロードを避けるために、サンプルアプリは Lambda レイヤーを作成し、関数に関連付けます。

Example blank-nodejs/template.yml – 依存関係レイヤー

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs20.x
```

```
      CodeUri: function/.
      Description: Call the AWS Lambda API
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
      Tracing: Active
      Layers:
        - !Ref libs
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs20.x
```

2-build-layer.sh スクリプトは、npm を使用して関数の依存関係をインストールし、Lambda ランタイムに必要な構造を持つフォルダに配置します。

Example 2-build-layer.sh - レイヤーの準備

```
#!/bin/bash
set -eo pipefail
mkdir -p lib/nodejs
rm -rf node_modules lib/nodejs/node_modules
npm install --production
mv node_modules lib/nodejs/
```

サンプルアプリケーションを初めてデプロイすると、AWS CLI によってレイヤーが関数コードとは別にパッケージ化され、両方がデプロイされます。以降のデプロイでは、lib フォルダーの内容が変更された場合にのみレイヤーアーカイブがアップロードされます。

AWS SDK での Lambda を使用する

AWS ソフトウェア開発キット (SDK) は、多くの一般的なプログラミング言語で使用できます。各 SDK には、デベロッパーが好みの言語でアプリケーションを簡単に構築できるようにする API、コード例、およびドキュメントが提供されています。

SDK ドキュメント	コードの例
AWS SDK for C++	AWS SDK for C++ コードの例
AWS CLI	AWS CLI コードの例
AWS SDK for Go	AWS SDK for Go コードの例
AWS SDK for Java	AWS SDK for Java コードの例
AWS SDK for JavaScript	AWS SDK for JavaScript コードの例
AWS SDK for Kotlin	AWS SDK for Kotlin コードの例
AWS SDK for .NET	AWS SDK for .NET コードの例
AWS SDK for PHP	AWS SDK for PHP コードの例
AWS Tools for PowerShell	Tools for PowerShell のコード例
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) コードの例
AWS SDK for Ruby	AWS SDK for Ruby コードの例
AWS SDK for Rust	AWS SDK for Rust コードの例
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP コードの例
AWS SDK for Swift	AWS SDK for Swift コードの例

Lambda に固有の例については、「AWS SDK を使用した Lambda のコード例」を参照してください。

可用性の例

必要なものが見つからなかった場合。このページの下側にある [Provide feedback (フィードバックを送信)] リンクから、コードの例をリクエストしてください。

AWS SDK を使用した Lambda のコード例

以下のコード例は、AWS Software Development Kit (SDK) で Lambda を使用する方法を示しています。

アクションはより大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。アクションは個々のサービス機能を呼び出す方法を示していますが、関連するシナリオやサービス間の例ではアクションのコンテキストが確認できます。

「シナリオ」は、同じサービス内で複数の関数を呼び出して、特定のタスクを実行する方法を示すコード例です。

クロスサービスの例は、複数の AWS のサービス で動作するサンプルアプリケーションです。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

開始方法

Hello Lambda

次のコード例では、Lambda の使用を開始する方法について示しています。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
```

```
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListFunctions」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

CMakeLists.txt CMake ファイルのコード。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")
```

```
# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_lambda.cpp ソースファイルのコード。

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
```

```
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;
```

```
                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
                    std::cout << "   "
                              <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「ListFunctions」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(context.TODO(),
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
```

```
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「ListFunctions」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
package com.example.lambda;

import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.ListFunctionsResponse;
import software.amazon.awssdk.services.lambda.model.FunctionConfiguration;
import java.util.List;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class ListLambdaFunctions {
    public static void main(String[] args) {
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
```

```
                .build();

        listFunctions(awsLambda);
        awsLambda.close();
    }

    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
 config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「ListFunctions」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
```

```
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「ListFunctions」を参照してください。

コードの例

- AWS SDK を使用した Lambda のアクション
  - AWS SDK または CLI で CreateAlias を使用する
  - AWS SDK または CLI で CreateFunction を使用する
  - AWS SDK または CLI で DeleteAlias を使用する
  - AWS SDK または CLI で DeleteFunction を使用する
  - AWS SDK または CLI で DeleteFunctionConcurrency を使用する
  - AWS SDK または CLI で DeleteProvisionedConcurrencyConfig を使用する
  - AWS SDK または CLI で GetAccountSettings を使用する
  - AWS SDK または CLI で GetAlias を使用する
  - AWS SDK または CLI で GetFunction を使用する
  - AWS SDK または CLI で GetFunctionConcurrency を使用する
  - AWS SDK または CLI で GetFunctionConfiguration を使用する
  - AWS SDK または CLI で GetPolicy を使用する
  - AWS SDK または CLI で GetProvisionedConcurrencyConfig を使用する
  - AWS SDK または CLI で Invoke を使用する
  - AWS SDK または CLI で ListFunctions を使用する
  - AWS SDK または CLI で ListProvisionedConcurrencyConfigs を使用する

- 貸出ライブラリ REST API を作成する
- ステップ関数でメッセンジャーアプリケーションを作成する
- ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成
- API Gateway で WebSocket チャットアプリケーションを作成する
- 顧客からのフィードバックを分析し、音声を合成するアプリケーションの作成
- ブラウザからの Lambda 関数の呼び出し
- S3 Object Lambda でアプリケーションのデータを変換する
- API Gateway を使用して Lambda 関数を呼び出す
- Step Functions を使用して Lambda 関数を呼び出す
- スケジュールされたイベントを使用した Lambda 関数の呼び出し

AWS SDK を使用した Lambda のアクション

以下のコード例は、AWS SDK を使用して個々の Lambda アクションを実行する方法を示しています。これらの抜粋は Lambda API を呼び出すもので、コンテキスト内で実行する必要がある大規模なプログラムからのコード抜粋です。それぞれの例には、GitHub へのリンクがあり、そこにはコードの設定と実行に関する説明が記載されています。

以下の例には、最も一般的に使用されるアクションのみ含まれています。詳細な一覧については、「AWS Lambda API リファレンス」を参照してください。

例

- AWS SDK または CLI で CreateAlias を使用する
- AWS SDK または CLI で CreateFunction を使用する
- AWS SDK または CLI で DeleteAlias を使用する
- AWS SDK または CLI で DeleteFunction を使用する
- AWS SDK または CLI で DeleteFunctionConcurrency を使用する
- AWS SDK または CLI で DeleteProvisionedConcurrencyConfig を使用する
- AWS SDK または CLI で GetAccountSettings を使用する
- AWS SDK または CLI で GetAlias を使用する
- AWS SDK または CLI で GetFunction を使用する
- AWS SDK または CLI で GetFunctionConcurrency を使用する
- AWS SDK または CLI で GetFunctionConfiguration を使用する

- AWS SDK または CLI で GetPolicy を使用する
- AWS SDK または CLI で GetProvisionedConcurrencyConfig を使用する
- AWS SDK または CLI で Invoke を使用する
- AWS SDK または CLI で ListFunctions を使用する
- AWS SDK または CLI で ListProvisionedConcurrencyConfigs を使用する
- AWS SDK または CLI で ListTags を使用する
- AWS SDK または CLI で ListVersionsByFunction を使用する
- AWS SDK または CLI で PublishVersion を使用する
- AWS SDK または CLI で PutFunctionConcurrency を使用する
- AWS SDK または CLI で PutProvisionedConcurrencyConfig を使用する
- AWS SDK または CLI で RemovePermission を使用する
- AWS SDK または CLI で TagResource を使用する
- AWS SDK または CLI で UntagResource を使用する
- AWS SDK または CLI で UpdateAlias を使用する
- AWS SDK または CLI で UpdateFunctionCode を使用する
- AWS SDK または CLI で UpdateFunctionConfiguration を使用する

AWS SDK または CLI で **CreateAlias** を使用する

以下のコード例は、CreateAlias の使用方法を示しています。

CLI

AWS CLI

Lambda 関数のエイリアスを作成する方法

次の create-alias の例では、my-function Lambda 関数のバージョン 1 を参照する LIVE という名前のエイリアスを作成します。

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

出力:

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "Description": "alias for live version of function"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「CreateAlias」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、指定されたバージョンおよびルーティング設定の新しい Lambda エイリアスを作成し、受信する呼び出しリクエストの割合を指定します。

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「CreateAlias」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **CreateFunction** を使用する

以下のコード例は、CreateFunction の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// <summary>
/// Creates a new Lambda function.
/// </summary>
/// <param name="functionName">The name of the function.</param>
/// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
/// bucket where the zip file containing the code is located.</param>
/// <param name="s3Key">The Amazon S3 key of the zip file.</param>
/// <param name="role">The Amazon Resource Name (ARN) of a role with the
/// appropriate Lambda permissions.</param>
/// <param name="handler">The name of the handler function.</param>
/// <returns>The Amazon Resource Name (ARN) of the newly created
/// Lambda function.</returns>
public async Task<string> CreateLambdaFunctionAsync(
    string functionName,
    string s3Bucket,
    string s3Key,
    string role,
    string handler)
{
    // Defines the location for the function code.
    // S3Bucket - The S3 bucket where the file containing
    //            the source code is stored.
    // S3Key    - The name of the file containing the code.
    var functionCode = new FunctionCode
    {
        S3Bucket = s3Bucket,
        S3Key = s3Key,
    };

    var createFunctionRequest = new CreateFunctionRequest
```

```
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「CreateFunction」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
```

```
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);
```

```
        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「CreateFunction」を参照してください。

CLI

AWS CLI

Lambda 関数を作成するには

次の `create-function` の例では、`my-function` という名前の Lambda 関数を作成します。

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs18.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

`my-function.zip` の内容:

```
This file is a deployment package that contains your function code and any
 dependencies.
```

出力:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2023-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs18.x",
    "Description": ""
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「CreateFunction」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
 string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
 &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython38,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
```

```
    FunctionName: aws.String(functionName)}, 1*time.Minute)
   if err != nil {
    log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
  functionName, err)
   } else {
    state = funcOutput.Configuration.State
   }
  }
  return state
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「CreateFunction」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.core.waiters.WaiterResponse;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.CreateFunctionRequest;
import software.amazon.awssdk.services.lambda.model.FunctionCode;
import software.amazon.awssdk.services.lambda.model.CreateFunctionResponse;
import software.amazon.awssdk.services.lambda.model.GetFunctionRequest;
import software.amazon.awssdk.services.lambda.model.GetFunctionResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;
import software.amazon.awssdk.services.lambda.model.Runtime;
import software.amazon.awssdk.services.lambda.waiters.LambdaWaiter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
```

```
/**
 * This code example requires a ZIP or JAR that represents the code of the
 * Lambda function.
 * If you do not have a ZIP or JAR, please refer to the following document:
 *
 * https://github.com/aws-doc-sdk-examples/tree/master/javav2/usecases/
creating_workflows_stepfunctions
 *
 * Also, set up your development environment, including your credentials.
 *
 * For information, see this documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */

public class CreateFunction {
    public static void main(String[] args) {

        final String usage = """

                Usage:
                    <functionName> <filePath> <role> <handler>\s

                Where:
                    functionName - The name of the Lambda function.\s
                    filePath - The path to the ZIP or JAR where the code is
 located.\s
                    role - The role ARN that has Lambda permissions.\s
                    handler - The fully qualified method name (for example,
 example.Handler::handleRequest). \s
                """;

        if (args.length != 4) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        String filePath = args[1];
        String role = args[2];
        String handler = args[3];
        Region region = Region.US_WEST_2;
```

```
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        createLambdaFunction(awsLambda, functionName, filePath, role, handler);
        awsLambda.close();
    }

    public static void createLambdaFunction(LambdaClient awsLambda,
            String functionName,
            String filePath,
            String role,
            String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            InputStream is = new FileInputStream(filePath);
            SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

            FunctionCode code = FunctionCode.builder()
                    .zipFile(fileToUpload)
                    .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                    .functionName(functionName)
                    .description("Created by the Lambda Java API")
                    .code(code)
                    .handler(handler)
                    .runtime(Runtime.JAVA8)
                    .role(role)
                    .build();

            // Create a Lambda function using a waiter.
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The function ARN is " +
functionResponse.functionArn());
```

```
        } catch (LambdaException | FileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「CreateFunction」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「CreateFunction」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java8
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
```

```
        return functionResponse.functionArn
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「CreateFunction」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「CreateFunction」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、AWS Lambda で MyFunction という名前の新しい C# (dotnetcore1.0 ランタイム) 関数を作成し、ローカルファイルシステムの zip ファイルから関数のコンパイル済みバイナリを提供します (相対パスまたは絶対パスを使用可能)。C# Lambda 関数は、AssemblyName::Namespace.ClassName::MethodName の指定を使用して関数のハンドラーを指定します。ハンドラー仕様のアセンブリ名 (.dll サフィックスなし)、名前空間、クラス名、メソッド名の部分を適切に置き換える必要があります。新しい関数には、指定された値で「envvar1」および「envvar2」の環境変数が設定されます。

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

出力:

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

例 2: この例は前の例と似ていますが、関数バイナリが最初に Amazon S3 バケット (目的の Lambda 関数と同じリージョンにある必要がある) にアップロードされ、結果の S3 オブジェクトが関数の作成時に参照される点が異なります。

```
Write-S3Object -BucketName mybucket -Key MyFunctionBinaries.zip -File .
\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName mybucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「CreateFunction」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.
```

```
        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.8",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「CreateFunction」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function. This
  #                      must include the file name and the function name.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
  #                            code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: "ruby2.7",
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    "LOG_LEVEL" => "info"
                                                  }
                                                }
```

```
                                                  })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「CreateFunction」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;
```

```
        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
```

```
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「CreateFunction」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example'
        ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
```

```
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「CreateFunction」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DeleteAlias** を使用する

以下のコード例は、DeleteAlias の使用方法を示しています。

CLI

AWS CLI

Lambda 関数のエイリアスを削除する方法

次の delete-alias の例では、my-function Lambda 関数から LIVE という名前のエイリアスを削除します。

```
aws lambda delete-alias \
    --function-name my-function \
```

```
    --name LIVE
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「DeleteAlias」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、コマンドに記述された Lambda 関数のエイリアスを削除します。

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「DeleteAlias」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DeleteFunction** を使用する

以下のコード例は、DeleteFunction の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「DeleteFunction」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
        std::cout << "The lambda function was successfully deleted." <<
std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「DeleteFunction」を参照してください。

CLI

AWS CLI

例 1: Lambda 関数を関数名で削除するには

次の delete-function の例では、関数名を指定して my-function という Lambda 関数を削除します。

```
aws lambda delete-function \
    --function-name my-function
```

このコマンドでは何も出力されません。

例 2: Lambda 関数を関数 ARN で削除するには

次の delete-function の例では、関数 ARN を指定して my-function という Lambda 関数を削除します。

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

このコマンドでは何も出力されません。

例 3: Lambda 関数を関数 ARN の一部で削除するには

次の delete-function の例では、関数 ARN の一部を指定して my-function という Lambda 関数を削除します。

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteFunction」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
 _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
 &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「DeleteFunction」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.DeleteFunctionRequest;
import software.amazon.awssdk.services.lambda.model.LambdaException;

/**
 * Before running this Java V2 code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 */
public class DeleteFunction {
    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <functionName>\s

                Where:
                    functionName - The name of the Lambda function.\s
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        Region region = Region.US_EAST_1;
```

```
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        deleteLambdaFunction(awsLambda, functionName);
        awsLambda.close();
    }

    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「DeleteFunction」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/**
 * @param {string} funcName
```

```
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「DeleteFunction」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「DeleteFunction」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「DeleteFunction」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の特定のバージョンを削除します。

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「DeleteFunction」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」で「DeleteFunction」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print "Done!".green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「DeleteFunction」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       info!("Deleting iam role {}", self.role_name);
       let delete_role = self
           .iam_client
           .delete_role()
           .role_name(self.role_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
           if let Some(location) = location {
               info!("Deleting object {location}");
               Some(
```

```
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「DeleteFunction」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
```

```
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「DeleteFunction」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DeleteFunctionConcurrency** を使用する

以下のコード例は、DeleteFunctionConcurrency の使用方法を示しています。

CLI

AWS CLI

関数から同時実行制限を削除する方法

次の delete-function-concurrency の例では、予約済みの同時実行制限を my-function 関数から削除します。

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパーガイド」の「Lambda 関数の同時実行数を予約する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「DeleteFunctionConcurrency」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の関数同時実行数を削除します。

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「DeleteFunctionConcurrency」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **DeleteProvisionedConcurrencyConfig** を使用する

以下のコード例は、DeleteProvisionedConcurrencyConfig の使用方法を示しています。

CLI

AWS CLI

プロビジョニングされた同時実行数の設定を削除する方法

次の delete-provisioned-concurrency-config の例では、指定した関数の GREEN エイリアスのプロビジョニングされた同時実行設定を削除します。

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- API の詳細については、「AWS CLI コマンド リファレンス」の「DeleteProvisionedConcurrencyConfig」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、特定のエイリアスのプロビジョニングされた同時実行設定を削除します。

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「DeleteProvisionedConcurrencyConfig」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetAccountSettings** を使用する

以下のコード例は、GetAccountSettings の使用方法を示しています。

CLI

AWS CLI

AWS リージョンでアカウントに関する詳細を取得する方法

次の get-account-settings の例では、アカウントの Lambda 制限および使用情報を表示します。

```
aws lambda get-account-settings
```

出力:

```
{
    "AccountLimit": {
       "CodeSizeUnzipped": 262144000,
       "UnreservedConcurrentExecutions": 1000,
       "ConcurrentExecutions": 1000,
       "CodeSizeZipped": 52428800,
       "TotalCodeSize": 80530636800
```

```
    },
    "AccountUsage": {
       "FunctionCount": 4,
       "TotalCodeSize": 9426
    }
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda の制限」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetAccountSettings」を参照してください。

PowerShell

Tools for PowerShell

例 1: このサンプルは、アカウント制限およびアカウント使用量を比較するために表示されます。

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

出力:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetAccountSettings」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetAlias** を使用する

以下のコード例は、GetAlias の使用方法を示しています。

CLI

AWS CLI

関数エイリアスに関する詳細を取得する方法

次の get-alias の例では、my-function Lambda 関数に LIVE という名前のエイリアスの詳細が表示されます。

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

出力:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetAlias」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、特定の Lambda 関数エイリアスのルーティング設定の重点を取得します。

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

出力:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetAlias」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetFunction** を使用する

以下のコード例は、GetFunction の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
```

```
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「GetFunction」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::GetFunctionRequest request;
       request.SetFunctionName(functionName);

       Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

       if (outcome.IsSuccess()) {
           std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
```

```
                    << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「GetFunction」を参照してください。

CLI

AWS CLI

関数に関する情報を取得するには

次の例 `get-function` では、`my-function` 関数の情報が表示されます。

```
aws lambda get-function \
    --function-name  my-function
```

出力:

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
```

```
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
        "LastModified": "2019-09-24T18:20:35.054+0000",
        "Runtime": "nodejs10.x",
        "Description": ""
    }
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetFunction」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(functionName string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「GetFunction」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「GetFunction」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「GetFunction」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
```

```
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「GetFunction」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
```

```
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「GetFunction」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
```

```
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「GetFunction」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_result =  lo_lmd->getfunction( iv_functionname = iv_function_name ).
     " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「GetFunction」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetFunctionConcurrency** を使用する

以下のコード例は、GetFunctionConcurrency の使用方法を示しています。

CLI

AWS CLI

関数の予約済み同時実行設定を表示する方法

次の get-function-concurrency の例では、指定した関数の予約済み同時実行設定を取得します。

```
aws lambda get-function-concurrency \
    --function-name my-function
```

出力:

```
{
    "ReservedConcurrentExecutions": 250
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetFunctionConcurrency」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の予約済み同時実行数が取得されます

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

出力:

```
ReservedConcurrentExecutions
----------------------------
100
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetFunctionConcurrency」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetFunctionConfiguration** を使用する

以下のコード例は、GetFunctionConfiguration の使用方法を示しています。

CLI

AWS CLI

Lambda 関数のバージョン固有設定を取得する方法

次の get-function-configuration の例では、my-function 関数のバージョン 2 の設定が表示されます。

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

出力:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
```

```
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetFunctionConfiguration」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数のバージョン固有設定を返します。

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

出力:

```
CodeSha256                : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                  : 1426
DeadLetterConfig          : Amazon.Lambda.Model.DeadLetterConfig
Description               : Verson 3 to test Aliases
Environment               : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn               : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                            :PowershellAlias
FunctionName              : MylambdaFunction123
Handler                   : lambda_function.launch_instance
KMSKeyArn                 :
LastModified              : 2019-12-25T09:52:59.872+0000
LastUpdateStatus          : Successful
LastUpdateStatusReason    :
LastUpdateStatusReasonCode :
Layers                    : {}
MasterArn                 :
MemorySize                : 128
RevisionId                : 5d7de38b-87f2-4260-8f8a-e87280e10c33
```

```
Role                            : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                         : python3.8
State                           : Active
StateReason                     :
StateReasonCode                 :
Timeout                         : 600
TracingConfig                   : Amazon.Lambda.Model.TracingConfigResponse
Version                         : 4
VpcConfig                       : Amazon.Lambda.Model.VpcConfigDetail
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetFunctionConfiguration」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetPolicy** を使用する

以下のコード例は、GetPolicy の使用方法を示しています。

CLI

AWS CLI

関数、バージョン、エイリアスのリソースベースの IAM ポリシーを取得する方法

次の get-policy の例では、my-function Lambda 関数に関する情報が表示されます。

```
aws lambda get-policy \
    --function-name my-function
```

出力:

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
        "Statement":
        [
```

```
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

詳細については、「AWS Lambda デベロッパー ガイド」の「AWS Lambda のリソースベースのポリシーを使用する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetPolicy」を参照してください。

PowerShell

Tools for PowerShell

例 1: このサンプルは、Lambda 関数の関数ポリシーが表示されます

```
Get-LMPolicy -FunctionName test -Select Policy
```

出力:

```
{"Version":"2012-10-17","Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetPolicy」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **GetProvisionedConcurrencyConfig** を使用する

以下のコード例は、GetProvisionedConcurrencyConfig の使用方法を示しています。

CLI

AWS CLI

プロビジョニングされた同時実行設定を表示する方法

次の get-provisioned-concurrency-config の例では、指定した関数の BLUE エイリアスにプロビジョニングされた同時実行設定の詳細が表示されます。

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

出力:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「GetProvisionedConcurrencyConfig」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の指定されたエイリアスにプロビジョニングされた同時実行設定を取得します。

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

出力:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「GetProvisionedConcurrencyConfig」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **Invoke** を使用する

以下のコード例は、Invoke の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
```

```
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「Invoke」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";
```

```
    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「Invoke」を参照してください。

CLI

AWS CLI

例 1: Lambda 関数を同期的に呼び出すには

次の例 `invoke` では、`my-function` 関数を同期的に呼び出します。AWS CLI バージョン 2 を使用している場合、`cli-binary-format` オプションは必須です。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI でサポートされるグローバルコマンドラインオプション」を参照してください。

```
aws lambda invoke \
```

```
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

出力:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「同期呼び出し」を参照してください。

例 2: Lambda 関数を非同期で呼び出すには

次の例 invoke では、my-function 関数を非同期で呼び出します。AWS CLI バージョン 2 を使用している場合、cli-binary-format オプションは必須です。詳細については、「AWS Command Line Interface ユーザーガイド」の「AWS CLI でサポートされるグローバルコマンドラインオプション」を参照してください。

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

出力:

```
{
    "StatusCode": 202
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「非同期呼び出し」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「Invoke」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog
 bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
 &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
```

```
 }
 return invokeOutput
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「Invoke」を参照してください。

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
import org.json.JSONObject;
import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.services.lambda.model.LambdaException;

public class LambdaInvoke {

    /*
     * Function names appear as
     * arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
     * you can retrieve the value by looking at the function in the AWS Console
     *
     * Also, set up your development environment, including your credentials.
     *
     * For information, see this documentation topic:
     *
     * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.
```

```
     * html
     */

    public static void main(String[] args) {
        final String usage = """

                Usage:
                    <functionName>\s

                Where:
                    functionName - The name of the Lambda function\s
                """;

        if (args.length != 1) {
            System.out.println(usage);
            System.exit(1);
        }

        String functionName = args[0];
        Region region = Region.US_WEST_2;
        LambdaClient awsLambda = LambdaClient.builder()
                .region(region)
                .build();

        invokeFunction(awsLambda, functionName);
        awsLambda.close();
    }

    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {

        InvokeResponse res = null;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            // Setup an InvokeRequest.
            InvokeRequest request = InvokeRequest.builder()
                    .functionName(functionName)
                    .payload(payload)
                    .build();
```

```
            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の「Invoke」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「Invoke」を参照してください。

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の「Invoke」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「Invoke」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
```

```
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「Invoke」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
```

```
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name}
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「Invoke」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   /** Invoke the lambda function using calculator InvokeArgs. */
   pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
```

```
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「Invoke」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
          `{`  &&
            `"action": "increment",`  &&
```

```
            `"number": 10` &&
          `}`
        ).
        oo_result =  lo_lmd->invoke(                  " oo_result is returned for
testing purposes. "
                iv_functionname = iv_function_name
                iv_payload = lv_json
            ).
        MESSAGE 'Lambda function invoked.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvrequestcontex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
      CATCH /aws1/cx_lmdrequesttoolargeex.
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
      CATCH /aws1/cx_lmdunsuppedmediatyp00.
        MESSAGE 'Invoke request body does not have JSON as its content type.'
TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「Invoke」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **ListFunctions** を使用する

以下のコード例は、ListFunctions の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「ListFunctions」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
```

```
                            <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                    functionConfiguration.GetRuntime()) << ": "
                            << functionConfiguration.GetHandler()
                            << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「ListFunctions」を参照してください。

CLI

AWS CLI

Lambda 関数の一覧を取得するには

次の例 `list-functions` では、現在のユーザーのすべての関数を一覧表示します。

```
aws lambda list-functions
```

出力:

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
```

```
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2023-09-23T18:32:33.857+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2023-10-01T16:47:28.490+0000",
            "Runtime": "nodejs18.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
```

```
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2023-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListFunctions」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(context.TODO())
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「ListFunctions」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「ListFunctions」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
```

```
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「ListFunctions」を参照してください。

PowerShell

Tools for PowerShell

例 1: このサンプルでは、すべての Lambda 関数をソートされたコードサイズで表示されます

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

出力:

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「ListFunctions」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「ListFunctions」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response["functions"].each do |function|
        functions.append(function["function_name"])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「ListFunctions」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「ListFunctions」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    TRY.
        oo_result =  lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
        DATA(lt_functions) = oo_result->get_functions( ).
        MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
```

```
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「ListFunctions」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **ListProvisionedConcurrencyConfigs** を使用する

以下のコード例は、ListProvisionedConcurrencyConfigs の使用方法を示しています。

CLI

AWS CLI

プロビジョニングされた同時実行設定のリストを取得する方法

次の list-provisioned-concurrency-configs の例では、指定された関数にプロビジョニングされた同時実行設定がリストされます。

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

出力:

```
{
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
```

```
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListProvisionedConcurrencyConfigs」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数にプロビジョニングされた同時実行設定のリストを取得します。

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「ListProvisionedConcurrencyConfigs」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **ListTags** を使用する

以下のコード例は、ListTags の使用方法を示しています。

CLI

AWS CLI

Lambda 関数のタグのリストを取得する方法

次の `list-tags` の例では、`my-function` Lambda 関数にアタッチされたタグが表示されます。

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

出力:

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「Lambda 関数をタグ付けする」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListTags」を参照してください。

PowerShell

Tools for PowerShell

例 1: 指定した関数に現在設定されているタグとその値を取得します。

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

出力:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
```

```
Washington Olympia
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「ListTags」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **ListVersionsByFunction** を使用する

以下のコード例は、ListVersionsByFunction の使用方法を示しています。

CLI

AWS CLI

関数のバージョンのリストを取得する方法

次の list-versions-by-function の例は、my-function Lambda 関数のバージョンのリストが表示されます。

```
aws lambda list-versions-by-function \
    --function-name my-function
```

出力:

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
```

```
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
```

```
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「ListVersionsByFunction」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の各バージョンのバージョン固有設定に関するリストを返します。

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

出力:

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
```

```
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「ListVersionsByFunction」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **PublishVersion** を使用する

以下のコード例は、PublishVersion の使用方法を示しています。

CLI

AWS CLI

関数の新しいバージョンを発行する方法

次の publish-version の例では、my-function Lambda 関数の新しいバージョンを発行します。

```
aws lambda publish-version \
    --function-name my-function
```

出力:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
```

```
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「PublishVersion」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数コードの既存のスナップショットのバージョンを作成します

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「PublishVersion」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **PutFunctionConcurrency** を使用する

以下のコード例は、PutFunctionConcurrency の使用方法を示しています。

CLI

AWS CLI

関数に予約済み同時実行制限を設定する方法

次の put-function-concurrency の例では、my-function 関数に 100 個の予約済み同時実行数を設定します。

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

出力:

```
{
    "ReservedConcurrentExecutions": 100
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「Lambda 関数の同時実行数を予約する」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutFunctionConcurrency」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、関数全体の同時実行設定を適用します。

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- API の詳細については、「AWS Tools for PowerShell コマンドレットリファレンス」の「PutFunctionConcurrency」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **PutProvisionedConcurrencyConfig** を使用する

以下のコード例は、PutProvisionedConcurrencyConfig の使用方法を示しています。

CLI

AWS CLI

プロビジョニングされた同時実行数を配分する方法

次の put-provisioned-concurrency-config の例では、指定した関数の BLUE エイリアスに 100 個のプロビジョニングされた同時実行数を割り当てます。

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

出力:

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- API の詳細については、「AWS CLI コマンドリファレンス」の「PutProvisionedConcurrencyConfig」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、プロビジョニングされた同時実行設定を関数のエイリアスに追加します。

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「PutProvisionedConcurrencyConfig」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **RemovePermission** を使用する

以下のコード例は、RemovePermission の使用方法を示しています。

CLI

AWS CLI

既存の Lambda 関数から許可を削除する方法

次の remove-permission の例では、my-function という名前の関数を呼び出す許可を削除します。

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパー ガイド」の「AWS Lambda のリソースベースのポリシーを使用する」を参照してください。

- API の詳細については、「AWS CLIコマンドリファレンス」の「RemovePermission」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、Lambda 関数の指定された StatementId の関数ポリシーを削除します。

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
```

```
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「RemovePermission」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **TagResource** を使用する

以下のコード例は、TagResource の使用方法を示しています。

CLI

AWS CLI

既存の Lambda 関数にタグを追加する方法

次の tag-resource の例では、指定した Lambda 関数に DEPARTMENT のキー名と Department A の値を持つタグを追加します。

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパーガイド」の「 Lambda 関数をタグ付けする」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「TagResource」を参照してください。

PowerShell

Tools for PowerShell

例 1: 3 つのタグ (Washington、Oregon、California) およびそれぞれに関連付けされた値を、ARN で識別される指定の関数に追加します。

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「TagResource」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UntagResource** を使用する

以下のコード例は、UntagResource の使用方法を示しています。

CLI

AWS CLI

既存の Lambda 関数からタグを削除する方法

次の untag-resource の例では、DEPARTMENT タグというキー名のタグを my-function Lambda 関数から削除します。

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

このコマンドでは何も出力されません。

詳細については、「AWS Lambda デベロッパーガイド」の「 Lambda 関数をタグ付けする」を参照してください。

- API の詳細については、「AWS CLI Command Reference」の「UntagResource」を参照してください。

PowerShell

Tools for PowerShell

例 1: 提供されたタグを関数から削除します。-Force スイッチが指定されていない限り、cmdlet は続行する前に確認を求めます。タグを削除するため、サービスが 1 回呼び出されます。

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

例 2: 提供されたタグを関数から削除します。-Force スイッチが指定されていない限り、cmdlet は続行する前に確認を求めます。提供されたタグにつき、サービスに 1 回呼び出しが行われます。

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「UntagResource」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateAlias** を使用する

以下のコード例は、UpdateAlias の使用方法を示しています。

CLI

AWS CLI

関数エイリアスを更新する方法

次の update-alias の例では、my-function Lambda 関数のバージョン 3 を参照するように、LIVE という名前のエイリアスを更新します。

```
aws lambda update-alias \
```

```
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

出力:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数のエイリアスを設定する」を参照してください。

- API の詳細については、AWS CLI コマンドリファレンスの「UpdateAlias」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、既存の Lambda 関数エイリアスの設定を更新します。RoutingConfiguration の値を更新し、トラフィックの 60% (0.6) をバージョン 1 に変換します。

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「UpdateAlias」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateFunctionCode** を使用する

以下のコード例は、UpdateFunctionCode の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };
```

```
        var response = await
_lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
{response.LastModified}.");
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「UpdateFunctionCode」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
instructions in the cpp_lambda/README.md file. "
                    << std::endl;
```

```
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「UpdateFunctionCode」を参照してください。

CLI

AWS CLI

Lambda 関数のコードを更新するには

次の例 update-function-code では、未公開 ($LATEST) バージョンの my-function 関数のコードを、指定した zip ファイルの内容に置き換えます。

```
aws lambda update-function-code \
    --function-name  my-function \
```

```
    --zip-file fileb://my-function.zip
```

出力:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateFunctionCode」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
 *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
```

```
    log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
  functionName, err)
   } else {
    state = funcOutput.Configuration.State
   }
  }
  return state
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「UpdateFunctionCode」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「UpdateFunctionCode」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
{
    return $this->lambdaClient->updateFunctionCode([
        'FunctionName' => $functionName,
        'S3Bucket' => $s3Bucket,
        'S3Key' => $s3Key,
    ]);
}
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「UpdateFunctionCode」を参照してください。

PowerShell

Tools for PowerShell

例 1:「MyFunction」という名前の関数を、指定された zip ファイルに含まれる新しいコンテンツで更新します。C# .NET Core Lambda 関数には、zip ファイルはコンパイルされたアセンブリが含まれている必要があります。

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

例 2: この例は前の例と似ていますが、更新されたコードを含む Amazon S3 オブジェクトを使用して関数を更新します。

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName mybucket -Key
 UpdatedCode.zip
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「UpdateFunctionCode」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
 contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
 in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
```

```
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の「UpdateFunctionCode」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.

  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
```

```
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「UpdateFunctionCode」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;
```

```
        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
```

```
            .build())
    }
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「UpdateFunctionCode」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
            iv_functionname = iv_function_name
            iv_zipfile = io_zip_file
         ).

       MESSAGE 'Lambda function code updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「UpdateFunctionCode」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK または CLI で **UpdateFunctionConfiguration** を使用する

以下のコード例は、UpdateFunctionConfiguration の使用方法を示しています。

アクション例は、より大きなプログラムからのコードの抜粋であり、コンテキスト内で実行する必要があります。次のコード例で、このアクションのコンテキストを確認できます。

- 関数の使用を開始します

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
```

```
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
        Aws::Client::ClientConfiguration clientConfig;
```

```
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

CLI

AWS CLI

関数の設定を変更するには

次のupdate-function-configuration の例では、未公開 ($LATEST) バージョンの my-function 関数のメモリサイズを 256 MB に変更しています。

```
aws lambda update-function-configuration \
```

```
    --function-name  my-function \
    --memory-size 256
```

出力:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

詳細については、「AWS Lambda デベロッパーガイド」の「AWS Lambda 関数の設定」を参照してください。

- API の詳細については、「AWS CLI コマンドリファレンス」の「UpdateFunctionConfiguration」を参照してください。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string,
 envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
```

```
            'Environment' => $environment,
        ]);
    }
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

PowerShell

Tools for PowerShell

例 1: この例では、既存の Lambda 関数の設定を更新します

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- API の詳細については、「AWS Tools for PowerShell Cmdlet リファレンス」の「UpdateFunctionConfiguration」を参照してください。

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
```

```
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- API の詳細については、AWS SDK for Python (Boto3) API リファレンスの「UpdateFunctionConfiguration」を参照してください。

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
class LambdaWrapper
  attr_accessor :lambda_client

  def initialize
```

```
    @lambda_client = Aws::Lambda::Client.new
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       "LOG_LEVEL" => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/** Update the environment for a function. */
pub async fn update_function_configuration(
    &self,
    environment: Environment,
) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
    info!(
        ?environment,
        "Updating environment for {}", self.lambda_name
    );
    let updated = self
        .lambda_client
        .update_function_configuration()
        .function_name(self.lambda_name.clone())
        .environment(environment)
        .send()
        .await
        .map_err(anyhow::Error::from)?;

    self.wait_for_function_ready().await?;

    Ok(updated)
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
             iv_functionname = iv_function_name
             iv_runtime = iv_runtime
             iv_description  = 'Updated Lambda function'
             iv_memorysize  = iv_memory_size
         ).

       MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の「UpdateFunctionConfiguration」を参照してください。

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した Lambda のシナリオ

以下のコード例は、AWS SDK を使用して Lambda で一般的なシナリオを実装する方法を示しています。これらのシナリオは、Lumbda 内で複数の関数を呼び出すことによって特定のタスクを実行する方法を示しています。それぞれのシナリオには、GitHub へのリンクがあり、コードを設定および実行する方法についての説明が記載されています。

例

- AWS SDK を使用して Lambda 関数で登録済みの Amazon Cognito ユーザーを自動的に確認する
- AWS SDK を使用して Lambda 関数を使用して登録済みの Amazon Cognito ユーザーを自動的に移行する
- AWS SDK を使用して Lambda 関数の作成と使用を開始する
- AWS SDK を使用して Amazon Cognito ユーザー認証後に Lambda 関数を使用してカスタムアクティビティデータを書き込む

AWS SDK を使用して Lambda 関数で登録済みの Amazon Cognito ユーザーを自動的に確認する

次のコード例は、Lambda 関数を使用して登録済みの Amazon Cognito ユーザーを確認する方法を示しています。

- `PreSignUp` トリガーの Lambda 関数を呼び出すようにユーザープールを設定します。
- Amazon Cognito でユーザーをサインアップする
- Lambda 関数は DynamoDB テーブルをスキャンし、登録済みのユーザーを自動的に確認します。
- 新しいユーザーとしてサインインし、リソースをクリーンアップします。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

コマンドプロンプトからインタラクティブのシナリオを実行します。

```
// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(userPoolId string, functionArn
 string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
```

```
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(clientId string, usersTable string)
 (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(usersTable)
 if err != nil {
  panic(err)
 }
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
```

```
    if errors.As(err, &invalidPassword) {
     password = runner.questioner.AskPassword("Enter another password:", 8)
    } else {
     panic(err)
    }
   } else {
    signedUp = true
   }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(clientId string, userName string, password
 string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))
```

```
 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Lambda 関数を使用して PreSignUp トリガーを処理します。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
```

```
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
```

```
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

一般的なタスクを実行する構造体を作成します。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}
```

```
// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
```

```
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
```

```
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Amazon Cognito アクションをラップする構造体を作成します。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
```

```
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
```

```
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
```

```
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
```

```
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
```

```
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

DynamoDB アクションをラップする構造体を作成します。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}
```

```
// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
```

```
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

CloudWatch Logs アクションをラップする構造体を作成します。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
```

```
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

AWS CloudFormation アクションをラップする構造体を作成します。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

リソースをクリーンアップします。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}
```

```
func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
```

```
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の以下のトピックを参照してください。
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して Lambda 関数を使用して登録済みの Amazon Cognito ユーザーを自動的に移行する

次のコード例は、Lambda 関数を使用して登録済みの Amazon Cognito ユーザーを自動的に移行する方法を示しています。

- `MigrateUser` トリガーの Lambda 関数を呼び出すようにユーザープールを設定します。
- ユーザープールにないユーザー名と E メールで Amazon Cognito にサインインします。
- Lambda 関数は DynamoDB テーブルをスキャンし、登録済みのユーザーをユーザープールに自動的に移行します。
- パスワードを忘れた場合のフローを実行して、移行したユーザーのパスワードをリセットします。
- 新しいユーザーとしてサインインし、リソースをクリーンアップします。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

コマンドプロンプトからインタラクティブのシナリオを実行します。

```
import (
 "errors"
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
```

```
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(userPoolId string, functionArn
 string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(usersTable string, clientId string) (bool,
 actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")
```

```
 runner.helper.AddKnownUser(usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(clientId string, user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
```

```
 log.Println("To complete this example and successfully migrate a user to
Cognito, you must enter an email\n" +
  "you own that can receive a confirmation code.")
 return
}
codeDelivery, err := runner.cognitoActor.ForgotPassword(clientId, user.UserName)
if err != nil {
 panic(err)
}
log.Printf("\nA confirmation code has been sent to %v.",
*codeDelivery.Destination)
code := runner.questioner.Ask("Check your email and enter it here:")

confirmed := false
password := runner.questioner.AskPassword("\nEnter a password that has at least
eight characters, uppercase, lowercase, numbers and symbols.\n"+
 "(the password will not display as you type):", 8)
for !confirmed {
 log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
 err = runner.cognitoActor.ConfirmForgotPassword(clientId, code, user.UserName,
password)
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   password = runner.questioner.AskPassword("\nEnter another password:", 8)
  } else {
   panic(err)
  }
 } else {
  confirmed = true
 }
}
log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
log.Println("Signing in with your username and password...")
authResult, err := runner.cognitoActor.SignIn(clientId, user.UserName, password)
if err != nil {
 panic(err)
}
log.Printf("Successfully signed in. Your access token starts with: %v...\n",
(*authResult.AccessToken)[:10])
runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
*authResult.AccessToken)

log.Println(strings.Repeat("-", 88))
```

```
}

// Run runs the scenario.
func (runner *MigrateUser) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Lambda 関数を使用して MigrateUser トリガーを処理します。

```
const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
  ExpressionAttributeValues: expr.Values(),
 })
```

```
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if output.Items == nil || len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

一般的なタスクを実行する構造体を作成します。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
```

```
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}
```

```
// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Amazon Cognito アクションをラップする構造体を作成します。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
```

```
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}
```

```
// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
```

```
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
```

```
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
```

```
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

DynamoDB アクションをラップする構造体を作成します。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
```

```
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
```

```
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
```

```
 }
 return err
}
```

CloudWatch Logs アクションをラップする構造体を作成します。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
```

```
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

AWS CloudFormation アクションをラップする構造体を作成します。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

リソースをクリーンアップします。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
```

```
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の以下のトピックを参照してください。
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して Lambda 関数の作成と使用を開始する

次のコード例は、以下を実行する方法を示しています。

- IAM ロールと Lambda 関数を作成し、ハンドラーコードをアップロードします。
- 1 つのパラメーターで関数を呼び出して、結果を取得します。
- 関数コードを更新し、環境変数で設定します。
- 新しいパラメーターで関数を呼び出して、結果を取得します。返された実行ログを表示します。
- アカウントの関数を一覧表示し、リソースをクリーンアップします。

詳細については、「コンソールで Lambda 関数を作成する」を参照してください。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

Lambda アクションを実行するメソッドを作成します。

```
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
```

```
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };
```

```
        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
 functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
```

```
        return response.Configuration;
    }


    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }


    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
```

```
        }

        return functionList;
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
```

```
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

シナリオを実行する関数を作成します。

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;
```

```
namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
 host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
```

```
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
```

```
            functionName,
            bucketName,
            incrementKey,
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
```

```
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
```

```
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);

                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
```

```
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";

                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
```

```
            Console.WriteLine("Couldn't delete the role.");
        }

        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
```

```
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

```
    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;
public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
 value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
 value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
 calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
 entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
```

```
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
```

```
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }

        PressEnter();
    }
}
```

数値をインクリメントする Lambda ハンドラーを定義します。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
```

```
        {
            return 0;
        }
    }
}
```

算術演算を実行する 2 番目の Lambda ハンドラーを定義します。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
```

```
                break;
            case "subtract":
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- API の詳細については、「AWS SDK for .NET API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK for C++

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
```

```
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_8);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
```

```
                    << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
```

```
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
 std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
 return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }
```

```
        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "This function uses an environment variable to control the logging
level."
            << std::endl;
    std::cout
            << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
DEBUG."
            << std::endl;
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }
```

```
   // 6.  Invoke the updated Lambda function.
   do {
       int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
4 ", 1,
                                                  4);
       int x = askQuestionForInt("Enter an integer for the x value ");
       int y = askQuestionForInt("Enter an integer for the y value ");

       Aws::Utils::Json::JsonValue calculateJsonPayload;
       calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
       calculateJsonPayload.WithInteger("x", x);
       calculateJsonPayload.WithInteger("y", y);
       Aws::Lambda::Model::InvokeResult calculatedResult;
       if (invokeLambdaFunction(calculateJsonPayload,
                                Aws::Lambda::Model::LogType::Tail,
                                calculatedResult, client)) {
           Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
           Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                   jsonValue.View().GetAllObjects();
           auto iter = values.find("result");
           if (iter != values.end() && iter->second.IsIntegerType()) {
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsInteger() <<
std::endl;
           }
           else if (iter != values.end() && iter->second.IsFloatingPointType())
{
               std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                         << operators[operatorIndex - 1] << " "
                         << y << " is " << iter->second.AsDouble() << std::endl;
           }
           else {
               std::cout << "There was an error in execution. Here is the log."
                         << std::endl;
               Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
               std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
           }
       }
```

```
        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                          <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                  functionConfiguration.GetRuntime()) << ": "
                          << functionConfiguration.GetHandler()
                          << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
```

```
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                    << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                    << outcome.GetError().GetMessage()
                    << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
```

```
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }
```

```
            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- API の詳細については、「AWS SDK for C++ API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

Lambda 関数の開始方法を示すインタラクティブなシナリオを作成します。

```
// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
```

```
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole()
 funcName := scenario.CreateFunction(role)
 scenario.InvokeIncrement(funcName)
 scenario.UpdateFunction(funcName)
 scenario.InvokeCalculator(funcName)
 scenario.ListFunctions()
 scenario.Cleanup(role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole() *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(context.TODO(), &iam.GetRoleInput{
```

```
 RoleName: aws.String(roleName)})
if err != nil {
 var noSuch *iamtypes.NoSuchEntityException
 if errors.As(err, &noSuch) {
  log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
 } else {
  log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
   roleName, err)
 }
} else {
 role = getOutput.Role
 log.Printf("Found role %v.\n", *role.RoleName)
}
if role == nil {
 trustPolicy := PolicyDocument{
  Version: "2012-10-17",
  Statement: []PolicyStatement{{
   Effect:    "Allow",
   Principal: map[string]string{"Service": "lambda.amazonaws.com"},
   Action:    []string{"sts:AssumeRole"},
  }},
 }
 policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
 createOutput, err := iamClient.CreateRole(context.TODO(), &iam.CreateRoleInput{
  AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
  RoleName:                 aws.String(roleName),
 })
 if err != nil {
  log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
 }
 role = createOutput.Role
 _, err = iamClient.AttachRolePolicy(context.TODO(), &iam.AttachRolePolicyInput{
  PolicyArn: aws.String(policyArn),
  RoleName:  aws.String(roleName),
 })
 if err != nil {
  log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
err)
 }
 log.Printf("Created role %v.\n", *role.RoleName)
 log.Println("Let's give AWS a few seconds to propagate resources...")
 scenario.helper.Pause(10)
}
log.Println(strings.Repeat("-", 88))
```

```
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(role *iamtypes.Role)
 string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(funcName, parameters, false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
```

```
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(funcName, zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
```

```
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(funcName, calcParameters, true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions() {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
```

```
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(role *iamtypes.Role, funcName
 string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(context.TODO(),
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(context.TODO(),
 &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(context.TODO(), &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}
```

個別の Lambda アクションをラップする構造体を作成します。

```
// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(functionName string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(context.TODO(),
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(functionName string, handlerName
 string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
```

```
 _, err := wrapper.LambdaClient.CreateFunction(context.TODO(),
 &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython38,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(functionName string, zipPackage
 *bytes.Buffer) types.State {
 var state types.State
```

```
 _, err := wrapper.LambdaClient.UpdateFunctionCode(context.TODO(),
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(context.TODO(),
 &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(functionName string,
 envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(context.TODO(),
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
```

```
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(context.TODO())
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(functionName string) {
 _, err := wrapper.LambdaClient.DeleteFunction(context.TODO(),
 &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(functionName string, parameters any, getLog
 bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
```

```
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(context.TODO(),
 &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}

// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

シナリオの実行に役立つ関数を実装する構造体を作成します。

```
// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
```

```
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

数値をインクリメントする Lambda ハンドラーを定義します。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
```

```
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

算術演算を実行する 2 番目の Lambda ハンドラーを定義します。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
```

```
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- API の詳細については、「AWS SDK for Go API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

                Usage:
                    <functionName> <filePath> <role> <handler> <bucketName> <key>
\s

                Where:
                    functionName - The name of the Lambda function.\s
                    filePath - The path to the .zip or .jar where the code is
 located.\s
                    role - The AWS Identity and Access Management (IAM) service
 role that has Lambda permissions.\s
```

```
                    handler - The fully qualified method name (for example,
example.Handler::handleRequest).\s
                    bucketName - The Amazon Simple Storage Service (Amazon S3)
bucket name that contains the .zip or .jar used to update the Lambda function's
code.\s
                    key - The Amazon S3 key name that represents the .zip or .jar
(for example, LambdaHello-1.0-SNAPSHOT.jar).
                    """;

       if (args.length != 6) {
           System.out.println(usage);
           System.exit(1);
       }

       String functionName = args[0];
       String filePath = args[1];
       String role = args[2];
       String handler = args[3];
       String bucketName = args[4];
       String key = args[5];

       Region region = Region.US_WEST_2;
       LambdaClient awsLambda = LambdaClient.builder()
               .region(region)
               .build();

       System.out.println(DASHES);
       System.out.println("Welcome to the AWS Lambda example scenario.");
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("1. Create an AWS Lambda function.");
       String funArn = createLambdaFunction(awsLambda, functionName, filePath,
role, handler);
       System.out.println("The AWS Lambda ARN is " + funArn);
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
       getFunction(awsLambda, functionName);
       System.out.println(DASHES);

       System.out.println(DASHES);
```

```
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    public static String createLambdaFunction(LambdaClient awsLambda,
            String functionName,
            String filePath,
            String role,
            String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
```

```
            InputStream is = new FileInputStream(filePath);
            SdkBytes fileToUpload = SdkBytes.fromInputStream(is);

            FunctionCode code = FunctionCode.builder()
                    .zipFile(fileToUpload)
                    .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                    .functionName(functionName)
                    .description("Created by the Lambda Java API")
                    .code(code)
                    .handler(handler)
                    .runtime(Runtime.JAVA8)
                    .role(role)
                    .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException | FileNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
```

```
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {

        InvokeResponse res;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                    .functionName(functionName)
                    .payload(payload)
                    .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);
```

```
        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                    .functionName(functionName)
                    .publish(true)
                    .s3Bucket(bucketName)
                    .s3Key(key)
                    .build();

            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                    .functionName(functionName)
                    .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                    .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                    .functionName(functionName)
```

```
                    .handler(handler)
                    .runtime(Runtime.JAVA11)
                    .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                    .functionName(functionName)
                    .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- API の詳細については、「AWS SDK for Java 2.x API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

AWS Identity and Access Management (IAM) ロールを作成して、Lambda にログへの書き込み権限を付与します。

```
    log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

Lambda 関数を作成し、ハンドラーコードをアップロードします。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
```

```
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

1 つのパラメーターで関数を呼び出して、結果を取得します。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

関数コードを更新し、Lambda 環境を環境可変で設定します。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });
```

```
  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  return client.send(command);
};
```

アカウントの関数を一覧表示します。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

IAM ロールと Lambda 関数を削除します。

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
```

```
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- API の詳細については、「AWS SDK for JavaScript API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
 <key>

        Where:
            functionName - The name of the AWS Lambda function.
            role - The AWS Identity and Access Management (IAM) service role that
 has AWS Lambda permissions.
```

```
            handler - The fully qualified method name (for example,
example.Handler::handleRequest).
            bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
            updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
            key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
            """

   if (args.size != 6) {
       println(usage)
       exitProcess(1)
   }

   val functionName = args[0]
   val role = args[1]
   val handler = args[2]
   val bucketName = args[3]
   val updatedBucketName = args[4]
   val key = args[5]

   println("Creating a Lambda function named $functionName.")
   val funArn = createScFunction(functionName, bucketName, key, handler, role)
   println("The AWS Lambda ARN is $funArn")

   // Get a specific Lambda function.
   println("Getting the $functionName AWS Lambda function.")
   getFunction(functionName)

   // List the Lambda functions.
   println("Listing all AWS Lambda functions.")
   listFunctionsSc()

   // Invoke the Lambda function.
   println("*** Invoke the Lambda function.")
   invokeFunctionSc(functionName)

   // Update the AWS Lambda function code.
   println("*** Update the Lambda function code.")
   updateFunctionCode(functionName, updatedBucketName, key)

   // println("*** Invoke the function again after updating the code.")
   invokeFunctionSc(functionName)
```

```
    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java8
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
```

```
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
```

```
    functionNameVal: String?,
    bucketName: String?,
    key: String?
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java11
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}

suspend fun delFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
```

```
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- API の詳細については、「AWS SDK for Kotlin API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コードサンプルリポジトリでの設定と実行の方法を確認してください。

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("-------------------------------------\n");
```

```
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("-------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";

        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "test-example-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
```

```
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
```

```
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
 equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
```

```
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
$listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
```

```
}
```

- API の詳細については、「AWS SDK for PHP API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

数値をインクリメントする Lambda ハンドラーを定義します。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
```

```
                is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

算術演算を実行する 2 番目の Lambda ハンドラーを定義します。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.
```

```
    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

Lambda アクションをラップする関数を作成します。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
 buffer. This
```

```
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
```

```
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
 created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
 iam_role_name)
            else:
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise
```

```
        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
```

```
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.8",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
```

```
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

```
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
```

```
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

シナリオを実行する関数を作成します。

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
    :param calculator_file: The name of the file that contains the calculator
 Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
 as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")
```

```
    print("-" * 88)
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
```

```
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
q.is_yesno)
    print("-" * 88)
```

```
    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- API の詳細については、「AWS SDK for Python (Boto3) API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

ログを書き込むことができる Lambda 関数に前提条件となる IAM アクセス権限を設定します。

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
  role_policy = {
    'Version': "2012-10-17",
    'Statement': [
      {
        'Effect': "Allow",
        'Principal': {
          'Service': "lambda.amazonaws.com"
        },
        'Action': "sts:AssumeRole"
      }
    ]
  }
  case action
  when "create"
    role = $iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    $iam_client.attach_role_policy(
      {
```

```
          policy_arn: "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole",
          role_name: iam_role_name
        }
      )
      $iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
        w.max_attempts = 5
        w.delay = 5
      end
      @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
      @logger.debug("Enforcing a 10-second sleep to allow IAM role to activate
 fully.")
      sleep(10)
      return role, role_policy.to_json
    when "destroy"
      $iam_client.detach_role_policy(
        {
          policy_arn: "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole",
          role_name: iam_role_name
        }
      )
      $iam_client.delete_role(
        role_name: iam_role_name
      )
      @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
    else
      raise "Incorrect action provided. Must provide 'create' or 'destroy'"
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating role or attaching policy:\n
 #{e.message}")
  end
```

呼び出しパラメータとして指定された数値を増やす Lambda ハンドラーを定義します。

```
require "logger"

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
```

```
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV["LOG_LEVEL"]
  logger.level = case log_level
                 when "debug"
                   Logger::DEBUG
                 when "info"
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug("This is a debug log message.")
  logger.info("This is an info log message. Code executed successfully!")
  number = event["number"].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

Lambda 関数のデプロイパッケージを圧縮します。

```
  # Creates a Lambda deployment package in .zip format.
  # This zip can be passed directly as a string to Lambda when creating the
 function.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
  def create_deployment_package(source_file)
    Dir.chdir(File.dirname(__FILE__))
    if File.exist?("lambda_function.zip")
      File.delete("lambda_function.zip")
      @logger.debug("Deleting old zip: lambda_function.zip")
    end
    Zip::File.open("lambda_function.zip", create: true) {
      |zipfile|
      zipfile.add("lambda_function.rb", "#{source_file}.rb")
    }
```

```
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read("lambda_function.zip").to_s
  rescue StandardError => e
    @logger.error("There was an error creating deployment package:\n
#{e.message}")
  end
```

新しい Lambda 関数の作成

```
  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function. This
  #                      must include the file name and the function name.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
  #                            code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: "ruby2.7",
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    "LOG_LEVEL" => "info"
                                                  }
                                                }
                                              })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
```

```
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

オプションのランタイムパラメータを使用して Lambda 関数を呼び出します。

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name}
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

Lambda 関数の設定を更新して、新しい環境変数を挿入します。

```
  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       "LOG_LEVEL" => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
```

```
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Lambda 関数のコードを、別のコードを含む別のデプロイパッケージで更新します。

```
  # Updates the code for a Lambda function by submitting a .zip archive that
contains
  # the code for the function.

  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name}) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

組み込みのページネーターを使用して、既存のすべての Lambda 関数を一覧表示します。

```
  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
```

```
    @lambda_client.list_functions.each do |response|
      response["functions"].each do |function|
        functions.append(function["function_name"])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

特定の Lambda 関数を削除します。

```
  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print "Done!".green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- API の詳細については、「AWS SDK for Ruby API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

このシナリオで使用した依存関係を含む Cargo.toml。

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "~4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

このシナリオの Lambda 呼び出しを効率化するユーティリティのコレクション。このファイルはクレート内の src/ations.rs というファイルです。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;
```

```
    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl ToString for Operation {
    fn to_string(&self) -> String {
        match self {
            Operation::Plus => "plus".to_string(),
            Operation::Minus => "minus".to_string(),
            Operation::Times => "times".to_string(),
            Operation::DividedBy => "divided-by".to_string(),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
```

```
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);
```

```
impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
 name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
 bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
 Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
 "rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
```

```
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

 .location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    // snippet-start:[lambda.rust.scenario.prepare_function]
    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
```

```
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
    // snippet-end:[lambda.rust.scenario.prepare_function]

    // snippet-start:[lambda.rust.scenario.create_function]
    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
```

```
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }
    // snippet-end:[lambda.rust.scenario.create_function]

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
```

```
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
```

```
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
                }
            }
```

```
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    // snippet-start:[lambda.rust.scenario.get_function]
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.get_function]

    // snippet-start:[lambda.rust.scenario.list_functions]
    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.list_functions]

    // snippet-start:[lambda.rust.scenario.invoke]
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
```

```
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
    // snippet-end:[lambda.rust.scenario.invoke]

    // snippet-start:[lambda.rust.scenario.update_function_code]
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }
    // snippet-end:[lambda.rust.scenario.update_function_code]

    // snippet-start:[lambda.rust.scenario.update_function_configuration]
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
```

```
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }
    // snippet-end:[lambda.rust.scenario.update_function_configuration]

    // snippet-start:[lambda.rust.scenario.delete_function]
    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);
```

```
        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
    // snippet-end:[lambda.rust.scenario.delete_function]

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
```

```
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };

        (delete_function, delete_bucket)
    }
}

/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

コマンドラインフラグを使用して一部の動作を制御し、シナリオを最初から最後まで実行するためのバイナリ。このファイルはクレート内の src/bin/scenario.rs というファイルです。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0

/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.
```

```
The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
```

```
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
```

```
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}

// snippet-start:[lambda.rust.scenario.log_invoke_output]
fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
// snippet-end:[lambda.rust.scenario.log_invoke_output]

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");
```

```
    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
```

```
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- API の詳細については、「AWS SDK for Rust API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",` &&
                 `"Statement": [` &&
                   `{` &&
                     `"Effect": "Allow",` &&
                     `"Action": [` &&
                       `"sts:AssumeRole"` &&
                     `],` &&
                     `"Principal": {` &&
                       `"Service": [` &&
                         `"lambda.amazonaws.com"` &&
                       `]` &&
                     `}` &&
                   `}` &&
                 `]` &&
               `}`.
       TRY.
           DATA(lo_create_role_output) =  lo_iam->createrole(
                   iv_rolename = iv_role_name
                   iv_assumerolepolicydocument = lv_policy_document
                   iv_description = 'Grant lambda permission to write to logs'
               ).
           MESSAGE 'IAM role created.' TYPE 'I'.
           WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
ready for use. "
         CATCH /aws1/cx_iamentityalrdyexex.
           MESSAGE 'IAM role already exists.' TYPE 'E'.
         CATCH /aws1/cx_iaminvalidinputex.
```

```
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lo_create_role_output->get_role( )->get_arn( )
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example'
             ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.
```

```
        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_initial_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json
                  ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
```

```
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables )
              ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
```

```
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}`
            ).
            DATA(lo_updated_invoke_output) =  lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json
                  ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
```

```
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole'
            ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- API の詳細については、「AWS SDK for SAP ABAP API リファレンス」の以下のトピックを参照してください。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用して Amazon Cognito ユーザー認証後に Lambda 関数を使用してカスタムアクティビティデータを書き込む

次のコード例は、Amazon Cognito ユーザー認証後に Lambda 関数を使用してカスタムアクティビティデータを書き込む方法を示しています。

- 管理者関数を使用して、ユーザーをユーザープールに追加します。
- `PostAuthentication` トリガーの Lambda 関数を呼び出すようにユーザープールを設定します。
- 新しいユーザーを Amazon Cognito にサインインします。
- Lambda 関数は、カスタム情報を CloudWatch Logs と DynamoDB テーブルに書き込みます。
- DynamoDB テーブルからカスタムデータを取得して表示し、リソースをクリーンアップします。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。用例一覧を検索し、AWS コード例リポジトリでの設定と実行の方法を確認してください。

コマンドプロンプトからインタラクティブのシナリオを実行します。

```
// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(userPoolId string, tableName string)
 (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
```

```
 users, err := runner.helper.GetKnownUsers(tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(userPoolId string,
 activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
```

```
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(clientId string, userName string, password
 string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(tableName string, userName
 string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(tableName)
 if err != nil {
```

```
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup()
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(stackOutputs["TableName"], userName)

 runner.resources.Cleanup()

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
```

```
 log.Println(strings.Repeat("-", 88))
}
```

Lambda 関数を使用して PostAuthentication トリガーを処理します。

```
const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
```

```
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Panicln(err)
 }
```

```
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

一般的なタスクを実行する構造体を作成します。

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(stackName string) (actions.StackOutputs, error)
 PopulateUserTable(tableName string)
 GetKnownUsers(tableName string) (actions.UserList, error)
 AddKnownUser(tableName string, user actions.User)
 ListRecentLogEvents(functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
```

```
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(stackName string)
 (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(tableName string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(tableName string) (actions.UserList,
 error) {
 knownUsers, err := helper.dynamoActor.Scan(tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(tableName string, user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
```

```
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Amazon Cognito アクションをラップする構造体を作成します。

```
type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}
```

```
// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(userPoolId string,
 triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(context.TODO(),
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(context.TODO(),
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
```

```
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(clientId string, userName string, password
 string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(context.TODO(),
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(clientId string, userName string, password
 string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(context.TODO(),
 &cognitoidentityprovider.InitiateAuthInput{
```

```
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(clientId string, userName string)
 (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(context.TODO(),
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(clientId string, code string,
 userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(context.TODO(),
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
```

```
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(userAccessToken string) error {
 _, err := actor.CognitoClient.DeleteUser(context.TODO(),
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(userPoolId string, userName string,
 userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(context.TODO(),
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
```

```
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(userPoolId string, userName
 string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(context.TODO(),
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

DynamoDB アクションをラップする構造体を作成します。

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(tableName string) error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
```

```
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(context.TODO(),
 &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(tableName string) (UserList, error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(context.TODO(), &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(tableName string, user User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
```

```
 }
 _, err = actor.DynamoClient.PutItem(context.TODO(), &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

CloudWatch Logs アクションをラップする構造体を作成します。

```
type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(functionName string)
 (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(context.TODO(),
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
```

```
func (actor CloudWatchLogsActions) GetLogEvents(functionName string,
 logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(context.TODO(),
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

AWS CloudFormation アクションをラップする構造体を作成します。

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(stackName string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(context.TODO(),
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
```

```
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

リソースをクリーンアップします。

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()
```

```
 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- API の詳細については、「AWS SDK for Go API リファレンス」の以下のトピックを参照してください。
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した Lambda 用のサーバーレスサンプル

以下のコード例は、AWS SDK で Lambda を使用する方法を示しています。

例

- Lambda 関数で Amazon RDS データベースに接続する
- Kinesis トリガーから Lambda 関数を呼び出す
- DynamoDB トリガーから Lambda 関数を呼び出す
- Amazon DocumentDB トリガーから Lambda 関数を呼び出す
- Amazon S3 トリガーから Lambda 関数を呼び出す
- Amazon SNS トリガーから Lambda 関数を呼び出す
- Amazon SQS トリガーから Lambda 関数を呼び出す
- Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート
- DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする
- Amazon SQS トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

Lambda 関数で Amazon RDS データベースに接続する

次のコード例は、RDS データベースに接続する Lambda 関数を実装する方法を示しています。この関数は、シンプルなデータベースリクエストを実行し、結果を返します。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda 関数での Amazon RDS データベースへの接続

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = "DatabaseName"
 var dbUser string = "DatabaseUser"
 var dbHost string = "mysqldb.123456789012.us-east-1.rds.amazonaws.com"
 var dbPort int = 3306
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = "us-east-1"

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }
```

```
 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

JavaScript

SDK for JavaScript (v2)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Javascript を使用した Lambda 関数での Amazon RDS データベースへの接続

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
```

```
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Kinesis トリガーから Lambda 関数を呼び出す

次のコード例では、Kinesis ストリームからレコードを受信することによってトリガーされるイベントを受け取る、Lambda 関数の実装方法を示しています。この関数は Kinesis ペイロードを取得し、それを Base64 からデコードして、そのレコードの内容をログ記録します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での Kinesis イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での Kinesis イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB トリガーから Lambda 関数を呼び出す

次のコードの例では、DynamoDB ストリームからレコードを受信することによってトリガーされるイベントを受け取る Lambda 関数の実装方法が示されています。関数は DynamoDB ペイロードを取得し、レコードの内容をログ記録します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
```

```
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で DynamoDB イベントの消費。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
```

```
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

TypeScript を使用した Lambda での DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
```

```
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
```

```
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で DynamoDB イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end
```

```
  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB イベントを利用します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
```

```
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon DocumentDB トリガーから Lambda 関数を呼び出す

次のコードの例では、DocumentDB 変更ストリームからレコードを受信することによってトリガーされるイベントを受け取る Lambda 関数の実装方法が示されています。関数は DocumentDB ペイロードを取得し、レコードの内容をログ記録します。

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で Amazon DocumentDB イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
```

```
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で Amazon DocumentDB イベントの消費。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
```

```
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で Amazon DocumentDB イベントの消費。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
```

```
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で Amazon DocumentDB イベントの消費。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon S3 トリガーから Lambda 関数を呼び出す

次のコード例は、S3 バケットにオブジェクトをアップロードすることによってトリガーされるイベントを受け取る Lambda 関数を実装する方法を示しています。この関数は、イベントパラメータから S3 バケット名とオブジェクトキーを取得し、Amazon S3 API を呼び出してオブジェクトのコンテンツタイプを取得してログに記録します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }
```

```
        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
```

```
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

exports.handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

TypeScript を使用して Lambda で S3 イベントを消費する。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });
```

```
export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用して Lambda で S3 イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;
```

```
require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で S3 イベントを消費します。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
 encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
 your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda での S3 イベントの消費。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で S3 イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
```

```
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon SNS トリガーから Lambda 関数を呼び出す

次のコード例は、SNS トピックからメッセージを受信することによってトリガーされるイベントを受け取る Lambda 関数を実装する方法を示しています。この関数はイベントパラメータからメッセージを取得し、各メッセージの内容を記録します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行する方法を確認してください。

.NET を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行する方法を確認してください。

JavaScript を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

TypeScript を使用した Lambda での SNS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で SNS イベントを消費します。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SNS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で SNS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon SQS トリガーから Lambda 関数を呼び出す

次のコード例では、SQS キューからメッセージを受信することによってトリガーされるイベントを受け取る、Lambda 関数の実装方法を示しています。この関数はイベントパラメータからメッセージを取得し、各メッセージの内容を記録します。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

TypeScript を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での SQS イベントの消費。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での SQS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SQS イベントの消費。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で SQS イベントを消費します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Kinesis トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

以下のコード例では、Kinesis ストリームからイベントを受け取る Lambda 関数のための、部分的なバッチレスポンスの実装方法を示しています。この関数は、レスポンスとしてバッチアイテムの失敗を報告し、対象のメッセージを後で再試行するよう Lambda に伝えます。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
```

```
namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
```

```
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Javascript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

TypeScript を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で Kinesis バッチアイテム失敗のレポートをします。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での Kinesis バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

DynamoDB トリガーで Lambda 関数のバッチアイテムの失敗をレポートする

次のコードの例では、DynamoDB ストリームからイベントを受け取る Lambda 関数の部分的なバッチレスポンスの実装方法が示されています。この関数は、レスポンスとしてバッチアイテムの失敗を報告し、対象のメッセージを後で再試行するよう Lambda に伝えます。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
```

```
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }
```

```
 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
```

```
public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 Serializable> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

TypeScript を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { DynamoDBBatchItemFailure, DynamoDBStreamEvent } from "aws-lambda";

export const handler = async (event: DynamoDBStreamEvent):
 Promise<DynamoDBBatchItemFailure[]> => {
```

```
    const batchItemsFailures: DynamoDBBatchItemFailure[] = []
    let curRecordSequenceNumber

    for(const record of event.Records) {
        curRecordSequenceNumber = record.dynamodb?.SequenceNumber

        if(curRecordSequenceNumber) {
            batchItemsFailures.push({
                itemIdentifier: curRecordSequenceNumber
            })
        }
    }

    return batchItemsFailures
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での DynamoDB バッチ項目失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';
```

```
class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}
```

```
$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用して Lambda で DynamoDB のバッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }
```

```
        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Amazon SQS トリガーを使用した Lambda 関数でのバッチアイテムの失敗のレポート

以下のコード例では、SQS キューからイベントを受け取る Lambda 関数のための、部分的なバッチレスポンスの実装方法を示しています。この関数は、レスポンスとしてバッチアイテムの失敗を報告し、対象のメッセージを後で再試行するよう Lambda に伝えます。

.NET

AWS SDK for .NET

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

.NET を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
```

```
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Go を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "encoding/json"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {

  if /* Your message processing condition here */ {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Java を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {

        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
        String messageId = "";
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
                messageId = message.getMessageId();
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(messageId));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

JavaScript を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
export const handler = async (event, context) => {
    const batchItemFailures = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

TypeScript を使用して Lambda で SQS バッチ項目の失敗を報告します。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
```

```
    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK for PHP

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

PHP を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }
```

```
    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Python を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                # process message
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Ruby を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
```

```
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK for Rust

> **Note**
>
> GitHub には、その他のリソースもあります。サーバーレスサンプルリポジトリで完全な例を検索し、設定および実行の方法を確認してください。

Rust を使用した Lambda での SQS バッチアイテム失敗のレポート。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
```

```
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

AWS SDK を使用した Lambda のクロスサービスの例

次のサンプルアプリケーションでは、AWS SDK を使用して Lambda を他の AWS のサービス と組み合わせます。それぞれの例には、GitHub へのリンクがあり、アプリケーションを設定および実行する方法についての説明を参照できます。

例

- COVID-19 データを追跡する API Gateway REST API を作成する
- 貸出ライブラリ REST API を作成する
- ステップ関数でメッセンジャーアプリケーションを作成する
- ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成
- API Gateway で WebSocket チャットアプリケーションを作成する
- 顧客からのフィードバックを分析し、音声を合成するアプリケーションの作成
- ブラウザからの Lambda 関数の呼び出し
- S3 Object Lambda でアプリケーションのデータを変換する
- API Gateway を使用して Lambda 関数を呼び出す

- Step Functions を使用して Lambda 関数を呼び出す
- スケジュールされたイベントを使用した Lambda 関数の呼び出し

COVID-19 データを追跡する API Gateway REST API を作成する

次のコード例は、架空のデータを使用して、米国の COVID-19 の日常的なケースを追跡するシステムをシミュレートする REST API を作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) で AWS Chalice を使用して Amazon API Gateway、AWS Lambda、Amazon DynamoDB を使用するサーバーレス REST API を作成する方法を示しています。REST API は、架空のデータを使用して、米国の COVID-19 の日常的なケースを追跡するシステムをシミュレートします。以下ではその方法を説明しています。

- API Gateway を介して送信される REST リクエストを処理するために呼び出される Lambda 関数内のルートを、AWS Chalice を使って定義します。
- Lambda 関数を使用して、DynamoDB テーブルにデータを取得して保存し、REST リクエストを処理します。
- AWS CloudFormation テンプレート内のテーブル構造とセキュリティロールのリソースを定義します。
- AWS Chalice と CloudFormation を使用して、必要なすべてのリソースをパッケージ化してデプロイします。
- CloudFormation を使用して、作成されたすべてのリソースをクリーンアップします。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

貸出ライブラリ REST API を作成する

以下のコード例は、Amazon Aurora データベースでバックアップされた REST API を使用して、常連客が書籍の貸出と返却できる貸出ライブラリを作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を Amazon Relational Database Service (Amazon RDS) API および AWS Chalice で使用して、Amazon Aurora データベースに基づく REST API を作成する方法を示します。Web サービスは完全にサーバーレスであり、常連客が本を借りたり返却したりできるシンプルな貸し出しライブラリを表しています。以下ではその方法を説明しています。

- サーバーレス Aurora データベースクラスターを作成および管理します。
- AWS Secrets Manager を使用してデータベース認証情報を管理します。
- Amazon RDS を使用してデータをデータベースに出し入れするデータストレージレイヤーを実装します。
- AWS Chalice を使用して、サーバーレス REST API を Amazon API Gateway および AWS Lambda にデプロイします。
- リクエストパッケージを使用して、Web サービスにリクエストを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API Gateway
- Aurora
- Lambda
- Secrets Manager

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ステップ関数でメッセンジャーアプリケーションを作成する

次のコード例は、データベーステーブルからメッセージレコードを取得する AWS Step Functions メッセンジャーアプリケーションを作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) を AWS Step Functions と使用して、Amazon DynamoDB テーブルからメッセージレコードを取得し、Amazon Simple Queue Service (Amazon SQS) で送信するメッセンジャーアプリケーションを作成する方法を紹介します。ステートマシンは AWS Lambda 関数を使用して、データベースで未送信メッセージをスキャンします。

- Amazon DynamoDB テーブルからメッセージレコードを取得および更新するステートマシンを作成します。
- ステートマシンの定義を更新して、Amazon Simple Queue Service (Amazon SQS) にもメッセージを送信します。
- ステートマシンの実行を開始および停止します。
- サービス統合を使用して、ステートマシンから Lambda、DynamoDB、および Amazon SQS に接続します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda
- Amazon SQS
- ステップ関数

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ユーザーがラベルを使用して写真を管理できる写真アセット管理アプリケーションの作成

以下のコード例は、ユーザーがラベルを使用して写真を管理できるサーバーレスアプリケーションの作成方法を示しています。

.NET

AWS SDK for .NET

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Amazon Rekognition を使用して画像内のラベルを検出し、保存して後で取得できるようにする写真アセット管理アプリケーションの開発方法を示します。

完全なソースコードと設定および実行の手順については、GitHub で完全な例を参照してください。

この例のソースについて詳しくは、AWS コミュニティでブログ投稿を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

API Gateway で WebSocket チャットアプリケーションを作成する

次のコード例は、Amazon API Gateway 上に構築された WebSocket API によって提供されるチャットアプリケーションを作成する方法を示しています。

Python

SDK for Python (Boto3)

AWS SDK for Python (Boto3) と Amazon API Gateway V2 を使用して、AWS Lambda や Amazon DynamoDB と統合する WebSocket API を作成する方法を示します。

- API Gateway で提供される WebSocket API を作成します。
- DynamoDB に接続を保存し、他のチャット参加者にメッセージを投稿する Lambda ハンドラを定義します。
- WebSocket チャットアプリケーションに接続し、WebSockets パッケージを使用してメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

顧客からのフィードバックを分析し、音声を合成するアプリケーションの作成

次のコード例では、顧客のコメントカードを分析し、それを元の言語から翻訳し、顧客の感情を判断し、翻訳されたテキストから音声ファイルを生成するアプリケーションの作成方法を示しています。

.NET

AWS SDK for .NET

このサンプルアプリケーションは、顧客フィードバックカードを分析し、保存します。具体的には、ニューヨーク市の架空のホテルのニーズを満たします。このホテルでは、お客様からのフィードバックをさまざまな言語で書かれた実際のコメントカードの形で受け取ります。そのフィードバックは、ウェブクライアントを通じてアプリにアップロードされます。コメントカードの画像をアップロードされると、次の手順が発生します。

- テキストは Amazon Textract を使用して、画像から抽出されます。
- Amazon Comprehend は、抽出されたテキストの感情とその言語を決定します。
- 抽出されたテキストは、Amazon Translate を使用して英語に翻訳されます。
- Amazon Polly は抽出されたテキストからオーディオファイルを合成します。

完全なアプリは AWS CDK を使用してデプロイすることができます。ソースコードとデプロイ手順については、GitHub のプロジェクトを参照してください。

この例で使用されているサービス

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK for Java 2.x

このサンプルアプリケーションは、顧客フィードバックカードを分析し、保存します。具体的には、ニューヨーク市の架空のホテルのニーズを満たします。このホテルでは、お客様からのフィードバックをさまざまな言語で書かれた実際のコメントカードの形で受け取ります。そのフィードバックは、ウェブクライアントを通じてアプリにアップロードされます。コメントカードの画像をアップロードされると、次の手順が発生します。

- テキストは Amazon Textract を使用して、画像から抽出されます。
- Amazon Comprehend は、抽出されたテキストの感情とその言語を決定します。
- 抽出されたテキストは、Amazon Translate を使用して英語に翻訳されます。

- Amazon Polly は抽出されたテキストからオーディオファイルを合成します。

完全なアプリは AWS CDK を使用してデプロイすることができます。ソースコードとデプロイ手順については、GitHub のプロジェクトを参照してください。

この例で使用されているサービス

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK for JavaScript (v3)

このサンプルアプリケーションは、顧客フィードバックカードを分析し、保存します。具体的には、ニューヨーク市の架空のホテルのニーズを満たします。このホテルでは、お客様からのフィードバックをさまざまな言語で書かれた実際のコメントカードの形で受け取ります。そのフィードバックは、ウェブクライアントを通じてアプリにアップロードされます。コメントカードの画像をアップロードされると、次の手順が発生します。

- テキストは Amazon Textract を使用して、画像から抽出されます。
- Amazon Comprehend は、抽出されたテキストの感情とその言語を決定します。
- 抽出されたテキストは、Amazon Translate を使用して英語に翻訳されます。
- Amazon Polly は抽出されたテキストからオーディオファイルを合成します。

完全なアプリは AWS CDK を使用してデプロイすることができます。ソースコードとデプロイ手順については、GitHub のプロジェクトを参照してください。次の抜粋は、AWS SDK for JavaScript が Lambda 関数内でどのように使用されるかを示しています。

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
```

```
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
```

```
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
```

```
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });
```

```
  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

この例で使用されているサービス

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK for Ruby

このサンプルアプリケーションは、顧客フィードバックカードを分析し、保存します。具体的には、ニューヨーク市の架空のホテルのニーズを満たします。このホテルでは、お客様からのフィードバックをさまざまな言語で書かれた実際のコメントカードの形で受け取ります。そのフィードバックは、ウェブクライアントを通じてアプリにアップロードされます。コメントカードの画像をアップロードされると、次の手順が発生します。

- テキストは Amazon Textract を使用して、画像から抽出されます。
- Amazon Comprehend は、抽出されたテキストの感情とその言語を決定します。
- 抽出されたテキストは、Amazon Translate を使用して英語に翻訳されます。
- Amazon Polly は抽出されたテキストからオーディオファイルを合成します。

完全なアプリは AWS CDK を使用してデプロイすることができます。ソースコードとデプロイ手順については、GitHub のプロジェクトを参照してください。

この例で使用されているサービス

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

AWS SDK デベロッパーガイドとコード例の詳細なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

ブラウザからの Lambda 関数の呼び出し

次のコード例は、ブラウザから AWS Lambda 関数を呼び出す方法を示しています。

JavaScript

SDK for JavaScript (v2)

Amazon DynamoDB のテーブルをユーザーの選択内容で更新する AWS Lambda 関数を使用した、ブラウザベースのアプリケーションを作成することができます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda

SDK for JavaScript (v3)

Amazon DynamoDB のテーブルをユーザーの選択内容で更新する AWS Lambda 関数を使用した、ブラウザベースのアプリケーションを作成することができます。このアプリは AWS SDK for JavaScript v3 を使用します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- Lambda

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

S3 Object Lambda でアプリケーションのデータを変換する

次のコード例では、S3 Object Lambda でアプリケーションのデータを変換する方法を示しています。

.NET

AWS SDK for .NET

オブジェクトがリクエストしたクライアントまたはアプリケーションのニーズに合うように、標準の S3 GET リクエストにカスタムコードを追加し、S3 から取得したリクエストされたオブジェクトを変更する方法を示しています。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- Lambda
- Amazon S3

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

API Gateway を使用して Lambda 関数を呼び出す

次のコード例は、Amazon API Gateway によって呼び出される AWS Lambda 関数の作成方法を示しています。

Java

SDK for Java 2.x

Lambda Java ランタイム API を使用して AWS Lambda 関数を作成する方法を示します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、Amazon API Gateway によって呼び出される Lambda 関数を作成する方法を示します。この関数は、Amazon DynamoDB テーブルをスキャンして、Amazon Simple Notification Service (Amazon SNS) を使用して、従業員に年間の記念日を祝福するテキストメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Lambda JavaScript ランタイム API を使用して AWS Lambda 関数を作成する方法を示します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、Amazon API Gateway によって呼び出される Lambda 関数を作成する方法を示します。この関数は、Amazon DynamoDB テーブルをスキャンして、Amazon Simple Notification Service (Amazon SNS) を使用して、従業員に年間の記念日を祝福するテキストメッセージを送信します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

この例は、AWS Lambda 関数を対象とする Amazon API Gateway REST API を作成して使用する方法を示しています。Lambda ハンドラーは、HTTP メソッドに基づいてルーティングす

る方法を示します。クエリ文字列、ヘッダー、および本文からデータを取得する方法。そして、JSON 応答を返す方法。

- Lambda 関数をデプロイします。
- API ゲートウェイ REST API を作成します。
- Lambda 関数をターゲットとする REST リソースを作成します。
- API Gateway に Lambda 関数を呼び出す権限を付与します。
- リクエストパッケージを使用して、REST API にリクエストを送信します。
- デモ中に作成されたすべてのリソースをクリーンアップします。

この例は GitHub で最もよく確認できます。完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- API Gateway
- Lambda

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Step Functions を使用して Lambda 関数を呼び出す

次のコード例は、AWS Lambda 関数を順次呼び出す AWS Step Functions ステートマシンを作成する方法を示しています。

Java

SDK for Java 2.x

AWS Step Functions と AWS SDK for Java 2.x を使用して AWS サーバーレスワークフローを作成する方法を示します。各ワークフローステップは、AWS Lambda 関数を使用して実装されます。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB

- Lambda
- Amazon SES
- Step Functions

JavaScript

SDK for JavaScript (v3)

AWS Step Functions と AWS SDK for JavaScript を使用して AWS サーバーレスワークフローを作成する方法を示します。各ワークフローステップは、AWS Lambda 関数を使用して実装されます。

Lambda はサーバーをプロビジョニングまたは管理サーバーなしでもコードを実行し、有効にするコンピューティングサービスです。Step Functions は、ビジネス上重要なアプリケーションを構築するために Lambda 関数と他の AWS のサービスを組み合わせることができるサーバーレスオーケストレーションサービスです。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- DynamoDB
- Lambda
- Amazon SES
- ステップ関数

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

スケジュールされたイベントを使用した Lambda 関数の呼び出し

次のコード例は、Amazon EventBridge スケジュールイベントによって呼び出される AWS Lambda 関数を作成する方法を示しています。

Java

SDK for Java 2.x

AWS Lambda 関数を呼び出す Amazon EventBridge スケジュールイベントを作成する方法を示します。cron 式を使用して Lambda 関数が呼び出されるタイミングをスケジュールするように EventBridge を設定します。この例では、Lambda Java ランタイム API を使用して Lambda 関数を作成します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、年間の記念日に従業員を祝福するモバイルテキストメッセージを従業員に送信するアプリを作成する方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

AWS Lambda 関数を呼び出す Amazon EventBridge スケジュールイベントを作成する方法を示します。cron 式を使用して Lambda 関数が呼び出されるタイミングをスケジュールするように EventBridge を設定します。この例では、Lambda JavaScript ランタイム API を使用して Lambda 関数を作成します。この例では、特定のユースケースを実行する異なる AWS サービスを呼び出します。この例では、年間の記念日に従業員を祝福するモバイルテキストメッセージを従業員に送信するアプリを作成する方法を示します。

完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例は、AWS SDK for JavaScript v3 デベロッパーガイドでも使用できます。

この例で使用されているサービス

- DynamoDB

- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

以下の例では、スケジュールした Amazon EventBridge イベントのターゲットとして、AWS Lambda 関数を登録する方法を示しています。Lambda ハンドラーは、後で取得するために Amazon CloudWatch Logs にわかりやすいメッセージと完全なイベントデータを書き込みます。

- Lambda 関数をデプロイします。
- EventBridge スケジュールイベントを作成し、Lambda 関数をターゲットにします。
- EventBridge に Lambda 関数を呼び出す許可を付与します
- CloudWatch Logs から最新のデータを出力して、スケジュールされた呼び出しの結果を表示しています。
- デモ中に作成されたすべてのリソースをクリーンアップします。

この例は GitHub で最もよく確認できます。完全なソースコードとセットアップおよび実行の手順については、GitHub で完全な例を参照してください。

この例で使用されているサービス

- CloudWatch ログ
- EventBridge
- Lambda

AWS SDK デベロッパーガイドとコード例の完全なリストについては、「AWS SDK での Lambda を使用する」を参照してください。このトピックには、使用開始方法に関する情報と、以前の SDK バージョンの詳細も含まれています。

Lambda クォータ

> **Important**
>
> 新規の AWS アカウント では、同時実行性とメモリクォータが低くなっています。これらのクォータは、使用状況に応じて AWS が自動的に引き上げます。

コンピューティングとストレージ

Lambda では、関数の実行と保存に使用できるコンピューティングおよびストレージリソースの量に対してクォータを設定します。同時実行とストレージのクォータは、AWS リージョン ごとに適用されます。Elastic Network Interface (ENI) クォータは、リージョンにかかわらず、仮想プライベートクラウド (VPC) ごとに適用されます。次のクォータは、デフォルト値から引き上げることができます。詳細については、「Service Quotas ユーザーガイド」の「クォータの引き上げのリクエスト」を参照してください。

リソース	デフォルトのクォータ	引き上げることができる最大
同時実行数	1,000	数万
アップロードされた関数 (.zip ファイルアーカイブ) とレイヤーのストレージ。各関数バージョンとレイヤーバージョンは、ストレージを消費します。 コードストレージ管理のベストプラクティスについては、Serverless Land の「Lambda コードストレージのモニタリング」を参照してください。	75 GB	Terabytes
コンテナイメージとして定義された関数のストレージ。これらのイメージは Amazon ECR に保存されます。	「Amazon ECR サービスクォータ」を参照してください。	

リソース	デフォルトのクォータ	引き上げることができる最大
仮想プライベートクラウド (VPC) ごとの Elastic Network Interfaces **Note** このクォータは、Amazon Elastic File System (Amazon EFS) などの他のサービスと共有されます。「Amazon VPC クォータ」を参照してください。	250	数千

同時実行および Lambda がトラフィックに応じて関数の同時実行数をスケーリングする方法の詳細については、「Lambda 関数のスケーリングについて」を参照してください。

関数の設定、デプロイ、実行

関数の設定、デプロイ、実行には、次のクォータが適用されます。特に明記されていない限り、変更できません。

Note
Lambda のドキュメント、ログメッセージ、およびコンソールでは、1024 KB を示すのに (MiB ではなく) MB を使用します。

リソース	クォータ
関数のメモリ割り当て	128 MB から 10,240 MB まで、1 MB 単位で増加できます。 注意: Lambda は、設定されたメモリの量に比例して CPU パワーを割り当てます。[メモリ (MB)] 設定を使用し

リソース	クォータ
	て、関数に割り当てられたメモリと CPU パワーを増減できます。1,769 MB の場合、1 つの vCPU に相当します。
関数タイムアウト	900 秒 (15 分)
関数の環境変数	4 KB (関数に関連付けられたすべての環境変数)
関数リソースベースのポリシー	20 KB
関数レイヤー	5 つのレイヤー
関数の同時実行スケーリング制限	各関数について、10 秒ごとに 1,000 の実行環境が必要です。
呼び出しペイロード (リクエストとレスポンス)	リクエストとレスポンスにそれぞれ 6 MB (同期) ストリーミング応答ごとに 20 MB (同期。ストリーミング応答のペイロードサイズは、デフォルト値から増やすことができます。詳しくは AWS Support にお問い合わせください。) 256 KB (非同期) リクエスト行とヘッダー値の合計サイズは 1 MB
ストリーミングレスポンスの帯域幅	関数のレスポンスでは、最初の 6 MB には上限がありません 6 MB を超えるレスポンスの場合、残りのレスポンスは 2 Mbps です

リソース	クォータ
デプロイパッケージ (.zip ファイルアーカイブ) のサイズ	50 MB (zip 圧縮済み、直接アップロード) 250 MB (解凍後) このクォータは、レイヤーやカスタムランタイムなど、アップロードするすべてのファイルに適用されます。 3 MB (コンソールエディタ)
コンテナイメージの設定サイズ	16 KB
コンテナイメージのコードパッケージサイズ	10 GB (非圧縮のイメージの最大サイズ、すべてのレイヤーを含む)
テストイベント (コンソールエディタ)	10
/tmp ディレクトリのストレージ	512 MB〜10,240 MB、1 MB 刻み
ファイルディスクリプタ	1,024
実行プロセス/スレッド	1,024

Lambda API リクエスト

次のクォータは Lambda API リクエストに関連付けられています。

リソース	クォータ
リージョンごとの関数あたりの呼び出しリクエスト (同期)	実行環境の各インスタンスは、1 秒あたり最大 10 件のリクエストを処理できます。つまり、呼び出しの合計上限数は、同時実行数上限の 10 倍になります。「Lambda 関数のスケー

リソース	クォータ
	リングについて」を参照してください。
リージョンごとの関数あたりの呼び出しリクエスト (非同期)	実行環境の各インスタンスは、無制限の数のリクエストを処理できます。つまり、呼び出しの合計上限数は、関数に利用できる同時実行数のみに基づく数になります。「Lambda 関数のスケーリングについて」を参照してください。
関数のバージョンまたはエイリアスあたりの呼び出しリクエスト頻度 (リクエスト数/秒)	10 x 割り当て済みのプロビジョニングされた同時実行数 Note このクォータは、プロビジョニングされた同時実行を使用する関数にのみ適用されます。
GetFunction API リクエスト数	1 秒あたり 100 リクエスト。増やすことはできません。
GetPolicy API リクエスト数	1 秒あたり 15 リクエスト。増やすことはできません。
残りのコントロールプレーン API リクエスト数 (呼び出し、GetFunction、および GetPolicy リクエストを除く)	すべての API で 1 秒あたり 15 リクエスト (API ごとに 1 秒あたり 15 リクエストではありません) 。増やすことはできません。

その他のサービス

AWS Identity and Access Management (IAM)、Amazon CloudFront (Lambda@Edge)、Amazon Virtual Private Cloud (Amazon VPC) などの他のサービスのクォータが Lambda 関数に影響を及ぼすことがあります。詳細については、「Amazon Web Services 全般のリファレンス」と「他の AWS サービスからのイベントを使用した Lambda の呼び出し」の「AWS のサービスクォータ」を参照してください。

ドキュメント履歴

次の表は、AWS Lambda 開発者ガイドに対する 2018 年 5 月以降の重要な変更点をまとめたものです。このドキュメントの更新に関する通知については、「RSS フィード」にサブスクライブできます。

変更	説明	日付
新しいリージョンでの SnapStart のサポート	ALambda SnapStart が、欧州 (スペイン)、欧州 (チューリッヒ)、アジアパシフィック (メルボルン)、アジアパシフィック (ハイデラバード)、および中東 (UAE) のリージョンで利用できるようになりました。	2024-01-12
AWS 管理ポリシーの更新	Service Quotas が既存の AWS マネージドポリシー (`AWSLambdaVPCAccessExecutionRole`) を更新しました。	2024 年 1 月 5 日
Python 3.12 ランタイム	Lambda では、マネージドランタイムおよびコンテナベースイメージとして Python 3.12 をサポートするようになりました。詳細については、「AWS コンピューティングブログ」の「Python 3.12 runtime now available in AWS Lambda」を参照してください。	2023 年 12 月 14 日
Java 21 ランタイム	Lambda では、マネージドランタイムおよびコンテナベースイメージ (`java21`) として	2023 年 11 月 16 日

	Java 21 をサポートするようになりました。	
Node.js 20.x ランタイム	Lambda では、マネージドランタイムおよびコンテナベースイメージ (`nodejs20.x`) として Node.js 20 をサポートするようになりました。詳細については、AWS コンピューティングブログの「Node.js 20.x runtime now available in AWS Lambda」を参照してください。	2023 年 11 月 14 日
provided.al2023 ランタイム	Lambda では Amazon Linux 2023 をマネージドランタイムおよびコンテナベースイメージとしてサポートするようになりました。詳細については、AWS コンピューティングブログの「Introducing the Amazon Linux 2023 runtime for AWS Lambda」を参照してください。	2023 年 11 月 9 日
デュアルスタックサブネットに対する IPv6 サポート	Lambda は、デュアルスタックサブネットへのアウトバウンド IPv6 トラフィックをサポートするようになりました。詳細については、「IPv6 support」を参照してください。	2023 年 10 月 12 日

サーバーレス関数とアプリケーションのテスト	クラウドでサーバーレス機能のデバッグとテストを自動化する手法について学びます。Python と Typescript 言語のセクションにテストの章とリソースが追加されました。詳細については、「サーバーレス関数とアプリケーションのテスト」を参照してください。	2023 年 6 月 16 日
Ruby 3.2 ランタイム	Lambda で、.Ruby 3.2 向けの新しいランタイムのサポートを開始しました。詳細については、「Ruby を使用した Lambda 関数のビルド」を参照してください。	2023 年 6 月 7 日
レスポンスストリーミング	Lambda は関数からのストリーミングレスポンスをサポートするようになりました。詳細については、「ストリームレスポンスに Lambda 関数を設定する」を参照してください。	2023 年 4 月 6 日
非同期呼び出しメトリクス	Lambda は非同期呼び出しメトリクスをリリースします。詳細については、「非同期呼び出しメトリクス」を参照してください。	2023 年 2 月 9 日

ランタイムのバージョンコントロール	Lambda は、セキュリティ更新、バグ修正、および新機能が含まれる新しいランタイムバージョンをリリースします。関数が新しいランタイムバージョンに更新されるタイミングを制御できるようになりました。詳細については、「Lambda ランタイムの更新」を参照してください。	2023 年 1 月 23 日
Lambda SnapStart	Lambda SnapStart を使用して、追加のリソースをプロビジョニングしたり、複雑なパフォーマンス最適化を実装したりすることなく、Java 関数の起動時間を短縮します。詳細については、「Lambda SnapStart による起動パフォーマンスの向上」を参照してください。	2022 年 11 月 28 日
Node.js 18 ランタイム	Lambda で、Node.js 18 の新しいランタイムがサポートされるようになりました。Node.js 18 は Amazon Linux 2 を使用しています。詳細については、「Node.js による Lambda 関数のビルド」を参照してください。	2022 年 11 月 18 日

lambda:SourceFunctionArn 条件キー	`lambda:SourceFunctionArn` 条件キーは、AWS リソース対するアクセスを、Lambda 関数の ARN に基づきフィルタリングします。詳細については、「Working with Lambda execution environment credentials」(Lambda実行環境の認証情報の使用) を参照してください。	2022 年 7 月 1 日
Node.js 16 ランタイム	Lambda で、Node.js 16 の新しいランタイムがサポートされるようになりました。Node.js 16 は Amazon Linux 2 を使用しています。詳細については、「Node.js による Lambda 関数のビルド」を参照してください。	2022 年 5 月 11 日
Lambda 関数 URL	Lambda において、関数専用の HTTP(S) エンドポイントを指定するために、関数 URL を使用できるようになりました。詳細については、「Lambda function URLs」(Lambda 関数URL) を参照してください。	2022 年 4 月 6 日
AWS Lambda コンソール上で共有されたテストイベント	Lambda が、同じ AWS アカウント内の他のユーザーとのテストイベントの共有をサポートするようになりました。詳細については、「コンソールでの Lambda 関数のテスト」を参照してください。	2022 年 3 月 16 日

リソースベースのポリシー内の PrincipalOrgId	Lambda で、AWS Organizations 内にある組織にアクセス許可を付与できるようになりました。詳細については、「AWS Lambda のリソースベースのポリシーを使用する」を参照してください。	2022 年 3 月 11 日
.NET 6 ランタイム	Lambda で、.NET 6 向けの新しいランタイムのサポートを開始しました。詳細については、「Lambda ランタイム」を参照してください。	2022 年 2 月 23 日
Kinesis、DynamoDB、および Amazon SQS イベントソースでのイベントフィルタリング	Lambda で、Kinesis、DynamoDB、および Amazon SQS のイベントソースに対する、イベントフィルタリングがサポートされました。詳細については、「Lambda のイベントフィルタリング」を参照してください。	2021 年 11 月 24 日
Amazon MSK とセルフマネージド Apache Kafka イベントソースの mTLS 認証	Lambda が Amazon MSK とセルフマネージド Apache Kafka イベントソースの mTLS 認証をサポートするようになりました。詳細については、「Amazon MSK で Lambda を使用する」を参照してください。	2021 年 11 月 19 日

Graviton2 の Lambda	Lambda は arm64 アーキテクチャを使用する関数で Graviton2 をサポートするようになりました。詳細については、Lambda 命令セットアーキテクチャを参照してください。	2021 年 9 月 29 日
Python 3.9 ランタイム	Lambda では、Python 3.9 の新しいランタイムのサポートを開始しました。詳細については、「Lambda ランタイム」を参照してください。	2021 年 8 月 16 日
Node.js、Python、および Java 向けの新しいランタイムバージョン	Node.js、Python、および Java の新しいランタイムバージョンを利用できます。詳細については、「Lambda ランタイム」を参照してください。	2021 年 7 月 21 日
Lambda でのイベントソースとしての RabbitMQ のサポート	Lambda は、イベントソースとして RabbitMQ の Amazon MQ のサポートを開始しました。Amazon MQ はクラウド内のメッセージブローカーを容易に設定および運用できる、Apache ActiveMQ および RabbitMQ 向けのマネージドメッセージブローカーサービスです。詳細については、「Amazon MQ で Lambda を使用する」を参照してください。	2021 年 7 月 7 日

Lambda のセルフマネージド Kafka 向けの SASL/PLAIN 認証	SASL/PLAIN は、Lambda のセルフマネージド Kafka イベントソースをサポートする認証メカニズムになりました。セルフマネージド Kafka クラスターで SASL/PLAIN を既に使用しているお客様は、Lambda を使用して、認証方法を変更することなく、コンシューマーアプリケーションを簡単に構築できるようになりました。詳細については、「セルフマネージド Apache Kafka で Lambda を使用する」を参照してください。	2021 年 6 月 29 日
Lambda の拡張 API	Lambda 拡張機能の一般的な可用性。拡張機能を使用して Lambda 関数を補強できます。Lambda パートナーが提供する拡張機能を使用することも、独自の Lambda 拡張機能を作成することもできます。詳細については、「Lambda 拡張機能 API」を参照してください。	2021 年 5 月 24 日
Lambda コンソールの新しい操作機能	Lambda コンソールが再設計され、パフォーマンスと一貫性が向上しました。	2021 年 3 月 2 日

Node.js 14 ランタイム	Lambda は、Node.js 14 の新しいランタイムのサポートを開始しました。Node.js 14 は Amazon Linux 2 を使用します。詳細については、「Node.js による Lambda 関数のビルド」を参照してください。	2021 年 1 月 27 日
Lambda のコンテナイメージ	Lambda は、コンテナイメージとして定義された関数のサポートを開始しました。Lambda に備わった、アプリケーションを構築する上での俊敏性と運用のシンプルさに、コンテナツールの柔軟性が加わります。詳細については、「Lambda でコンテナイメージを使用する」を参照してください。	2020 年 12 月 1 日

Lambda 関数のコード署名	Lambda が、コード署名のサポートを開始しました。管理者は Lambda 関数を設定することで、デプロイの際に署名付きコードのみを受け入れることができます。この署名は、Lambda によりチェックされ、コードが変更または改ざんされていないことが確認されます。さらに、コードが信頼できるデベロッパーにより署名されていることも、デプロイを受け入れる前に Lambda により確認されます。詳細については、「Lambda のコード署名の設定」を参照してください。	2020 年 11 月 23 日
プレビュー: Lambda ランタイムログ API	Lambda が、ランタイムログ API のサポートを開始しました。Lambda 拡張機能はログ API を使用して、実行環境でログストリーミングをサブスクライブすることができます。詳細については、「Lambda ランタイムログ API 」を参照してください。	2020 年 11 月 12 日

Amazon MQ 用の新しいイベントソース	Lambda が、イベントソースとして Amazon MQ のサポートを開始しました。Lambda 関数を使用して、Amazon MQ メッセージブローカーからのレコードを処理できます。詳細については、「Amazon MQ で Lambda を使用する」を参照してください。	2020 年 11 月 5 日
プレビュー: Lambda の拡張 API	Lambda 拡張機能を使用して、Lambda 関数を補強できます。Lambda パートナーが提供する拡張機能を使用することも、独自の Lambda 拡張機能を作成することもできます。詳細については、「Lambda 拡張機能 API」を参照してください。	2020 年 10 月 8 日
AL2 での Java 8 およびカスタムランタイムのサポート	Lambda が Amazon Linux 2 で Java 8 およびカスタムランタイムのサポートを開始しました。詳細については、「Lambda ランタイム」を参照してください。	2020 年 8 月 12 日
Amazon Managed Streaming for Apache Kafka 用の新しいイベントソース	Lambda が、イベントソースとして Amazon MSK のサポートを開始しました。Amazon MSK で Lambda 関数を使用して、Kafka トピックのレコードを処理します。詳細については、「Amazon MSK で Lambda を使用する」を参照してください。	2020 年 8 月 11 日

Amazon VPC 設定用の IAM 条件キー	VPC 設定で Lambda 固有の条件キーを使用できるようになりました。例えば、組織内のすべての関数を VPC に接続するように要求できます。また、関数のユーザーに対して使用を許可または拒否するサブネットとセキュリティグループを指定することもできます。詳細については、「IAM 関数用の VPC の設定」を参照してください。	2020 年 8 月 10 日
Kinesis HTTP/2 ストリームコンシューマーの同時実行設定	拡張ファンアウト (HTTP/2 ストリーム) を使用する Kinesis コンシューマーに対して、ParallelizationFactor、MaximumRetryAttempts、MaximumRecordAgeInSeconds、DestinationConfig、BisectBatchOnFunctionError の同時実行設定を使用できるようになりました。詳細については、「Amazon Kinesis でAWS Lambdaを使用する」を参照してください。	2020 年 7 月 7 日

Kinesis HTTP/2 ストリームコンシューマー用のバッチウィンドウ	HTTP/2 ストリームのバッチウィンドウ (MaximumBatchingWindowInSeconds) を設定できるようになりました。Lambda は完全なバッチを収集するまで、またはバッチウィンドウの期限が切れるまで、ストリーミングからレコードを読み取ります。詳細については、「Amazon Kinesis でAWS Lambdaを使用する」を参照してください。	2020 年 6 月 18 日
Amazon EFS ファイルシステムのサポート	Amazon EFS ファイルシステムを Lambda 関数に接続して、共有ネットワークファイルにアクセスできるようになりました。詳細については、「Lambda 関数のファイルシステムアクセスの設定」を参照してください。	2020 年 6 月 16 日
Lambda コンソールでの AWS CDK サンプルアプリケーション	Lambda コンソールに TypeScript の AWS Cloud Development Kit (AWS CDK) を使用するサンプルアプリケーションが含まれるようになりました。AWS CDK は、TypeScript、Python、Java、または .NET でアプリケーションリソースを定義するためのフレームワークです。	2020 年 6 月 1 日

AWS Lambda での .NET Core 3.1.0 ランタイムのサポート	AWS Lambda は .NET Core 3.1.0 ランタイムをサポートするようになりました。詳細については、「.NET Core CLI」を参照してください。	2020 年 3 月 31 日
API Gateway HTTP API のサポート	API Gateway で Lambda を使用する (HTTP API のサポートを含む) ドキュメントを更新および拡張しました。AWS CloudFormation で API と関数を作成するサンプルアプリケーションを追加しました。詳細については、「Amazon API Gateway で Lambda を使用する」を参照してください。	2020 年 3 月 23 日
Ruby 2.7	新しいランタイムが Ruby 2.7、ruby2.7 で利用可能になりました。これは、Amazon Linux 2 を使用する最初の Ruby ランタイムです。詳細については、「Ruby による Lambda 関数のビルド」を参照してください。	2020 年 2 月 19 日

同時実行に関するメトリクス	Lambda は、すべての関数、エイリアス、バージョンの ConcurrentExecutions メトリクスのレポートを開始しました。このメトリクスのグラフは、関数のモニタリングページで表示できます。以前は、ConcurrentExecutions は、アカウントレベルでのみ報告され、予約された同時実行を使用する関数に対してのみ報告されていました。詳細については、「AWS Lambda 関数メトリクス」を参照してください。	2020 年 2 月 18 日

関数のステータスに関する更新	関数の状態は、すべての関数にデフォルトで適用されるようになりました。関数を VPC に接続すると、Lambda は共有 Elastic Network Interface を作成します。これにより、追加のネットワークインターフェイスを作成することなく、関数のスケールアップができます。この間は、設定の更新やバージョンの発行など、関数に対する追加のオペレーションは実行できません。場合によっては、呼び出しも影響を受けます。関数の現在の状態に関する詳細は、Lambda API から入手できます。 この更新は、段階的にリリースされます。詳細については、AWS コンピューティングブログの「Updated Lambda states lifecycle for VPC networking」を参照してください。状態の詳細については、「AWS Lambda 関数の状態」を参照してください。	2020 年 1 月 24 日

関数設定 API 出力に関する更新	VPC に接続する関数の StateReasonCode (InvalidSubnet、InvalidSecurityGroup) および LastUpdateStatusReasonCode (SubnetOutOfIPAddresses、InvalidSubnet、InvalidSecurityGroup) に理由コードを追加しました。状態の詳細については、「AWS Lambda 関数の状態」を参照してください。	2020 年 1 月 20 日
プロビジョニングされた同時実行数	プロビジョニングされた同時実行数を関数バージョンまたはエイリアスに割り当てることができるようになりました。プロビジョニングされた同時実行数により、レイテンシーの変動なしに関数を拡張できます。詳細については、「Lambda 関数の同時実行数の管理」を参照してください。	2019 年 12 月 3 日
データベースプロキシ作成	Lambda コンソールを使用して、Lambda 関数のデータベースプロキシを作成できるようになりました。データベースプロキシを使用すると、データベース接続を使い果たすことなく、関数の同時実行レベルを上げることができます。詳細については、「Lambda 関数のデータベースアクセスの設定」を参照してください。	2019 年 12 月 3 日

期間メトリクスでのパーセンタイルのサポート	パーセンタイルに基づいて期間メトリクスをフィルタ処理できるようになりました。詳細については、「AWS Lambda メトリクス」を参照してください。	2019 年 11 月 26 日
ストリームイベントソースにおける同時実行数の増加	DynamoDB Streams と Kinesis ストリームのイベントソースマッピングの新しいオプションにより、各シャードの複数のバッチを一度に処理できます。シャードごとの同時バッチの数を増やすと、関数の同時実行数はストリームのシャード数の最大 10 倍になる場合があります。詳細については、「Lambda イベントソースマッピング」を参照してください。	2019 年 11 月 25 日

関数ステータス	関数を作成または更新すると、その関数をサポートするリソースが Lambda によってプロビジョニングされている間、その関数は保留状態になります。関数を VPC に接続すると、Lambda は、関数が呼び出されたときにネットワークインターフェースを作成せずに、すぐに共有 Elastic Network Interface を作成できます。その結果、VPC に接続された関数のパフォーマンスが向上しますが、オートメーションの更新が必要になる場合があります。詳細については、「AWS Lambda 関数の状態」を参照してください。	2019 年 11 月 25 日
非同期呼び出しでのエラー処理オプション	非同期呼び出しに新しい設定オプションを使用できるようになりました。Lambda の再試行の制限、最大イベント有効期間を設定できます。詳細については、「非同期呼び出しのエラー処理の設定」を参照してください。	2019 年 11 月 25 日

ストリームイベントソースでのエラー処理	ストリームから読み取るイベントソースマッピングに新しい設定オプションを使用できるようになりました。DynamoDB Streams と Kinesis ストリームのイベントソースマッピングを設定して再試行を制限し、最大レコード有効期間を設定できます。エラーが発生した場合は、再試行する前にバッチを分割し、失敗したバッチの呼び出しレコードをキューまたはトピックに送信するように、イベントソースマッピングを設定できます。詳細については、「Lambda イベントソースマッピング」を参照してください。	2019 年 11 月 25 日
非同期呼び出しの送信先	非同期呼び出しのレコードを別のサービスに送信するように、Lambda を設定できるようになりました。呼び出しレコードには、イベント、コンテキスト、関数のレスポンスに関する詳細が含まれます。呼び出しレコードを SQS キュー、SNS トピック、Lambda 関数、EventBridge イベントバスに送信できます。詳細については、「非同期呼び出しの送信先の設定」を参照してください。	2019 年 11 月 25 日

Node.js、Python、および Java 向けの新しいランタイム	新しいランタイムが Node.js 12、Python 3.8、および Java 11 で利用できます。詳細については、「Lambda ランタイム」を参照してください。	2019 年 11 月 18 日
Amazon SQS イベントソースでの FIFO キューのサポート	FIFO (先入れ先出し) キューから読み取るイベントソースマッピングを作成できるようになりました。以前は、標準キューのみがサポートされていました。詳細については、「Amazon SQS で Lambda を使用する」を参照してください。	2019 年 11 月 18 日
Lambda コンソールを使用したアプリケーションの作成	Lambda コンソールでのアプリケーションの作成が正式リリースされました。手順については、「Managing applications in the Lambda console」を参照してください。	2019 年 10 月 31 日

Lambda コンソールを使用したアプリケーションの作成 (ベータ)	Lambda コンソールの統合された継続的デリバリーパイプラインを使用して、Lambda アプリケーションを作成できるようになりました。コンソールには、独自のプロジェクトの開始点として使用できるサンプルアプリケーションが用意されています。ソース管理のために AWS CodeCommit と GitHub のいずれかを選択します。変更をリポジトリにプッシュするたびに、含まれているパイプラインによって変更が自動的にビルドおよびデプロイされます。手順については、「Managing applications in the Lambda console」を参照してください。	2019 年 10 月 3 日

VPC 接続された関数におけるパフォーマンスの向上	Lambda で、Virtual Private Cloud (VPC) サブネットのすべての関数で共有される新しいタイプの Elastic Network Interface が使用されるようになりました。関数を VPC に接続すると、Lambda は選択したセキュリティグループとサブネットの組み合わせごとにネットワークインターフェイスを作成します。共有ネットワークインターフェイスが利用可能になると、関数のスケールアップ時に追加のネットワークインターフェイスを作成する必要がなくなります。これにより、起動時間が大幅に短縮されます。詳細については、「VPC 内のリソースにアクセスできるように Lambda 関数を設定する」を参照してください。	2019 年 9 月 3 日
ストリームバッチ設定	Amazon DynamoDB と Amazon Kinesis イベントソースマッピングのバッチウィンドウを設定できるようになりました。バッチ全体が使用可能になるまで受信レコードをバッファするように、最大 5 分のバッチウィンドウを設定します。これにより、ストリームがアクティブでない場合に関数が呼び出される回数が減ります。	2019 年 8 月 29 日

CloudWatch Logs Insights の統合	Lambda コンソールのモニタリングページに Amazon CloudWatch Logs Insights からのレポートが含まれるようになりました。詳細については、「AWS Lambda コンソールで関数をモニタリングする」を参照してください。	2019 年 6 月 18 日
Amazon Linux 2018.03	Lambda の実行環境が Amazon Linux 2018.03 を使用するように更新されています。詳細については、「実行環境」を参照してください。	2019 年 5 月 21 日
Node.js 10	Node.js 10、nodejs10.x 用の新しいランタイムが利用可能になりました。このランタイムは Node.js 10.15 を使用し、最新のポイントリリースの Node.js 10 で定期的に更新されます。また、Node.js 10 は Amazon Linux 2 を使用する最初のランタイムです。詳細については、「Node.js による Lambda 関数のビルド」を参照してください。	2019 年 5 月 13 日

GetLayerVersionByArn API	GetLayerVersionByArn API を使用して、バージョン ARN を入力としたレイヤーバージョン情報をダウンロードします。GetLayerVersion と比較すると、解析してレイヤー名およびバージョン番号を取得する代わりに、GetLayerVersionByArn では ARN を直接使用することができます。	2019 年 4 月 25 日
Ruby	AWS Lambda で新しいランタイムの Ruby 2.5 がサポートされるようになりました。詳細については、「Ruby による Lambda 関数のビルド」を参照してください。	2018 年 11 月 29 日
レイヤー	Lambda レイヤーにより、ライブラリ、カスタムランタイム、その他の依存関係を、関数コードとは別にパッケージ化してデプロイすることができます。レイヤーを他のアカウント、または世界のすべてのユーザーと共有します。詳細については、「Lambda レイヤー」を参照してください。	2018 年 11 月 29 日
カスタムランタイム	カスタムランタイムを構築して、任意のプログラミング言語で Lambda 関数を実行します。詳細については、「カスタム Lambda ランタイム」を参照してください。	2018 年 11 月 29 日

Application Load Balancer のトリガー	Elastic Load Balancing は、Application Load Balancer のターゲットとして Lambda 関数のサポートを開始しました。詳細については、「Application Load Balancer で Lambda を使用する」を参照してください。	2018 年 11 月 29 日
Kinesis HTTP/2 ストリームコンシューマーのトリガーとしての使用	Kinesis HTTP/2 データストリームコンシューマーを使用してイベントを AWS Lambda に送信できます。ストリームコンシューマーは、各シャードの専用の読み取りスループットをデータストリーム内に配置し、HTTP/2 を使用してレイテンシーを最小限に抑えています。詳細については、「Kinesis で Lambda を使用する」を参照してください。	2018 年 11 月 19 日
「Python 3.7」	AWS Lambda で新しいランタイムの Python 3.7 がサポートされるようになりました。詳細については、「Python を使用した Lambda 関数のビルド」を参照してください。	2018 年 11 月 19 日
非同期の関数呼び出しにおけるペイロード制限の引き上げ	非同期呼び出しの最大ペイロードサイズが 128 KB から 256 KB に増加し、Amazon SNS トリガーからの最大メッセージサイズと一致しました。詳細については、「Lambda のクォータ」を参照してください。	2018 年 11 月 16 日

AWS GovCloud (米国東部) リージョン	AWS Lambda が AWS GovCloud (米国東部) リージョンで利用できるようになりました。	2018 年 11 月 12 日
別のデベロッパーガイドへの AWS SAM トピックの移動	多数のトピックが、AWS Serverless Application Model (AWS SAM) を使用したサーバーレスアプリケーションの構築に重点を置いていました。これらのトピックは、「AWS Serverless Application Model デベロッパーガイド」に移動されました。	2018 年 10 月 25 日
コンソールでの Lambda アプリケーションの表示	Lambda アプリケーションのステータスは、Lambda コンソールの [Applications (アプリケーション)] ページで確認できます。このページには、AWS CloudFormation スタックのステータスが表示されます。ここには、スタックのリソースに関する情報を表示できるページへのリンクが含まれています。また、アプリケーションのメトリクス集約を表示し、カスタムモニタリングダッシュボードを作成することもできます。	2018 年 10 月 11 日
関数実行のタイムアウト制限	長時間実行される関数を許可すると、設定可能な最大実行タイムアウトは 5 分から 15 分以内に増加します。詳細については、「Lambda の制限事項」を参照してください。	2018 年 10 月 10 日

AWS Lambda における PowerShell Core 言語のサポート	AWS Lambda では PowerShell Core 言語をサポートするようになりました。詳細については、「PowerShell で Lambda 関数の作成用モデルをプログラミングする」を参照してください。	2018 年 9 月 11 日
AWS Lambda における .NET Core 2.1.0 ランタイムのサポート	AWS Lambda は .NET Core 2.1.0 ランタイムをサポートするようになりました。詳細については、「.NET Core CLI」を参照してください。	2018 年 7 月 9 日
更新を RSS で今すぐ入手可能	RSS フィードを購読して、このガイドのリリースをフォローできるようになりました。	2018 年 7 月 5 日
イベントソースとしての Amazon SQS のサポート	AWS Lambda は、イベントソースとして Amazon Simple Queue Service (Amazon SQS) のサポートを開始しました。詳細については、「Lambda 関数の呼び出し」を参照してください。	2018 年 28 月 6 日
中国 (寧夏) リージョン	AWS Lambda が中国 (寧夏) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2018 年 6 月 28 日

以前の更新

次の表に、2018 年 6 月以前の AWS Lambda 開発者ガイドの各リリースにおける重要な変更点を示します。

変更	説明	日付
Node.js ランタイム 8.10 ランタイムのサポート	AWS Lambda で Node.js ランタイムバージョン 8.10 がサポートされるようになりました。詳細については、「」を参照してくださいNode.js による Lambda 関数の構築	2018 年 4 月 2 日
関数およびエイリアスリビジョン ID	AWS Lambda が関数バージョンおよびエイリアスのリビジョン ID をサポートするようになりました。この ID を使用して、関数バージョンあるいはエイリアスリソースを更新するときに、条件付き更新を追跡して適用できます。	2018 年 1 月 25 日
Go と .NET 2.0 におけるランタイムのサポート	AWS Lambda に Go と .NET 2.0 のランタイムへのサポートが追加されました。詳細については、「Go による Lambda 関数の構築」および「C# による Lambda 関数の構築」を参照してください。	2018 年 1 月 15 日
コンソールの再設計	AWS Lambda で新しい Lambda コンソールを導入し、エクスペリエンスを簡素化しました。また、Cloud9 Code Editor を追加して、関数コードのデバッグと修正の機能を強化しました。詳細については、「Lambda コンソールエディタを使用したコードの編集」を参照してください。	2017 年 11 月 30 日
個々の関数に対する同時実行数の制限の設定	AWS Lambda で、個々の関数に対して同時実行数の制限を設定できるようになりました。詳細については、「関数に対する予約済み同時実行数の設定」を参照してください。	2017 年 11 月 30 日
エイリアスによるトラフィックの移行	AWS Lambda で、エイリアスによるトラフィックの移行がサポートされるようになりました。詳細については、「Lambda 関数のローリングデプロイの作成」を参照してください。	2017 年 11 月 28 日

変更	説明	日付
コードの段階的なデプロイ	AWS Lambda で、Code Deploy を活用し、Lambda 関数の新しいバージョンを安全にデプロイできるようになりました。詳細については、「コードの段階的なデプロイ」を参照してください。	2017 年 11 月 28 日
中国 (北京) リージョン	AWS Lambda が中国 (北京) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2017 年 11 月 9 日
SAM Local の紹介	AWS Lambda で SAM Local (SAM CLI に名称変更) を導入しました。これは AWS CLI ツールであり、サーバーレスアプリケーションを Lambda ランタイムにアップロードする前にローカルで開発、テスト、分析するための環境を提供します。詳細については、「サーバーレスアプリケーションのテストとデバッグ」を参照してください。	2017 年 8 月 11 日
カナダ (中部) リージョン	AWS Lambda は、カナダ (中部) リージョンで使用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2017 年 6 月 22 日
南米 (サンパウロ) リージョン	AWS Lambda が南米 (サンパウロ) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2017 年 6 月 6 日
AWS Lambda では がサポートされています。AWS X-Ray	Lambda に X-Ray のサポートが導入されました。これにより Lambda アプリケーションでパフォーマンスの問題を検出、分析、最適化できます。詳細については、「AWS X-Ray を使用した Lambda 関数呼び出しの視覚化」を参照してください。	2017 年 4 月 19 日

変更	説明	日付
アジアパシフィック (ムンバイ) リージョン	AWS Lambda がアジアパシフィック (ムンバイ) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2017 年 3 月 28 日
AWS Lambda で Node.js ランタイム v6.10 がサポートされるようになりました。	AWS Lambda に Node.js ランタイム v6.10 のサポートが追加されました。詳細については、「」を参照してくださいNode.js による Lambda 関数の構築	2017 年 3 月 22 日
欧州 (ロンドン) リージョン	AWS Lambda が欧州 (ロンドン) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2017 年 2 月 1 日
AWS Lambda での .NET ランタイム、Lambda@Edge (プレビュー)、デッドレターキュー、サーバーレスアプリケーションの自動デプロイメントのサポート。	AWS Lambda に C# のサポートが追加されました。詳細については、「C# による Lambda 関数の構築」を参照してください。 Lambda@Edge では、CloudFront イベントに応じて、AWSエッジロケーションで Lambda 関数を実行することができます。詳細については、「 CloudFront Lambda@Edge AWS Lambdaで を使用する」を参照してください。	2016 年 12 月 3 日
サポートされているイベントソースとして Amazon Lex が AWS Lambda に追加されました。	Lambda および Amazon Lex を使用して、Slack や Facebook などの各種サービス用にチャットボットをすばやく構築できます。詳細については、「Amazon Lex で AWS Lambda を使用する」を参照してください。	2016 年 11 月 30 日
米国西部 (北カリフォルニア) リージョン	AWS Lambda が米国西部 (北カリフォルニア) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2016 年 11 月 21 日

変更	説明	日付
Lambda ベースのアプリケーションを作成およびデプロイし、Lambda 関数設定用の環境変数を使用するための、AWS SAM を導入しました。	AWS SAM: AWS SAM を使用して、サーバーレスアプリケーション内でリソースを表現するための構文を定義できるようになりました。アプリケーションをデプロイするには、必要なリソースを、AWS CloudFormation テンプレートファイル (JSON または YAML で記述された) の関連するアクセス許可ポリシーと共にアプリケーションの一部として指定します。デプロイアーティファクトをパッケージ化し、テンプレートをデプロイします。詳細については、「AWS Lambda アプリケーション」を参照してください。 環境変数: 環境変数を使用して、関数コード以外の Lambda 関数の設定を指定できます。詳細については、「Lambda 環境変数を使用したコードの値の設定」を参照してください。	2016 年 11 月 18 日
アジアパシフィック (ソウル) リージョン	AWS Lambda がアジアパシフィック (ソウル) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2016 年 8 月 29 日
アジアパシフィック (シドニー) リージョン	Lambda がアジアパシフィック (シドニー) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2016 年 6 月 23 日
Lambda コンソールの更新	Lambda コンソールが更新され、ロール作成プロセスが簡単になりました。	2016 年 6 月 23 日
AWS Lambda で Node.js ランタイム v4.3 がサポートされるようになりました。	AWS Lambda に Node.js ランタイム v4.3 のサポートが追加されました。詳細については、「Node.js による Lambda 関数の構築」を参照してください。	2016 年 4 月 07 日

変更	説明	日付
欧州 (フランクフルト) リージョン	Lambda が欧州 (フランクフルト) リージョンで利用可能になりました。Lambda リージョンおよびエンドポイントの詳細については、「AWS 全般のリファレンス」の「リージョンとエンドポイント」を参照してください。	2016 年 3 月 14 日
VPC サポート	VPC 内のリソースにアクセスできるよう Lambda 関数を設定できるようになりました。詳細については、「Lambda 関数に Amazon VPC 内のリソースへのアクセスを許可する」を参照してください。	2016 年 2 月 11 日
Lambda ランタイムが更新されました。	実行環境が更新されました。	2015 年 11 月 4 日
バージョニングサポート、Lambdafunctions の開発コード用の Python、スケジュールされたイベント、実行時間の増加	Python を使用した Lambda 関数コードの開発が可能になりました。詳細については、「Python による Lambda 関数の構築」を参照してください。 バージョニング: Lambda 関数の複数のバージョンを維持できます。バージョニング機能を使用して、異なる環境 (たとえば、開発、テスト、本稼働) で実行される Lambda 関数のバージョンを制御できます。詳細については、「Lambda 関数のバージョン」を参照してください。 スケジュールされているイベント: Lambda コンソールを使用して、スケジュールに基づいて定期的にコードを呼び出すように Lambda を設定できます。固定の間隔を指定する (時間、日、曜日の数字) ことも、Cron 式を指定することもできます。詳細については、「Amazon EventBridge スケジューラで Lambda を使用する」を参照してください。 実行時間の増加: Lambda 関数の実行時間を最長 5 分までセットアップできるようになりました。大容量データの取り込みやジョブの処理のような時間がかかる関数を実行できます。	2015 年 10 月 08 日

変更	説明	日付
DynamoDB Streams のサポート	DynamoDB Streams は一般公開され、DynamoDB が利用可能なすべてのリージョンでこれを使用することができます。テーブルに対して DynamoDB Streams を有効にして、テーブルのトリガーとして Lambda 関数を使用できます。トリガーは、DynamoDB テーブルに対して行われた更新に応じて行うカスタムアクションです。チュートリアル例については、「チュートリアル: Amazon DynamoDB Streams で AWS Lambda を使用する」を参照してください。	2015 年 7 月 14 日
Lambda が、REST 対応クライアントによる Lambda 関数呼び出しのサポートを開始しました。	これまでは、ウェブ、モバイル、IoT アプリケーションから Lambda 関数を呼び出すには、AWS SDK (AWS SDK for Java、AWS SDK for Android、AWS SDK for iOS など) が必要でした。Lambda は、Amazon API Gateway を使用して作成できるカスタマイズされた API により、REST 対応クライアントによる Lambda 関数呼び出しのサポートを開始しました。Lambda 関数のエンドポイント URL にリクエストを送信できます。エンドポイントでセキュリティを設定してオープンアクセスを許可したり、AWS Identity and Access Management (IAM) を利用してアクセスを許可したり、API キーを使用して他のユーザーによる Lambda 関数へのアクセスを計測したりできるようになりました。 「使用開始」の実習例については、「Amazon API Gateway エンドポイントを使用した Lambda 関数の呼び出し」を参照してください。 Amazon API Gateway の詳細については、https://aws.amazon.com/api-gateway/ を参照してください。	2015 年 7 月 09 日

変更	説明	日付
Lambda コンソールでは、簡単に Lambda 関数を作成およびテストするために、設計図が提供されるようになりました。	Lambda コンソールは、一連の設計図を提供します。各設計図には、Lambda 関数用に、簡単に Lambda ベースのアプリケーションを作成するために使用できる Lambda 関数のサンプルイベントソース設定とサンプルコードが用意されています。Lambda のすべての「使用開始」の実習で、設計図を使用するようになりました。詳細については、「Lambda の開始方法」を参照してください。	2015 年 7 月 09 日
Lambda は、Lambda 関数を作成するために Java のサポートを開始しました。	Java で Lambda コードを記述できるようになりました。詳細については、「Java による Lambda 関数の構築」を参照してください。	2015 年 6 月 15 日
Lambda は、Lambda 関数の作成または更新時に、Amazon S3 オブジェクトを関数の .zip として指定するサポートを開始しました。	Lambda 関数デプロイパッケージ (.zip ファイル) を、Lambda 関数を作成するのと同じリージョンで Amazon S3 バケットにアップロードできます。次に、Lambda 関数を作成または更新するときに、バケット名とオブジェクトキー名を指定できます。	2015 年 5 月 28 日

変更	説明	日付
Lambda は、モバイルバックエンド用のサポートを追加して一般公開されるようになりました。	Lambda は、本稼働環境用に一般公開されるようになりました。このリリースでは、インフラストラクチャをプロビジョニングまたは管理することなく、自動的にスケーリングする Lambda を使用して、モバイル、タブレット、IoT バックエンドを簡単に構築できる新機能も導入されています。Lambda は、リアルタイム (同期) および非同期イベント両方のサポートを開始しました。その他の機能には、簡単なイベントソース設定と管理があります。Lambda 関数用のリソースポリシーの導入によって、アクセス許可モデルとプログラミングモデルが簡素化されました。 それに応じてドキュメントが更新されました。詳細については、以下のトピックを参照してください。 Lambda の開始方法 AWS Lambda	2015 年 4 月 9 日
プレビューリリース	『AWS Lambda 開発者ガイド』のプレビューリリース。	2014 年 11 月 13 日