aws

개발자 안내서

AWS Lambda

AWS Lambda: 개발자 안내서

Table of Contents

AWS Lambda이란 무엇인가요?

AWS Lambda를 사용하면 서버를 프로비저닝하거나 관리할 필요 없이 코드를 실행할 수 있습니다.

Lambda는 고가용성 컴퓨팅 인프라에서 코드를 실행하고 서버와 운영 체제 유지 관리, 용량 프로비저닝 및 자동 조정, 코드 및 보안 패치 배포, 로깅 등 모든 컴퓨팅 리소스 관리를 수행합니다. Lambda를 사용하면 Lambda가 지원하는 언어 런타임 중 하나로 코드를 제공하기만 하면 됩니다.

Lambda 함수에 코드를 구성합니다. Lambda 서비스는 필요할 때만 함수를 실행하고 자동으로 확장됩니다. 사용한 컴퓨팅 시간만큼만 비용을 지불하고, 코드가 실행되지 않을 때는 요금이 부과되지 않습니다. 자세한 내용은 AWS Lambda 요금을 참조하세요.

> **Tip**
>
> 서버리스 솔루션을 빌드하는 방법을 알아보려면 서버리스 개발자 안내서를 확인하세요.

Lambda를 사용해야 하는 경우

Lambda는 빠르게 스케일 업해야 하고 수요가 없을 때는 0으로 스케일 다운해야 하는 애플리케이션 시나리오에 이상적인 컴퓨팅 서비스입니다. 예를 들어 Lambda를 다음에 사용할 수 있습니다.

- 파일 처리: 업로드 후 Amazon Simple Storage Service(S3)를 사용하여 Lambda 데이터 처리를 실시간으로 트리거합니다.
- 스트림 처리: Lambda 및 Amazon Kinesis를 사용하여 애플리케이션 작업 추적, 거래 주문 처리, 클릭스트림 분석, 데이터 정리, 로그 필터링, 인덱싱, 소셜 미디어 분석, 사물 인터넷(IoT) 디바이스 데이터 텔레메트리 및 계측을 위한 실시간 스트리밍 데이터를 처리합니다.
- 웹 애플리케이션: Lambda를 다른 AWS 서비스와 결합하여 여러 데이터 센터에서 고가용성 구성으로 자동으로 스케일 업/스케일 다운되고 실행되는 강력한 웹 애플리케이션을 빌드합니다.
- IoT 백엔드: Lambda를 사용하여 서버리스 백엔드를 구축함으로써 웹, 모바일, IoT 및 서드 파티 API 요청을 처리합니다.
- 모바일 백엔드: Lambda 및 Amazon API Gateway를 사용하여 백엔드를 구축함으로써 API 요청을 인증하고 처리합니다. AWS Amplify를 사용하여 iOS, Android, 웹 및 React Native 프론트엔드와 손쉽게 통합합니다.

Lambda를 사용하면 사용자는 자신의 코드에 대해서만 책임을 갖습니다. Lambda는 메모리, CPU, 네트워크 및 기타 리소스의 균형을 제공하는 컴퓨팅 플릿을 관리하여 코드를 실행합니다. Lambda가 이

러한 리소스를 관리하므로 컴퓨팅 인스턴스에 로그인하거나 제공된 런타임에 운영 체제를 사용자 지정할 수 없습니다. Lambda는 사용자를 대신하여 용량 관리, 모니터링 및 Lambda 함수 로깅을 비롯한 운영 및 관리 활동을 수행합니다.

주요 기능

다음 주요 기능은 확장 가능하고 안전하며 쉽게 확장할 수 있는 Lambda 애플리케이션 프로그램을 개발하는 데 도움이 됩니다.

환경 변수

환경 변수를 사용하여 코드를 업데이트하지 않고 함수의 동작을 조정합니다.

버전

예를 들어 안정적인 프로덕션 버전의 사용자에게 영향을 주지 않고 베타 테스트에 새 함수를 사용할 수 있도록 버전으로 함수 배포를 관리합니다.

컨테이너 이미지

기존 컨테이너 도구를 재사용하거나 기계 학습과 같은 상당한 종속 구성 요소에 의존하는 더 큰 워크로드를 배포할 수 있도록 AWS에서 제공하는 기본 이미지 또는 대체 기본 이미지를 사용하여 Lambda 함수에 대한 컨테이너 이미지를 생성합니다.

계층

라이브러리와 기타 종속 구성 요소를 패키징하여 배포 아카이브의 크기를 줄이고 코드를 더 빠르게 배포할 수 있도록 합니다.

Lambda 확장

모니터링, 관측성, 보안 및 거버넌스를 위한 도구로 Lambda 함수를 보강합니다.

함수 URL

Lambda 함수에 전용 HTTP(S) 엔드포인트를 추가합니다.

응답 스트리밍

Node.js 함수에서 클라이언트로 응답 페이로드를 다시 스트리밍하여 첫 번째 바이트까지 시간(TTFB) 성능을 개선하거나 더 큰 페이로드를 반환하도록 Lambda 함수 URL을 구성합니다.

동시성 및 크기 조정 컨트롤

프로덕션 애플리케이션의 크기 조정 및 응답성에 대해 세밀한 제어를 적용합니다.

코드 서명

승인된 개발자만 변경되지 않은 신뢰할 수 있는 코드를 Lambda 함수에 게시하는지 확인합니다.

프라이빗 네트워킹

데이터베이스, 캐시 인스턴스, 내부 서비스 등의 리소스에 대해 프라이빗 네트워크를 생성합니다.

파일 시스템 액세스

함수 코드가 높은 동시성으로 안전하고 공유 리소스에 액세스하고 수정할 수 있게 Amazon Elastic File System(Amazon EFS)을 로컬 디렉터리에 탑재하도록 함수를 구성합니다.

Lambda SnapStart for Java

일반적으로 함수 코드를 변경하지 않고 추가 비용 없이 Java 런타임의 시작 성능을 최대 10배 향상시킵니다.

Lambda 시작하기

Lambda 사용을 시작하려면 Lambda 콘솔에서 함수를 생성해야 합니다. 몇 분 안에 함수를 생성하여 배포하고 콘솔에서 테스트할 수 있습니다.

자습서를 진행하면서 Lambda 이벤트 객체를 사용하여 함수에 인수를 전달하는 방법과 같은 몇 가지 기본적인 Lambda 개념을 배우게 됩니다. 함수에서 로그 출력을 반환하는 방법과 CloudWatch Logs에서 함수의 간접 호출 로그를 보는 방법도 배웁니다.

작업을 단순화하려면 Python 또는 Node.js 런타임을 사용하여 함수를 생성하세요. 이러한 해석된 언어를 사용하면 콘솔의 기본 제공 코드 편집기에서 함수 코드를 직접 편집할 수 있습니다. Java 및 C#과 같은 컴파일된 언어를 사용하면 로컬 빌드 머신에 배포 패키지를 생성하여 Lambda에 업로드해야 합니다. 다른 런타임을 사용하여 Lambda에 함수를 배포하는 방법에 대해 알아보려면 the section called “추가 리소스 및 다음 단계” 섹션의 링크를 참조하세요.

> **Tip**
>
> 서버리스 솔루션을 빌드하는 방법을 알아보려면 서버리스 개발자 안내서를 확인하세요.

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

콘솔로 Lambda 함수 생성

이 예제에서 함수는 `"length"`와 `"width"`라는 레이블이 붙은 2개의 정수 값을 포함하는 JSON 객체를 사용합니다. 함수는 이러한 값을 곱하여 면적을 계산하고 이 값을 JSON 문자열로 반환합니다.

또한 함수는 CloudWatch 로그 그룹의 이름과 함께 계산된 면적을 인쇄합니다. 자습서 후반부에서는 CloudWatch Logs를 사용하여 함수의 간접 호출 레코드를 보는 방법을 배웁니다.

함수를 생성하려면 먼저 콘솔을 사용하여 기본 Hello world 함수를 생성합니다. 다음 단계에서는 자체 함수 코드를 추가합니다.

콘솔로 Hello world 함수 생성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 생성을 선택합니다.
3. 새로 작성을 선택합니다.
4. 기본 정보 창의 함수 이름에 **`myLambdaFunction`**을 입력합니다.
5. 런타임에서 Node.js 20.x 또는 Python 3.12를 선택합니다.
6. 아키텍처를 x86_64로 설정된 상태로 두고 함수 생성을 선택합니다.

Lambda는 `Hello from Lambda!`라는 메시지를 반환하는 함수를 생성합니다. Lambda는 함수에 대한 실행 역할도 생성합니다. 실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 AWS Identity and Access Management(IAM) 역할입니다. 함수에 대해 Lambda가 생성하는 역할은 CloudWatch Logs에 쓸 수 있는 기본 권한을 부여합니다.

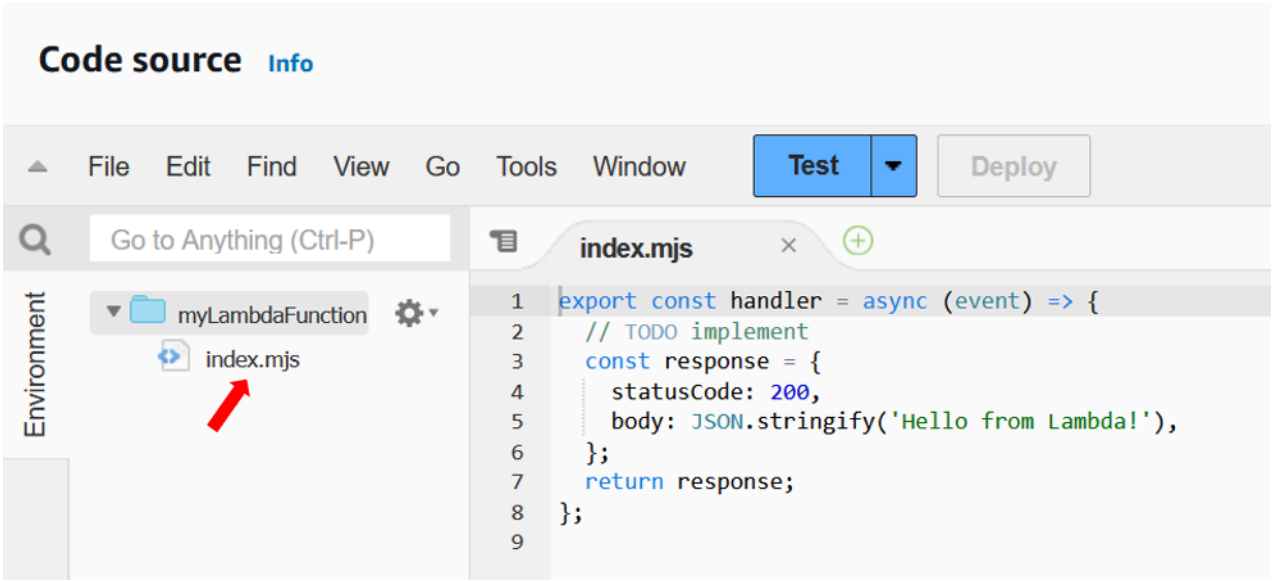
이제 콘솔의 기본 제공 코드 편집기를 사용하여 Lambda가 생성한 Hello world 코드를 사용자의 함수 코드로 바꿉니다.

Node.js

콘솔에서 코드 수정

1. 코드 탭을 선택합니다.

   콘솔의 기본 제공 코드 편집기에서 Lambda가 생성한 함수 코드를 볼 수 있습니다. 코드 편집기에 index.mjs 탭이 표시되지 않으면 다음 다이어그램과 같이 파일 탐색기에서 index.mjs를 선택합니다.



2. 다음 코드를 index.mjs 탭에 붙여 넣고 Lambda가 생성한 코드를 바꿉니다.

```
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. 배포를 선택하여 함수의 코드를 업데이트합니다. Lambda가 변경 사항을 배포하면 콘솔에 함수가 성공적으로 업데이트되었음을 알리는 배너가 표시됩니다.

함수 코드 이해

다음 단계로 넘어가기 전에 함수 코드를 살펴보고 몇 가지 주요 Lambda 개념을 이해하는 시간을 갖도록 하겠습니다.

- Lambda 핸들러:

  Lambda 함수에는 `handler`라는 Node.js 함수가 포함되어 있습니다. Node.js의 Lambda 함수에는 둘 이상의 Node.js 함수가 포함될 수 있지만 handler 함수는 항상 코드의 진입점입니다. 함수가 간접적으로 호출되면 Lambda가 이 메서드를 실행합니다.

  콘솔을 사용하여 Hello world 함수를 생성하면 Lambda는 자동으로 함수에 대한 핸들러 메서드의 이름을 `handler`로 설정합니다. 이 Node.js 함수의 이름을 편집하지 마세요. 그러면 함수를 간접적으로 호출할 때 Lambda에서 코드를 실행할 수 없게 됩니다.

  Node.js의 Lambda 핸들러에 대해 자세히 알아보려면 the section called “핸들러” 섹션을 참조하세요.

- Lambda 이벤트 객체:

  `handler` 함수는 `event`와 `context`라는 두 가지 인수를 사용합니다. Lambda의 이벤트는 함수가 처리할 데이터가 포함된 JSON 형식의 문서입니다.

  다른 AWS 서비스에서 함수를 간접적으로 호출하는 경우 이벤트 객체에는 간접 호출을 일으킨 이벤트에 대한 정보가 포함됩니다. 예를 들어, 객체가 업로드될 때 Amazon Simple Storage Service(S3) 버킷이 함수를 간접적으로 호출하면 이벤트에 Amazon S3 버킷 이름과 객체 키가 포함됩니다.

  이 예제에서는 2개의 키-값 쌍이 포함된 JSON 형식의 문서를 입력하여 콘솔에 이벤트를 생성합니다.

- Lambda 컨텍스트 객체:

  함수가 사용하는 두 번째 인수는 `context`입니다. Lambda는 컨텍스트 객체를 함수에 자동으로 전달합니다. 컨텍스트 객체에는 함수 간접 호출과 실행 환경에 관한 정보가 포함됩니다.

컨텍스트 객체를 사용하여 모니터링 목적으로 함수의 간접 호출에 대한 정보를 출력할 수 있습니다. 이 예제에서 함수는 `logGroupName` 파라미터를 사용하여 CloudWatch 로그 그룹의 이름을 출력합니다.

Node.js의 Lambda 컨텍스트 객체에 대해 자세히 알아보려면 the section called “컨텍스트” 섹션을 참조하세요.

- Lambda에서 로깅:

  Node.js를 사용하면 `console.log` 및 `console.error`와 같은 콘솔 메서드를 사용하여 함수의 로그에 정보를 보낼 수 있습니다. 예제 코드에서는 `console.log` 문을 사용하여 계산된 면적과 함수의 CloudWatch Logs 그룹의 이름을 출력합니다. `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용할 수도 있습니다.

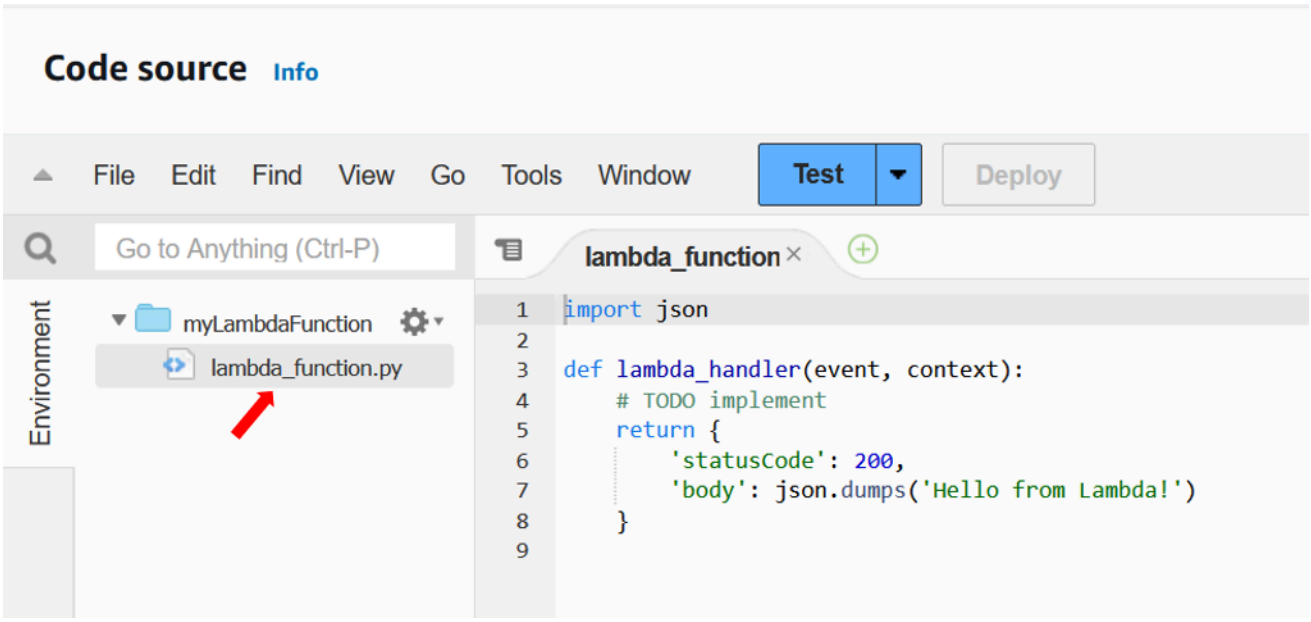
  자세한 내용은 the section called “로깅”을 참조하십시오. 다른 런타임에서 로깅에 대해 알아보려면 관심 있는 런타임에 대한 'Lambda 함수 빌드' 페이지를 참조하세요.

Python

콘솔에서 코드 수정

1. 코드 탭을 선택합니다.

   콘솔의 기본 제공 코드 편집기에서 Lambda가 생성한 함수 코드를 볼 수 있습니다. 코드 편집기에 lambda_function.py 탭이 표시되지 않으면 다음 다이어그램과 같이 파일 탐색기에서 lambda_function.py를 선택합니다.



2. 다음 코드를 lambda_function.py 탭에 붙여 넣고 Lambda가 생성한 코드를 바꿉니다.

```
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. 배포를 선택하여 함수의 코드를 업데이트합니다. Lambda가 변경 사항을 배포하면 콘솔에 함수가 성공적으로 업데이트되었음을 알리는 배너가 표시됩니다.

함수 코드 이해

다음 단계로 넘어가기 전에 함수 코드를 살펴보고 몇 가지 주요 Lambda 개념을 이해하는 시간을 갖도록 하겠습니다.

- Lambda 핸들러:

  Lambda 함수에는 `lambda_handler`라는 Python 함수가 포함되어 있습니다. Python의 Lambda 함수에는 둘 이상의 Python 함수가 포함될 수 있지만 handler 함수는 항상 코드의 진입점입니다. 함수가 간접적으로 호출되면 Lambda가 이 메서드를 실행합니다.

  콘솔을 사용하여 Hello world 함수를 생성하면 Lambda는 자동으로 함수에 대한 핸들러 메서드의 이름을 `lambda_handler`로 설정합니다. 이 Python 함수의 이름을 편집하지 마세요. 그러면 함수를 간접적으로 호출할 때 Lambda에서 코드를 실행할 수 없게 됩니다.

  Python의 Lambda 핸들러에 대해 자세히 알아보려면 the section called “핸들러” 섹션을 참조하세요.

- Lambda 이벤트 객체:

  `lambda_handler` 함수는 `event`와 `context`라는 두 가지 인수를 사용합니다. Lambda의 이벤트는 함수가 처리할 데이터가 포함된 JSON 형식의 문서입니다.

  다른 AWS 서비스에서 함수를 간접적으로 호출하는 경우 이벤트 객체에는 간접 호출을 일으킨 이벤트에 대한 정보가 포함됩니다. 예를 들어, 객체가 업로드될 때 Amazon Simple Storage Service(S3) 버킷이 함수를 간접적으로 호출하면 이벤트에 Amazon S3 버킷 이름과 객체 키가 포함됩니다.

  이 예제에서는 2개의 키-값 쌍이 포함된 JSON 형식의 문서를 입력하여 콘솔에 이벤트를 생성합니다.

- Lambda 컨텍스트 객체:

  함수가 사용하는 두 번째 인수는 `context`입니다. Lambda는 컨텍스트 객체를 함수에 자동으로 전달합니다. 컨텍스트 객체에는 함수 간접 호출과 실행 환경에 관한 정보가 포함됩니다.

컨텍스트 객체를 사용하여 모니터링 목적으로 함수의 간접 호출에 대한 정보를 출력할 수 있습니다. 이 예제에서 함수는 `log_group_name` 파라미터를 사용하여 CloudWatch 로그 그룹의 이름을 출력합니다.

Python의 Lambda 컨텍스트 객체에 대해 자세히 알아보려면 the section called “컨텍스트” 섹션을 참조하세요.

- Lambda에서 로깅:

  Python을 사용하면 `print` 문이나 Python 로깅 라이브러리를 사용하여 정보를 함수의 로그로 보낼 수 있습니다. 캡처된 내용의 차이를 설명하기 위해 예제 코드에서는 두 가지 방법을 모두 사용합니다. 프로덕션 애플리케이션에서는 로깅 라이브러리를 사용하는 것이 좋습니다.

  자세한 내용은 the section called “로깅”을 참조하십시오. 다른 런타임에서 로깅에 대해 알아보려면 관심 있는 런타임에 대한 'Lambda 함수 빌드' 페이지를 참조하세요.

콘솔을 사용하여 간접적으로 Lambda 함수 호출

Lambda 콘솔을 사용하여 함수를 간접적으로 호출하려면 먼저 함수에 보낼 테스트 이벤트를 생성합니다. 이벤트는 `"length"` 및 `"width"` 키가 있는 2개의 키-값 페어를 포함하는 JSON 형식의 문서입니다.

테스트 이벤트 생성

1. 코드 소스 창에서 테스트를 선택합니다.
2. 새 이벤트 생성을 선택합니다.
3. 이벤트 이름에 **myTestEvent**를 입력합니다.
4. 이벤트 JSON 패널에서 다음을 붙여 넣어 기본값을 바꿉니다.

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

5. Save(저장)를 선택합니다.

이제 함수를 테스트하고 Lambda 콘솔과 CloudWatch Logs를 사용하여 함수의 간접 호출 레코드를 봅니다.

함수 테스트 및 콘솔에서 간접 호출 레코드 보기

- 코드 소스 창에서 테스트를 선택합니다. 함수 실행이 완료되면 실행 결과 탭에 응답 및 함수 로그가 표시됩니다. 다음과 유사한 결과가 출력됩니다.

  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
```

이 예제에서는 콘솔의 테스트 기능을 사용하여 코드를 간접적으로 호출했습니다. 즉, 콘솔에서 함수의 실행 결과를 직접 볼 수 있습니다. 함수가 콘솔 외부에서 간접적으로 호출되는 경우 CloudWatch Logs를 사용해야 합니다.

CloudWatch Logs에서 함수의 간접 호출 레코드 보기

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수에 대한 로그 그룹(/aws/lambda/myLambdaFunction)을 선택합니다. 이는 함수가 콘솔에 인쇄한 로그 그룹 이름입니다.
3. 로그 스트림 탭에서 함수의 간접 호출에 대한 로그 스트림을 선택합니다.

   다음과 유사한 출력 화면이 표시되어야 합니다.

   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
```

정리

예제 함수를 사용한 작업을 마치면 해당 함수를 삭제하세요. 함수의 로그를 저장하는 로그 그룹과 콘솔에서 생성된 실행 역할을 삭제할 수도 있습니다.

Lambda 함수 삭제하기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. Delete function(함수 삭제) 대화 상자에 delete를 입력한 후 Delete(삭제)를 선택합니다.

로그 그룹 삭제

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수의 로그 그룹(`/aws/lambda/my-function`)을 선택합니다.
3. 작업(Actions), 로그 그룹 삭제(Delete log group(s))를 선택합니다.
4. 로그 그룹 삭제(Delete log group(s)) 대화 상자에서 삭제(Delete)를 선택합니다.

실행 역할을 삭제하려면

1. AWS Identity and Access Management(IAM) 콘솔의 역할 페이지를 엽니다.
2. 함수의 실행 역할(예: `myLambdaFunction-role-31exxmpl`)을 선택합니다.
3. 삭제를 선택합니다.
4. Delete role(역할 삭제) 대화 상자에 역할 이름을 입력한 후 Delete(삭제)를 선택합니다.

AWS CloudFormation 및 AWS Command Line Interface(AWS CLI)를 사용하여 함수, 로그 그룹 및 역할의 생성 및 정리를 자동화할 수 있습니다.

추가 리소스 및 다음 단계

이제 콘솔을 사용하여 간단한 Lambda 함수를 생성하고 테스트했으니 다음 단계를 수행하세요.

- 코드에 종속 항목을 추가하고.zip 배포 패키지를 사용하여 코드를 배포하는 방법을 알아보세요. 다음 중 관심 있는 언어에 대한 링크를 선택하세요.

  Node.js

  the section called “.zip 파일 아카이브 배포” 섹션을 참조하세요.

  Typescript

  the section called “.zip 파일 아카이브 배포”섹션을 참조하십시오.

  Python

  the section called “.zip 파일 아카이브 배포”섹션을 참조하십시오.

  Ruby

  the section called “.zip 파일 아카이브 배포”섹션을 참조하십시오.

  Java

  the section called “.zip 파일 아카이브 배포”섹션을 참조하십시오.

  Go

  the section called “.zip 파일 아카이브 배포”섹션을 참조하십시오.

  C#

  the section called “배포 패키지” 부분 참조
- 다른 AWS 서비스에서 간접적으로 호출할 Lambda 함수를 구성하는 방법을 알아보려면 Amazon S3 트리거를 사용하여 Lambda 함수 호출 자습서를 수행하세요.
- 다른 AWS 서비스와 함께 Lambda를 사용하는 보다 복잡한 예제를 보려면 다음 자습서 중 하나를 선택하세요.
  - API Gateway에서 Lambda 사용: Lambda 함수를 간접적으로 호출하는 Amazon API Gateway REST API를 생성합니다.

- Amazon RDS에 액세스하기 위해 Lambda 함수 사용: Lambda 함수를 사용하여 RDS 프록시를 통해 Amazon Relational Database Service(RDS) 데이터베이스에 데이터를 씁니다.
- Amazon S3 트리거를 사용하여 썸네일 이미지 생성: 이미지 파일이 Amazon S3 버킷에 업로드될 때마다 Lambda 함수를 사용하여 썸네일을 생성합니다.

AWS Lambda 기본

Lambda 함수는 Lambda 서비스의 주요 리소스입니다.

Lambda 콘솔, Lambda API,AWS CloudFormation 또는 AWS SAM을 사용하여 함수를 구성할 수 있습니다. 함수에 대한 코드를 만들고 배포 패키지를 사용하여 코드를 업로드합니다. 이벤트가 발생하면 Lambda는 함수를 호출합니다. Lambda는 동시성 및 크기 조정 제한에 따라 함수의 여러 인스턴스를 병렬로 실행합니다.

주제

Lambda 개념

Lambda는 함수의 인스턴스를 실행하여 이벤트를 처리합니다. Lambda API를 사용하여 함수를 직접 호출하거나, 함수를 호출하도록 AWS 서비스 또는 리소스를 구성할 수 있습니다.

개념

- 함수
- 트리거
- Event
- 실행 환경
- 명령 세트 아키텍처
- 배포 패키지
- 런타임
- 계층
- 확장
- 동시성
- Qualifier
- 대상

함수

함수는 Lambda에서 코드를 실행하기 위해 호출할 수 있는 리소스입니다. 함수에는 사용자가 함수에 전달하거나 다른 AWS 서비스가 함수에 보내는 이벤트를 처리하는 코드가 있습니다.

트리거

트리거는 Lambda 함수를 호출하는 리소스 또는 구성입니다. 트리거에는 함수 및 이벤트 소스 매핑을 호출하도록 구성할 수 있는 AWS 서비스가 포함됩니다. 이벤트 소스 매핑은 스트림 또는 대기열에서 항목을 읽고 함수를 호출하는 Lambda의 리소스입니다. 자세한 내용은 Lambda 함수 간접 호출 방법 이해 및 다른 AWS 서비스의 이벤트로 Lambda 간접 호출 섹션을 참조하세요.

Event

이벤트는 처리할 Lambda 함수에 대한 데이터가 포함된 JSON 형식 문서입니다. 런타임은 이벤트를 객체로 변환한 후 함수 코드에 전달합니다. 함수를 호출할 때, 이벤트의 구조와 내용을 결정합니다.

Example 사용자 지정 이벤트 - 날씨 데이터

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

AWS 서비스가 함수를 호출할 때 서비스가 이벤트의 형태를 정의합니다.

Example 서비스 이벤트 - Amazon SNS 알림

```
{
  "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 서비스의 이벤트에 대한 자세한 내용은 다른 AWS 서비스의 이벤트로 Lambda 간접 호출 단원을 참조하세요.

실행 환경

실행 환경은 Lambda 함수를 위한 안전하고 격리된 런타임 환경을 제공합니다. 실행 환경은 함수를 실행하는 데 필요한 프로세스와 리소스를 관리합니다. 실행 환경은 함수 및 함수와 관련된 모든 익스텐션에 대한 수명 주기 지원을 제공합니다.

자세한 설명은 Lambda 실행 환경 섹션을 참조하세요.

명령 세트 아키텍처

명령 세트 아키텍처에 따라 Lambda가 함수를 실행하는 데 사용하는 컴퓨터 프로세서의 유형이 결정됩니다. Lambda는 다음과 같은 명령 세트 아키텍처를 선택할 수 있는 옵션을 제공합니다.

- arm64 - AWS Graviton2 프로세서에 사용되는 64비트 ARM 아키텍처입니다.
- x86_64 - x86 기반 프로세서에 사용되는 64비트 x86 아키텍처입니다.

자세한 설명은 Lambda 함수에 대한 명령 세트 아키텍처 구성 섹션을 참조하세요.

배포 패키지

배포 패키지를 사용하여 Lambda 함수 코드를 배포합니다. Lambda는 다음과 같은 두 가지 유형의 배포 패키지를 지원합니다.

- 함수 코드 및 종속 항목이 포함된 .zip 파일 아카이브. Lambda는 함수를 위한 운영 체제 및 런타임을 제공합니다.
- Open Container Initiative(OCI) 사양과 호환되는 컨테이너 이미지. 함수 코드와 종속 항목을 이미지에 추가합니다. 운영 체제와 Lambda 런타임도 포함해야 합니다.

자세한 설명은 Lambda 배포 패키지 섹션을 참조하세요.

런타임

런타임은 실행 환경에서 실행되는 언어별 환경을 제공합니다. 런타임은 Lambda와 함수 간의 호출 이벤트, 컨텍스트 정보 및 응답을 전달합니다. Lambda에서 제공하는 런타임을 사용하거나 나만의 런타임을 빌드할 수 있습니다. 코드를 .zip 파일 아카이브로 패키징하는 경우 프로그래밍 언어와 일치하는 런타임을 사용하도록 함수를 구성해야 합니다. 컨테이너 이미지의 경우 이미지를 빌드할 때 런타임을 포함합니다.

자세한 설명은 Lambda 런타임 섹션을 참조하세요.

계층

Lambda 계층은 추가 코드 또는 기타 콘텐츠를 포함할 수 있는 .zip 파일 아카이브입니다. 계층에는 라이브러리, 사용자 정의 런타임, 데이터 또는 구성 파일이 포함될 수 있습니다.

계층은 Lambda 함수와 함께 사용할 수 있는 라이브러리 및 기타 종속성을 패키징하는 편리한 방법을 제공합니다. 계층을 사용하면 업로드된 배포 아카이브의 크기가 줄어들고 코드를 더 빠르게 배포할 수 있습니다. 계층은 또한 코드 공유 및 책임 분리를 촉진하므로 비즈니스 로직 작성시 더 빠르게 반복할 수 있습니다.

함수당 최대 5개의 계층을 포함할 수 있습니다. 계층은 표준 Lambda 배포 크기 할당량에 포함됩니다. 함수에 계층을 포함하면 실행 환경의 `/opt` 디렉터리로 콘텐츠가 추출됩니다.

기본적으로 생성하는 계층은 AWS 계정에 비공개입니다. 계층을 다른 계정과 공유하거나 계층을 공개하도록 선택할 수 있습니다. 함수가 다른 계정에서 발행한 계층을 사용하는 경우 함수가 계층 버전을

삭제한 후 또는 계층에 대한 액세스 권한이 취소된 후에도 해당 계층 버전을 계속 사용할 수 있습니다. 그러나 삭제된 계층 버전을 사용하여 새 함수를 작성하거나 함수를 업데이트할 수는 없습니다.

컨테이너 이미지로 배포된 함수는 계층을 사용하지 않습니다. 대신 이미지를 빌드할 때 기본 설정 런타임, 라이브러리 및 기타 종속 항목을 컨테이너 이미지로 패키징합니다.

자세한 설명은 Lambda 계층 섹션을 참조하세요.

확장

Lambda 익스텐션을 사용하면 함수를 보강할 수 있습니다. 예를 들어 익스텐션을 사용하여 원하는 모니터링, 관찰, 보안 및 거버넌스 도구와 함수를 통합할 수 있습니다. AWS Lambda 파트너가 제공하는 다양한 도구 중에서 선택하거나 자체적인 Lambda 익스텐션을 만들 수 있습니다.

내부 익스텐션은 런타임 프로세스에서 실행되며 런타임과 동일한 수명 주기를 공유합니다. 외부 익스텐션은 실행 환경에서 별도의 프로세스로 실행됩니다. 외부 익스텐션은 함수가 호출되기 전에 초기화되고 함수의 런타임과 동시에 실행되며 함수 호출이 완료된 후에도 계속 실행됩니다.

자세한 설명은 Lambda 확장을 사용하여 Lambda 함수 보강 섹션을 참조하세요.

동시성

동시성은 특정 시각에 함수가 제공하는 요청의 수입니다. 함수가 호출되면 이벤트를 처리하도록 Lambda가 인스턴스를 프로비저닝합니다. 함수 코드가 실행을 마치면, 다른 요청을 처리할 수 있습니다. 요청이 처리되는 동안 함수가 다시 호출되면, 다른 인스턴스가 프로비저닝되어 함수의 동시성이 증가합니다.

동시성에는 AWS 리전 수준의 할당량이 적용됩니다. 동시성을 제한하거나 특정 수준의 동시성에 도달할 수 있도록 개별 함수를 구성할 수 있습니다. 자세한 설명은 함수에 대해 예약된 동시성 구성 섹션을 참조하세요.

Qualifier

함수를 호출하거나 조회할 때 버전 또는 별칭을 지정하는 한정자를 포함할 수 있습니다. 버전은 숫자 한정자가 있는 함수 코드 및 구성의 변경 불가능한 스냅샷입니다. 예: `my-function:1`. 별칭은 다른 버전에 매핑하거나 두 버전 간에 트래픽을 분할하도록 업데이트할 수 있는 버전에 대한 포인터입니다. 예: `my-function:BLUE`. 버전 및 별칭을 함께 사용하여 클라이언트가 함수를 호출할 수 있는 안정적인 인터페이스를 제공할 수 있습니다.

자세한 설명은 Lambda 함수 버전 섹션을 참조하세요.

대상

대상은 Lambda가 동기식 호출의 이벤트를 전송할 수 있는 AWS 리소스입니다. 사용자는 처리에 실패한 이벤트의 대상을 구성할 수 있습니다. 일부 서비스는 성공적으로 처리된 이벤트의 대상도 지원합니다.

자세한 내용은 비동기식 호출에 대한 대상 구성 섹션을 참조하세요.

Lambda 프로그래밍 모델

Lambda는 모든 런타임에 공통적인 프로그래밍 모델을 제공합니다. 프로그래밍 모델은 코드와 Lambda 시스템 간의 인터페이스를 정의합니다. 함수 구성에서 핸들러를 정의하여 Lambda에게 함수의 진입점을 알려줍니다. 런타임은 호출 이벤트와 컨텍스트(예: 함수 이름, 요청 ID)를 포함하는 핸들러로 객체를 전달합니다.

핸들러가 첫 번째 이벤트 처리를 완료하면 런타임이 다른 이벤트를 보냅니다. 함수의 클래스가 메모리에 유지되므로, 초기화 코드에서 핸들러 메서드 외부에서 선언된 클라이언트 및 변수를 재사용할 수 있습니다. 후속 이벤트에 대한 처리 시간을 절약하려면 초기화 중에 AWS SDK 클라이언트와 같은 재사용 가능한 리소스를 생성합니다. 초기화된 후에는 함수의 각 인스턴스가 수천 개의 요청을 처리할 수 있습니다.

함수는 `/tmp` 디렉터리의 로컬 스토리지에도 액세스할 수 있습니다. 디렉터리 콘텐츠는 실행 환경이 일시 중지되어도 그대로 유지되기 때문에 일시적인 캐시를 여러 호출에서 사용할 수 있습니다. 자세한 내용은 Lambda 실행 환경을 참조하세요.

AWS X-Ray 추적이 활성화된 경우 런타임이 초기화와 실행에 대해 별도의 하위 세그먼트를 기록합니다.

런타임은 함수의 로깅 출력을 캡처하여 Amazon CloudWatch Logs로 보냅니다. 런타임은 함수의 출력을 로깅하는 것 외에도 호출이 시작되고 끝날 때 항목을 로깅합니다. 여기에는 요청 ID, 요금이 청구되는 소요 시간, 초기화 소요 시간 및 기타 세부 정보가 들어 있는 보고서 로그가 포함됩니다. 함수에서 오류가 발생하는 경우, 런타임은 해당 오류를 호출자에게 반환합니다.

> **Note**
>
> 로깅에는 CloudWatch 로그 할당량이 적용됩니다. 로그 데이터는 조절로 인해, 또는 경우에 따라 함수의 인스턴스가 중지될 때 손실될 수 있습니다.

Lambda는 수요가 증가하면 추가 인스턴스를 실행하고 수요가 감소하면 인스턴스를 중지하여 함수 규모를 조정합니다. 이 모델은 다음과 같은 애플리케이션 아키텍처의 변형을 초래합니다.

- 달리 지정되지 않는 한, 수신 요청은 비순차적으로, 또는 동시에 처리될 수 있습니다.
- 오래 지속되는 함수의 인스턴스를 사용하지 말고 대신 애플리케이션의 상태를 다른 곳에 저장하세요.
- 로컬 스토리지 및 클래스 수준 객체를 사용하여 성능을 강화할 수 있지만, 실행 환경으로 전송하는 배포 패키지의 크기와 데이터의 양을 최소로 유지하세요.

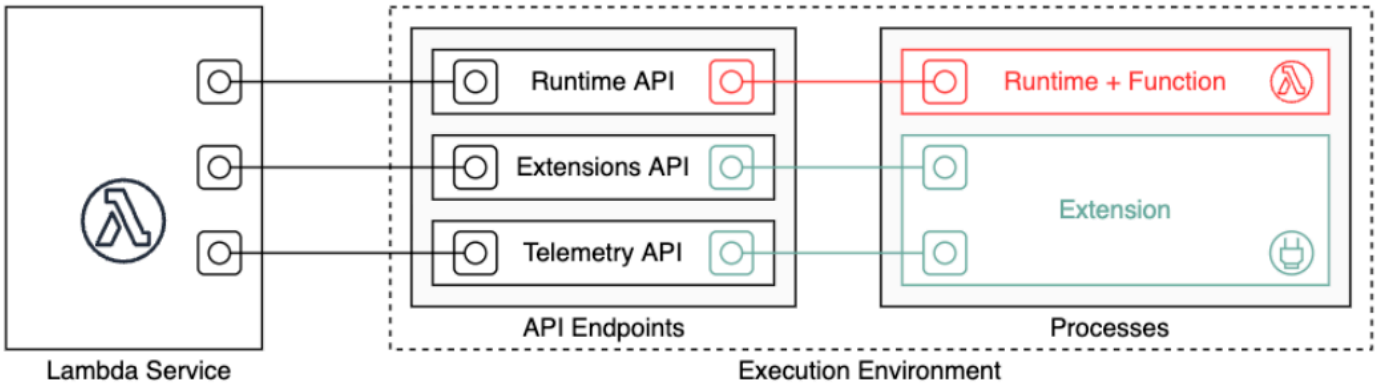
다음 장에서는 선호하는 프로그래밍 언어의 프로그래밍 모델을 실습 과정을 겸하여 소개합니다.

- Node.js를 사용하여 Lambda 함수 빌드
- Python을 사용하여 Lambda 함수 빌드
- Ruby를 사용하여 Lambda 함수 빌드
- Java를 사용하여 Lambda 함수 빌드
- Go를 사용하여 Lambda 함수 빌드
- C#을 사용하여 Lambda 함수 빌드
- PowerShell을 사용하여 Lambda 함수 빌드

Lambda 실행 환경

Lambda는 안전하고 격리된 런타임 환경을 제공하는 실행 환경에서 함수를 호출합니다. 실행 환경은 함수를 실행하는 데 필요한 리소스를 관리합니다. 또한 실행 환경은 함수의 런타임 및 함수와 관련된 모든 외부 익스텐션에 대한 수명 주기 지원을 제공합니다.

함수의 런타임은 런타임 API를 사용하여 Lambda와 통신합니다. 익스텐션은 익스텐션 API를 사용하여 Lambda와 통신합니다. 또한 확장은 텔레메트리 API를 사용하여 함수의 로그 메시지와 기타 텔레메트리 데이터를 수신할 수 있습니다.



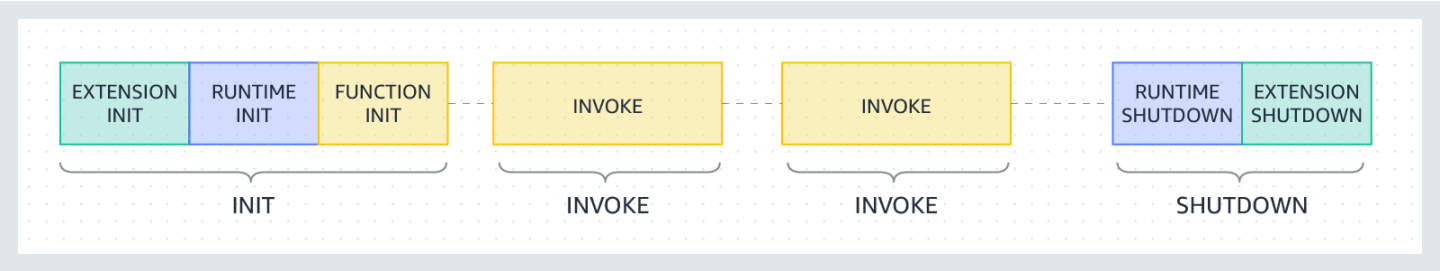
Lambda 함수를 생성할 때 함수에서 허용된 메모리 용량 및 최대 실행 시간 같은 구성 정보를 지정합니다. Lambda는 이 정보를 사용하여 실행 환경을 설정합니다.

함수의 런타임 및 각 외부 익스텐션은 실행 환경 내에서 실행되는 프로세스입니다. 권한, 리소스, 자격 증명 및 환경 변수는 함수와 익스텐션 간에 공유됩니다.

주제

- Lambda 실행 환경 수명 주기
- 함수에서 상태 비저장 구현

Lambda 실행 환경 수명 주기



각 단계는 Lambda가 런타임 및 등록된 모든 익스텐션에 전송하는 이벤트로 시작됩니다. 런타임 및 각 익스텐션이 `Next` API 요청을 전송하여 완료를 나타냅니다. 런타임 및 각 익스텐션이 완료되고 대기 중인 이벤트가 없으면 Lambda는 실행 환경을 중지합니다.

주제

초기화 단계

`Init` 단계에서 Lambda는 다음 세 가지 작업을 수행합니다.

- 모든 익스텐션 시작(`Extension init`)
- 런타임 부트스트랩(`Runtime init`)
- 함수의 정적 코드 실행(`Function init`)
- 모든 `beforeCheckpoint` 런타임 후크 실행(Lambda SnapStart만 해당)

`Init` 단계는 런타임 및 모든 익스텐션이 `Next` API 요청을 전송하여 준비되었음을 알린 후에 종료됩니다. `Init` 단계는 10초로 제한됩니다. 세 작업이 모두 10초 이내에 완료되지 않으면 Lambda는 첫 번째 함수 호출 시 구성된 함수 타임아웃으로 `Init` 단계를 다시 시도합니다.

Lambda SnapStart가 활성화되면 함수 버전을 게시할 때 `Init` 단계가 발생합니다. Lambda는 초기화된 실행 환경의 메모리 및 디스크 상태 스냅샷을 저장하고 암호화된 스냅샷을 유지하며 짧은 지연 시간으로 액세스할 수 있도록 스냅샷을 캐싱합니다. `beforeCheckpoint` 런타임 후크가 있는 경우 코드는 `Init` 단계가 끝날 때 실행됩니다.

> **Note**
>
> 10초 시간 제한은 프로비저닝된 동시성 또는 SnapStart를 사용하는 함수에는 적용되지 않습니다. 프로비저닝된 동시성 및 SnapStart 함수인 경우 초기화 코드를 최대 15분 동안 실행할 수 있습니다. 시간 제한은 130초 또는 구성된 함수 제한 시간(최대 900초) 중 더 높은 값입니다.

프로비저닝된 동시성을 사용하면 함수에 대한 PC 설정을 구성할 때 Lambda가 실행 환경을 초기화합니다. 또한 Lambda는 간접 호출에 앞서 초기화된 실행 환경을 항상 사용할 수 있도록 합니다. 함수의 간접 호출 단계와 초기화 단계 사이에 시간 격차가 발생할 수 있습니다. 함수의 런타임 및 메모리 구성에 따라 초기화된 실행 환경에서 첫 번째 간접 호출 시 가변적 지연 시간이 나타날 수도 있습니다.

온디맨드 동시성을 사용하는 함수의 경우 Lambda는 간접 호출 요청에 앞서 실행 환경을 초기화하는 경우가 있습니다. 이 경우 함수의 초기화 단계와 간접 호출 단계 사이에 시간 격차가 발생할 수 있습니다. 이 동작에 의존하지 않는 것이 좋습니다.

Init 단계 중 실패

`Init` 단계 중에 함수가 충돌하거나 시간이 초과되면 Lambda는 `INIT_REPORT` 로그에 오류 정보를 생성합니다.

Example — 타임아웃에 대한 INIT_REPORT 로그

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

Example — 확장 실패에 대한 INIT_REPORT 로그

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

`Init` 단계가 성공하면 SnapStart가 활성화되지 않는 한 Lambda는 `INIT_REPORT` 로그를 내보내지 않습니다. SnapStart 함수는 항상 `INIT_REPORT`를 내보냅니다. 자세한 내용은 Lambda에 대한 모니터링 SnapStart 단원을 참조하십시오.

복원 단계(Lambda SnapStart만 해당)

SnapStart 함수를 처음 호출하고 함수가 스케일 업되면 Lambda는 이 함수를 처음부터 초기화하는 대신 유지된 스냅샷에서 새 실행 환경을 재개합니다. `afterRestore()` 런타임 후크가 있는 경우 코드는 `Restore` 단계가 끝날 때 실행됩니다. `afterRestore()` 런타임 후크 지속 시간에 대해 요금이 청구됩니다. 런타임(JVM)이 로드되고 `afterRestore()` 런타임 후크가 제한 시간(10초) 내에 완료되어야 합니다. 그렇지 않으면 SnapStartTimeoutException이 발생합니다. `Restore` 단계가 완료되면 Lambda가 함수 핸들러(호출 단계)를 간접적으로 호출합니다.

복원 단계 중 실패

`Restore` 단계가 실패하면 Lambda는 `RESTORE_REPORT` 로그에 오류 정보를 생성합니다.

Example — 타임아웃에 대한 RESTORE_REPORT 로그

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

Example — 런타임 후크 실패에 대한 RESTORE_REPORT 로그

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

RESTORE_REPORT log에 대한 자세한 내용은 Lambda에 대한 모니터링 SnapStart을 참조하십시오.

호출 단계

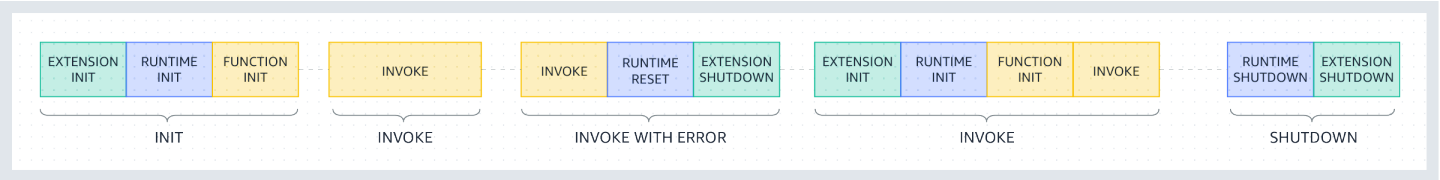
Next API 요청에 대한 응답으로 Lambda 함수가 호출되면 Lambda는 런타임과 각 익스텐션에 Invoke 이벤트를 전송합니다.

함수의 제한 시간 설정은 전체 Invoke 단계의 기간을 제한합니다. 예를 들어 함수 제한 시간을 360초로 설정하면 함수와 모든 익스텐션이 360초 내에 완료되어야 합니다. 독립적인 호출 후 단계는 없습니다. 기간은 모든 호출 시간(런타임 + 익스텐션)의 합계이며 함수와 모든 익스텐션의 실행이 완료될 때까지 계산되지 않습니다.

호출 단계는 런타임 및 모든 익스텐션이 Next API 요청을 전송하여 완료되었음을 알린 후에 종료됩니다.

호출 단계 중 실패

Invoke 단계 중에 Lambda 함수가 충돌하거나 시간이 초과되면 Lambda는 실행 환경을 재설정합니다. 다음 다이어그램은 호출 실패 시 Lambda 실행 환경 동작을 보여줍니다.



이전 다이어그램에서

- 첫 번째 단계는 오류 없이 실행되는 INIT 단계입니다.
- 두 번째 단계는 오류 없이 실행되는 INVOKE 단계입니다.
- 어떤 시점에서 함수에 호출 오류(함수 시간 초과나 런타임 오류)가 발생했다고 하겠습니다. INVOKE WITH ERROR라고 표시된 세 번째 단계가 이 시나리오를 보여줍니다. 이 경우 Lambda 서비스가 재

설정을 수행합니다. 재설정은 Shutdown 이벤트처럼 동작합니다. 먼저 Lambda는 런타임을 종료한 다음 등록된 각 외부 익스텐션에 Shutdown 이벤트를 전송합니다. 이벤트에는 종료 이유가 포함됩니다. 이 환경이 새 호출에 사용되는 경우 Lambda는 익스텐션과 런타임을 다음 호출과 함께 다시 초기화합니다.

> **Note**
>
> Lambda 재설정으로 다음 초기화 단계 전에 /tmp 디렉터리의 내용이 지워지지 않습니다. 이 동작은 일반 종료 단계와 동일합니다.

- 네 번째 단계는 간접 호출 실패 직후의 INVOKE 단계를 나타냅니다. 여기서 Lambda는 INIT 단계를 다시 실행하여 환경을 다시 초기화합니다. 이를 억제된 초기화라고 합니다. 억제된 초기화가 발생하는 경우 Lambda는 CloudWatch 로그에 추가 INIT 단계를 명시적으로 보고하지는 않습니다. 대신 REPORT 줄의 지속 시간에 추가 INIT 기간 + INVOKE 기간이 포함되어 있음을 알 수 있습니다. 예를 들어 CloudWatch에 다음과 같은 로그가 표시되어 있다고 가정하겠습니다.

```
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
```

  이 예제에서 REPORT 타임스탬프와 START 타임스탬프 간의 차이는 2.5초입니다. 이는 Lambda가 수행한 추가 INIT(억제된 초기화)를 고려하지 않기 때문에 보고된 지속 시간인 3022.91 밀리초와 일치하지 않습니다. 이 예시에서는 실제 INVOKE 단계에 2.5초가 걸렸다고 추론할 수 있습니다.

  이 동작에 대해 더 자세히 알아보려면 Lambda 텔레메트리 API을 사용하면 됩니다. Telemetry API는 호출 단계 사이에 억제된 초기화가 발생할 때마다 INIT_START, INIT_RUNTIME_DONE, INIT_REPORT, phase=invoke 이벤트를 내보냅니다.
- 다섯 번째 단계는 오류 없이 실행되는 SHUTDOWN 단계를 나타냅니다.

종료 단계

Lambda는 런타임을 종료하려고 할 때 Shutdown 이벤트를 등록된 각 외부 익스텐션에 전송합니다. 익스텐션은 이 시간 동안 최종 정리 작업을 수행할 수 있습니다. Shutdown 이벤트는 Next API 요청에 대한 응답입니다.

기간: 전체 Shutdown 단계는 2초로 제한됩니다. 런타임 또는 익스텐션이 응답하지 않는 경우, Lambda는 신호(SIGKILL)를 통해 이를 종료합니다.

함수와 모든 익스텐션이 완료된 후 Lambda는 다른 함수 호출을 예상하여 일정 시간 동안 실행 환경을 유지합니다. 그러나 Lambda는 몇 시간마다 실행 환경을 종료하여 런타임 업데이트 및 유지 관리를 허용하며, 이는 지속적으로 호출되는 함수에 대해서도 동일합니다. 실행 환경이 무기한 지속될 것이라고 가정하면 안 됩니다. 자세한 내용은 함수에서 상태 비저장 구현 단원을 참조하십시오.

함수가 다시 호출되면 Lambda는 재사용을 위해 환경을 재개합니다. 실행 환경을 재사용하는 것은 다음과 같은 의미를 가집니다.

- 함수의 핸들러 메서드 외부에서 선언된 객체는 함수가 다시 호출될 때 추가로 최적화가 되도록 초기화된 상태로 유지됩니다. 예를 들어 Lambda 함수가 연결을 재설정하는 대신 데이터베이스 연결을 설정하면 원래 연결이 후속 호출에 사용됩니다. 코드에 로직을 추가하여 연결을 새로 생성하기에 앞서 연결이 이미 존재하는지 확인하는 것이 좋습니다.
- 각 실행 환경은 `/tmp` 디렉터리에 512MB에서 10,240MB 사이 1MB 단위로 디스크 공간을 제공합니다. 디렉터리 콘텐츠는 실행 환경이 일시 중지되어도 그대로 유지되기 때문에 일시적인 캐시를 여러 호출에서 사용할 수 있습니다. 코드를 추가하여 캐시에 저장한 데이터가 포함되어 있는지 확인할 수 있습니다. 배포 크기 제한에 대한 자세한 내용은 Lambda 할당량 섹션을 참조하세요.
- Lambda 함수에 의해 초기화되었고 함수 종료 시 완료되지 않은 백그라운드 프로세스 또는 콜백은 Lambda가 실행 환경을 재사용하는 경우에 재개됩니다. 코드가 종료되려면 먼저 코드의 백그라운드 프로세스 또는 콜백이 완료되어야 합니다.

함수에서 상태 비저장 구현

Lambda 함수 코드를 작성할 때는 실행 환경이 단일 호출에만 존재한다고 가정하여 상태 비저장 환경으로 취급합니다. Lambda는 몇 시간마다 실행 환경을 종료하여 런타임 업데이트 및 유지 관리를 허용하며, 이는 지속적으로 호출되는 함수에 대해서도 동일합니다. 함수 시작 시 필수 상태(예: Amazon DynamoDB 테이블에서 장바구니 가져오기)를 초기화합니다. 종료하기 전에 Amazon Simple Storage Service(S3), DynamoDB 또는 Amazon Simple Queue Service(Amazon SQS) 같은 내구성이 뛰어난 스토어에 영구적인 데이터 변경 사항을 커밋합니다. 기존 데이터 구조, 임시 파일 또는 카운터 또는 애그리게이트 등의 호출에 걸친 상태에 의존하기를 피합니다. 이렇게 하면 함수가 각 호출을 독립적으로 처리할 수 있습니다.

Lambda 배포 패키지

AWS Lambda 함수의 코드는 스크립트 또는 컴파일된 프로그램과 해당 종속 항목으로 구성됩니다. 함수 코드는 배포 패키지를 사용하여 Lambda에 배포합니다. Lambda는 컨테이너 이미지 및 .zip 파일 아카이브의 두 가지 배포 패키지를 지원합니다.

주제

- 컨테이너 이미지
- .zip 파일 아카이브
- 계층
- 다른 AWS 서비스를 사용하여 배포 패키지 빌드

컨테이너 이미지

컨테이너 이미지에는 기본 운영 체제, 런타임, Lambda 익스텐션, 애플리케이션 코드 및 해당 종속 항목이 포함됩니다. 기계 학습 모델과 같은 정적 데이터를 이미지에 추가할 수도 있습니다.

Lambda는 컨테이너 이미지를 빌드하는 데 사용할 수 있는 일련의 오픈 소스 기본 이미지를 제공합니다. 컨테이너 이미지를 생성하고 테스트하려면 AWS Serverless Application Model(AWS SAM) 명령줄 인터페이스(CLI) 또는 Docker CLI와 같은 기본 컨테이너 도구를 사용할 수 있습니다.

컨테이너 이미지를 관리형 AWS 컨테이너 이미지 레지스트리 서비스인 Amazon Elastic Container Registry(Amazon ECR)에 업로드하세요. 함수에 이미지를 배포하려면 Lambda 콘솔, Lambda API, 명령줄 도구 또는 AWS SDK를 사용하여 Amazon ECR 이미지 URL을 지정하세요.

Lambda 컨테이너 이미지에 대한 자세한 내용은 컨테이너 이미지를 사용하여 Lambda 함수 생성 단원을 참조하세요.

.zip 파일 아카이브

.zip 파일 아카이브에는 애플리케이션 코드와 해당 종속 항목이 포함됩니다. Lambda 콘솔이나 도구 키트를 사용하여 함수를 작성하면 Lambda가 코드의.zip 파일 아카이브를 자동으로 생성합니다.

Lambda API, 명령줄 도구 또는 AWS SDK를 사용하여 함수를 생성하는 경우 배포 패키지를 생성해야 합니다. 또한 함수가 컴파일된 언어를 사용하거나 함수에 종속 항목을 추가하는 경우에도 배포 패키지를 생성해야 합니다. 함수 코드를 배포하려면 Amazon Simple Storage Service(Amazon S3) 또는 로컬 컴퓨터에서 배포 패키지를 업로드하세요.

Lambda 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 Amazon Simple Storage Service(Amazon S3) 버킷에 .zip 파일을 배포 패키지로 업로드할 수 있습니다.

Lambda 콘솔 사용

다음 단계에서는 Lambda 콘솔을 사용하여 .zip 파일을 배포 패키지로 업로드하는 방법을 보여줍니다.

Lambda 콘솔에서 .zip 파일을 업로드하는 방법

1. Lambda 콘솔에서 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 코드 소스 창에서 업로드 원본(Upload from)을 선택한 다음 .zip 파일을 선택합니다.
4. [업로드(Upload)]를 선택하여 로컬 .zip 파일을 선택합니다.
5. 저장을 선택합니다.

AWS CLI 사용

AWS Command Line Interface(AWS CLI)을(를) 사용하여 .zip 파일을 배포 패키지로 업로드할 수 있습니다. 언어별 지침은 다음 주제를 참조하세요.

Node.js

- .zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포

Python

- Python Lambda 함수에 대한 .zip 파일 아카이브 작업

Ruby

- Ruby Lambda 함수에 대한 .zip 파일 아카이브 작업

Java

- .zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포

Go

- .zip 파일 아카이브를 사용하여 Go Lambda 함수 배포

C#

- .zip 파일 아카이브를 사용하여 C# Lambda 함수를 빌드 및 배포

PowerShell

- .zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포

Amazon S3 사용

Amazon Simple Storage Service(Amazon S3)을 사용하여 .zip 파일을 배포 패키지로 업로드할 수 있습니다. 자세한 내용은 단원을 참조하십시오.

계층

.zip 파일 아카이브를 사용하여 함수 코드를 배포하는 경우 라이브러리, 사용자 지정 런타임 및 기타 함수 종속 항목에 대한 배포 메커니즘으로 Lambda 계층을 사용할 수 있습니다. 계층을 사용하면 개발 중인 함수 코드를 변경되지 않는 코드 및 리소스와는 별도로 관리할 수 있습니다. 사용자가 생성한 계층, AWS에서 제공하는 계층 또는 다른 AWS 고객들의 계층을 사용하도록 함수를 구성할 수 있습니다.

컨테이너 이미지에는 계층을 사용할 수 없습니다. 대신 이미지를 빌드할 때 기본 설정 런타임, 라이브러리, 기타 종속 항목을 컨테이너 이미지로 패키징하세요.

계층에 대한 자세한 내용은 Lambda 계층 단원을 참조하세요.

다른 AWS 서비스를 사용하여 배포 패키지 빌드

다음 섹션에서는 Lambda 함수에 대한 종속성을 패키지하는 데 사용할 수 있는 다른 AWS 서비스를 설명합니다.

C 또는 C++ 라이브러리가 포함된 배포 패키지

배포 패키지에 네이티브 라이브러리가 포함된 경우 AWS Serverless Application Model(AWS SAM)를 사용하여 배포 패키지를 빌드할 수 있습니다. AWS SAM CLI `sam build` 명령을 `--use-container`와 함께 사용하여 배포 패키지를 만들 수 있습니다. 이 옵션은 Lambda 실행 환경과 호환되는 도커 이미지 내부에 배포 패키지를 빌드합니다.

자세한 내용은 AWS Serverless Application Model 개발자 안내서의 sam 빌드를 참조하세요.

50MB를 초과하는 배포 패키지

배포 패키지가 50MB보다 큰 경우 Amazon S3 버킷에 함수 코드와 종속 항목을 업로드하십시오.

배포 패키지를 생성하고, Lambda 함수를 생성하려는 AWS 리전의 Amazon S3 버킷에 .zip 파일을 업로드할 수 있습니다. Lambda 함수를 생성할 때 Lambda 콘솔에서 S3 버킷 이름과 객체 키 이름을 지정하거나 AWS CLI를 사용합니다.

Amazon S3 콘솔을 사용하여 버킷을 생성하려면 Amazon Simple Storage Service 사용 설명서의 버킷 생성을 참조하세요.

코드형 인프라(IaC)와 함께 Lambda 사용

Lambda는 코드를 배포하고 함수를 생성하는 여러 가지 방법을 제공합니다. 예를 들어, Lambda 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 Lambda 함수를 수동으로 생성하거나 업데이트할 수 있습니다. 이러한 수동 옵션 외에도 AWS는 코드형 인프라(IaC)를 사용하여 Lambda 함수 및 서버리스 애플리케이션을 배포하기 위한 다양한 솔루션을 제공합니다. IaC로 수동 프로세스와 설정을 사용하는 대신 코드를 사용하여 Lambda 함수 및 기타 AWS 리소스를 프로비저닝하고 유지할 수 있습니다.

대부분의 경우 Lambda 함수는 격리되어 실행되지 않습니다. 대신 데이터베이스, 대기열, 스토리지와 같은 다른 리소스와 함께 서버리스 애플리케이션의 일부를 구성합니다. IaC로 배포 프로세스를 자동화하여 많은 개별 AWS 리소스를 포함하는 전체 서버리스 애플리케이션을 빠르고 반복적으로 배포하고 업데이트할 수 있습니다. 이 접근법을 사용하면 개발 주기가 단축되고 구성 관리가 쉬워지며 리소스가 매번 같은 방식으로 배포될 수 있습니다.

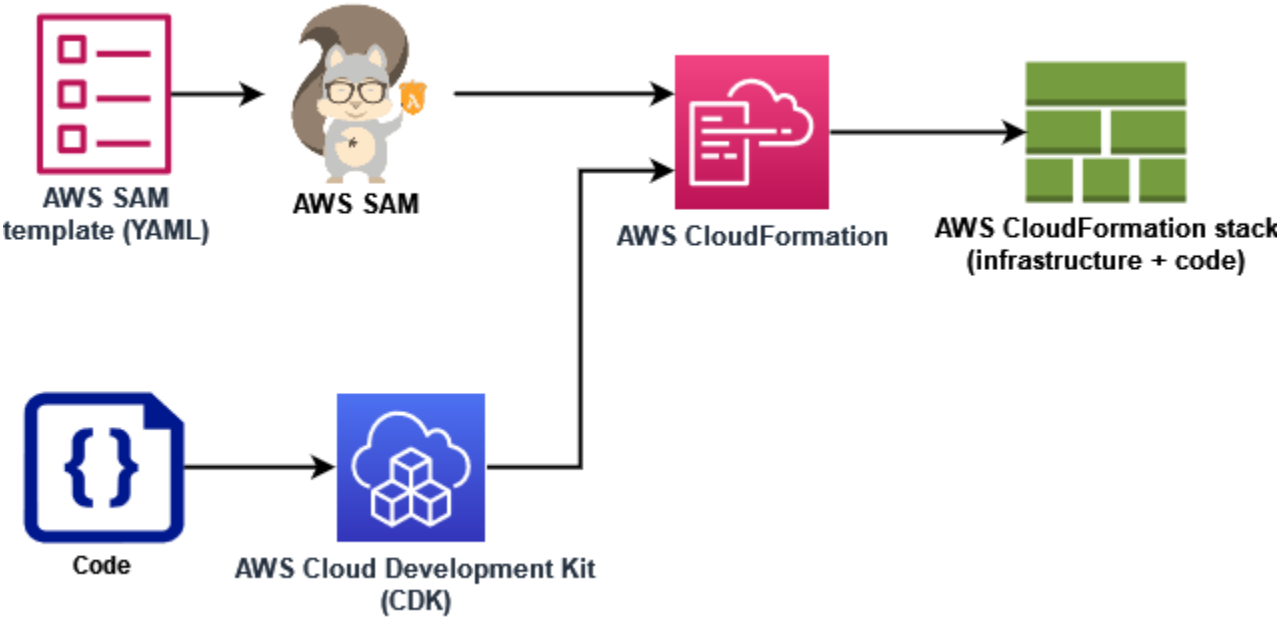
주제

Lambda용 IaC 도구

IaC를 사용하여 Lambda 함수 및 서버리스 애플리케이션을 배포할 수 있도록 AWS는 다양한 도구와 서비스를 제공합니다.

AWS CloudFormation은 클라우드 리소스를 생성하고 구성하기 위해 AWS에서 제공한 첫 번째 서비스입니다. AWS CloudFormation로 인프라와 코드를 정의하는 텍스트 템플릿을 생성할 수 있습니다. AWS가 새로운 서비스를 더 많이 도입하고 AWS CloudFormation 템플릿 생성의 복잡성이 증가함에 따라 두 가지 도구가 추가로 출시되었습니다. AWS SAM은 서버리스 애플리케이션을 정의하기 위한 새로운 템플릿 기반 프레임워크입니다. AWS Cloud Development Kit (AWS CDK)은 널리 사용되는 여러 프로그래밍 언어의 코드 구조를 사용하여 인프라를 정의하고 프로비저닝하기 위한 코드 우선 접근법입니다.

AWS SAM, AWS CDK 모두를 사용하면 AWS CloudFormation은 백그라운드에서 작동하여 인프라를 구축하고 배포할 수 있습니다. 다음 다이어그램은 이러한 도구 간의 관계를 보여주며 다이어그램 다음 단락에서 주요 기능을 설명합니다.



- AWS CloudFormation- 리소스와 CloudFormation 속성을 설명하는 YAML 또는 JSON 템플릿을 사용하여 AWS 리소스를 모델링하고 설정합니다. CloudFormation 리소스를 안전하고 반복 가능한 방식으로 프로비저닝하므로 수동 단계 없이 인프라와 애플리케이션을 자주 구축할 수 있습니다. 구성을 변경할 때 스택을 업데이트하기 위해 수행할 적절한 작업을 CloudFormation 결정합니다. CloudFormation 변경 내용을 롤백할 수도 있습니다.
- AWS Serverless Application Model(AWS SAM) - AWS SAM은 서버리스 애플리케이션을 정의하기 위한 오픈 소스 프레임워크입니다. AWS SAM 템플릿은 속기 구문을 사용하여 리소스당 몇 줄의 텍스트(YAML)만으로 함수, API, 데이터베이스 및 이벤트 소스 매핑을 정의합니다. 배포 중에 AWS SAM은 AWS SAM 구문을 AWS CloudFormation 구문으로 변환하고 확장합니다. 따라서 모든 CloudFormation 구문을 AWS SAM 템플릿에 추가할 수 있습니다. 이렇게 하면 AWS SAM 모든 기능을 사용할 수 CloudFormation 있지만 구성 줄 수는 더 적습니다.
- AWS Cloud Development Kit (AWS CDK)- 를 AWS CDK 사용하면 코드 구조를 사용하여 인프라를 정의하고 이를 통해 AWS CloudFormation 프로비전할 수 있습니다. AWS CDK기존 IDE TypeScript, 테스트 도구 및 워크플로 패턴을 사용하여 Python, Java, .NET 및 Go (개발자 미리 보기) 를 사용하여 애플리케이션 인프라를 모델링할 수 있습니다. 반복 가능한 배포, 간편한 롤백, 드리프트 감지를 비롯한 AWS CloudFormation의 모든 이점을 얻을 수 있습니다.

AWS는 또한 간단한 그래픽 인터페이스를 사용하여 IaC 템플릿을 개발하기 위한 AWS Application Composer이라는 서비스를 제공합니다. Application Composer를 사용하여 시각적 캔버스에서 AWS 서비스를 드래그, 그룹화 및 연결하여 애플리케이션 아키텍처를 설계합니다. 그런 다음 Application

Composer는 디자인을 기반으로 응용 프로그램을 배포하는 데 사용할 수 있는 AWS SAM 템플릿 또는 AWS CloudFormation 템플릿을 생성합니다.

아래 the section called “Lambda용 IaC 시작하기” 섹션에서는 Application Composer를 사용하여 기존 Lambda 함수를 기반으로 서버리스 애플리케이션을 위한 템플릿을 개발합니다.

Lambda용 IaC 시작하기

이 자습서에서는 기존 Lambda 함수에서 AWS SAM 템플릿을 생성한 다음 다른 AWS 리소스를 추가하여 Application Composer에서 서버리스 애플리케이션을 구축함으로써 Lambda와 함께 IaC를 사용하기 시작할 수 있습니다.

Application Composer를 사용하지 않고 템플릿을 사용하는 방법을 배우기 위해 AWS SAM 또는 AWS CloudFormation 자습서를 실행하여 시작하려는 경우 이 페이지 끝에 있는 the section called “다음 단계” 섹션에서 다른 리소스로 연결되는 링크를 찾을 수 있습니다.

이 자습서를 수행하면서 AWS 리소스가 AWS SAM에 지정되는 방식과 같은 몇 가지 기본 개념을 배우게 됩니다. 또한 Application Composer를 사용하여 AWS SAM 또는 AWS CloudFormation을 사용하여 배포할 수 있는 서버리스 애플리케이션을 빌드하는 방법도 배우게 됩니다.

이 자습서를 완료하려면 다음 단계를 수행하게 됩니다.

- Lambda 함수 예를 생성합니다.
- Lambda 콘솔을 사용하여 함수의 AWS SAM 템플릿 확인
- 함수 구성을 AWS Application Composer으로 내보내기하고 함수 구성을 기반으로 간단한 서버리스 애플리케이션 설계
- 서버리스 애플리케이션을 배포하기 위한 기반으로 사용할 수 있는 업데이트된 AWS SAM 템플릿 저장

이 the section called “다음 단계” 섹션에서는 AWS SAM 및 Application Composer에 대해 자세히 알아보는 데 사용할 수 있는 리소스를 찾을 수 있습니다. 이러한 리소스에는 AWS SAM을 사용하여 서버리스 애플리케이션을 배포하는 방법을 알려주는 고급 자습서 링크가 포함되어 있습니다.

사전 조건

이 자습서에서는 Application Composer의 로컬 동기화 기능을 사용하여 템플릿과 코드 파일을 로컬 빌드 컴퓨터에 저장합니다. 이 기능을 사용하려면 웹 애플리케이션이 로컬 파일 시스템에서 파일을 읽고 쓰고 저장할 수 있도록 하는 File System Access API를 지원하는 브라우저가 필요합니다. 구글 크롬이

나 Microsoft Edge를 사용하는 것이 좋습니다. 파일 시스템 액세스 API에 대한 자세한 내용은 파일 시스템 액세스 API란 무엇일까요?를 참조하세요.

Lambda 함수 생성

이 첫 단계에서는 자습서의 나머지 부분을 완료하는 데 사용할 수 있는 Lambda 함수를 생성합니다. 작업을 단순화하기 위해 Lambda 콘솔을 사용하여 Python 3.11 런타임을 사용하여 기본 'Hello world' 함수를 생성합니다.

콘솔을 사용하여 'Hello world' Lambda 함수 생성

1. Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. 처음부터 작성을 선택한 상태로 두고, 기본 정보에서 **LambdaIaCDemo**를 함수 이름에 입력합니다.
4. 런타임에서 Python 3.11을 선택합니다.
5. 함수 생성을 선택합니다.

함수에 대한 AWS SAM 템플릿 보기

함수 구성을 Application Composer로 내보내기 전에 Lambda 콘솔을 사용하여 함수의 현재 구성을 AWS SAM 템플릿으로 확인합니다. 이 섹션의 단계를 수행하면 AWS SAM 템플릿의 구조 및 Lambda 함수와 같은 리소스를 정의하여 서버리스 애플리케이션 지정을 시작하는 방법을 알아볼 수 있습니다.

함수에 대한 AWS SAM 템플릿 보기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 앞에서 생성한 함수(`LambdaIaCDemo`)를 선택합니다.
3. 함수 개요 창에서 템플릿을 선택합니다.

   함수 구성을 나타내는 다이어그램 대신 함수의 AWS SAM 템플릿이 표시됩니다. 템플릿은 다음과 같아야 합니다.

```
# 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:*
```

함수의 YAML 템플릿을 살펴보고 몇 가지 주요 개념을 이해하는 시간을 갖도록 하겠습니다.

템플릿은 `Transform: AWS::Serverless-2016-10-31` 선언으로 시작합니다. AWS SAM 템플릿은 백그라운드에서 AWS CloudFormation을 통해 배포되기 때문에 이 선언이 필요합니다. `Transform` 문을 사용하면 템플릿이 AWS SAM 템플릿 파일로 식별됩니다.

`Transform` 선언 다음에는 `Resources` 섹션이 나타납니다. AWS 리소스가 함께 배포하려는 AWS SAM 템플릿과 여기에 정의됩니다. AWS SAM 템플릿에는 AWS SAM 리소스와 AWS CloudFormation 리소스의 조합이 포함될 수 있습니다. 배포 중에 AWS SAM 템플릿이 AWS CloudFormation 템플릿으로 확장돼, 유효한 모든 AWS CloudFormation 구문을 AWS SAM 템플릿에 추가할 수 있기 때문입니다.

현재 템플릿 `Resources` 섹션에 정의된 리소스는 Lambda 함수 `LambdaIaCDemo`뿐입니다. AWS SAM 템플릿에 Lambda 함수를 추가하려면 `AWS::Serverless::Function` 리소스 유형을 사용합니다. Lambda 함수의 `Properties` 리소스는 함수의 런타임, 함수 핸들러 및 기타 구성 옵션을 정의합니다. 함수를 배포하는 데 AWS SAM이 사용해야 하는 함수 소스 코드의 경로도 여기에 정의되어 있습니다. 의 Lambda 함수 AWS SAM 리소스에 대해 자세히 알아보려면 개발자 안내서를 AWS::Serverless::Function참조하십시오. AWS SAM

템플릿은 함수 속성 및 구성뿐만 아니라 함수에 대한 AWS Identity and Access Management(IAM) 정책도 지정합니다. 이 정책은 Amazon CloudWatch Logs에 로그를 쓸 수 있는 권한을 함수에 부여합니다. Lambda 콘솔에서 함수를 생성하면 Lambda는 이 정책을 함수에 자동으로 연결합니다. AWS SAM템플릿의 함수에 대한 IAM 정책을 지정하는 방법에 대한 자세한 내용은 AWS SAM개발자 안내서 AWS::Serverless::Function페이지의 `policies` 속성을 참조하십시오.

AWS SAM 템플릿 구조에 대한 자세한 내용은 AWS SAM 템플릿 분석을 참조하세요.

AWS Application Composer을 서버리스 애플리케이션을 설계하는 데 사용합니다.

함수의 AWS SAM 템플릿을 출발점으로 사용하여 간단한 서버리스 애플리케이션을 구축하려면 함수 구성을 Application Composer로 내보내고 Application Composer의 로컬 동기화 모드를 활성화해야 합니다. 로컬 동기화는 함수의 코드와 AWS SAM 템플릿을 로컬 빌드 컴퓨터에 자동으로 저장하고, Application Composer에 다른 AWS 리소스를 추가할 때 저장된 템플릿이 동기화된 상태로 유지됩니다.

함수를 Application Composer로 내보내기

1. 함수 개요 창에서 Application Composer로 내보내기를 선택합니다.

   함수의 구성 및 코드를 Application Composer로 내보내기 위해 Lambda는 계정에 Amazon S3 버킷을 생성하여 이 데이터를 임시로 저장합니다.
2. 대화 상자에서 확인 및 프로젝트 생성을 선택하여 이 버킷의 기본 이름을 수락하고 함수의 구성 및 코드를 Application Composer로 내보내기합니다.

3. (선택 사항) Lambda가 생성하는 Amazon S3 버킷의 다른 이름을 선택하려면 새 이름을 입력하고 확인 및 프로젝트 생성을 선택합니다. Amazon S3 버킷에 이름은 전역적으로 고유해야 하며 버킷 이름 지정 규칙을 따라야 합니다.

   프로젝트 확인 및 생성을 선택하면 애플리케이션 컴포저 콘솔이 열립니다. 캔버스에서 Lambda 함수를 확인할 수 있습니다.

4. 메뉴 드롭다운에서 로컬 동기화 활성화를 선택합니다.
5. 이때 열리는 대화 상자에서 폴더 선택을 선택하고 로컬 빌드 머신에서 폴더를 선택합니다.
6. 활성화를 선택하여 로컬 동기화를 활성화합니다.

함수를 Application Composer로 내보내려면 특정 API 작업을 사용할 권한이 필요합니다. 함수를 내보낼 수 없는 경우 the section called “필요한 권한” 섹션을 참조하고, 필요한 권한이 있는지 확인합니다.

> **Note**
>
> 함수를 Application Composer로 내보낼 때 Lambda가 생성하는 버킷에는 표준 Amazon S3 요금이 적용됩니다. Lambda가 버킷에 넣은 객체는 10일 후에 자동으로 삭제되지만 Lambda는 버킷 자체를 삭제하지 않습니다.
> AWS 계정에 추가되어 추가 요금이 부과되지 않도록 하려면 함수를 Application Composer로 내보내기한 후 버킷 삭제의 지침을 따릅니다. Lambda가 생성하는 Amazon S3 버킷에 대한 자세한 내용은 the section called “Application Composer”를 참조하세요.

Application Composer에서 서버리스 애플리케이션 설계하기

로컬 동기화를 활성화한 후 Application Composer에서 변경한 내용은 로컬 빌드 컴퓨터에 저장된 AWS SAM 템플릿에 반영됩니다. 이제 추가 AWS 리소스를 Application Composer 캔버스에 드래그 앤 드롭하여 애플리케이션을 빌드할 수 있습니다. 이 예시에서는 Lambda 함수의 트리거로 Amazon SQS 단순 대기열을 추가하고, 데이터를 쓸 함수의 DynamoDB 테이블을 추가합니다.

1. 다음 작업을 수행하여 Lambda 함수에 Amazon SQS 트리거를 추가합니다.

   a. 리소스 팔레트의 검색 필드에 **SQS**를 입력합니다.

   b. SQS Queue 리소스를 캔버스로 드래그하여 Lambda 함수 왼쪽에 배치합니다.

   c. 세부 정보를 선택하고 논리적 ID에 **LambdaIaCQueue**를 입력합니다.

   d. 저장을 선택합니다.

e. SQS 대기열 카드에서 구독 포트를 클릭하고 Lambda 함수 카드의 왼쪽 포트로 드래그하여 Amazon SQS와 Lambda 리소스를 연결합니다. 두 리소스 사이에 선이 표시되면 연결이 성공했다는 의미입니다. Application Composer는 또한 캔버스 아래쪽에 두 리소스가 성공적으로 연결되었음을 알리는 메시지를 표시합니다.

2. 다음을 수행하여 Lambda 함수가 데이터를 작성할 수 있도록 Amazon DynamoDB 테이블을 추가합니다.

   a. 리소스 팔레트의 검색 필드에 **DynamoDB**를 입력합니다.

   b. DynamoDB 테이블 리소스를 캔버스로 드래그하여 Lambda 함수 오른쪽에 배치합니다.

   c. 세부 정보를 선택하고 논리적 ID에 **LambdaIaCTable**를 입력합니다.

   d. 저장을 선택합니다.

   e. Lambda 함수 카드의 오른쪽 포트를 클릭하고 DynamoDB 카드의 왼쪽 포트로 드래그하여 DynamoDB 테이블을 Lambda 함수에 연결합니다.

이제 이러한 추가 리소스를 추가했으니 Application Composer가 생성한 업데이트된 AWS SAM 템플릿을 살펴보겠습니다.

업데이트된 AWS SAM 템플릿 보기

- 응용 프로그램 작성기 캔버스에서 템플릿을 선택하여 캔버스 보기에서 템플릿 보기로 전환합니다.

이제 AWS SAM 템플릿에 다음과 같은 추가 리소스 및 속성이 포함되어야 합니다.

- `LambdaIaCQueue` 식별자를 사용하는 Amazon SQS 대기열

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

애플리케이션 컴포저를 사용하여 Amazon SQS 대기열을 추가하면 애플리케이션 컴포저가 `MessageRetentionPeriod` 속성을 설정합니다. `FifoQueue` 속성을 SQS 대기열 카드에서 세부 정보를 선택하고 Fifo 대기열을 선택하거나 선택 취소하여 설정할 수도 있습니다.

대기열에 다른 속성을 설정하려면 템플릿을 수동으로 편집하여 추가할 수 있습니다. `AWS::SQS::Queue` 리소스 및 사용 가능한 속성에 대해 자세히 알아보려면 AWS::SQS::Queue를 AWS CloudFormation 사용 설명서에서 참조하십시오.

- Amazon SQS 대기열을 함수에 대한 트리거로 지정하는 Lambda 함수 정의의 `Events` 속성입니다.

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

`Events` 속성은 이벤트 유형과 유형에 따라 달라지는 속성 세트로 구성됩니다. Lambda 함수를 트리거하도록 구성할 수 있는 AWS 서비스 차이점과 설정할 수 있는 속성에 대해 알아보려면 개발자 안내서를 EventSource참조하십시오. AWS SAM

- `LambdaIaCTable` 식별자가 있는 DynamoDB 테이블

```
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 Composer를 사용하여 DynamoDB 테이블을 추가할 때 DynamoDB 테이블 카드에서 세부 정보를 선택하고 키 값을 편집하여 테이블의 키를 설정할 수 있습니다. 또한 애플리케이션 컴포저는 `BillingMode` 및 `StreamViewType`을 비롯한 여러 다른 속성에 대한 기본값을 설정합니다.

AWS SAM 템플릿에 추가할 수 있는 이러한 속성과 기타 속성에 대해 자세히 알아보려면 AWS::DynamoDB::Table을 AWS CloudFormation사용 설명서에서 참조하십시오.

- 추가한 DynamoDB 테이블에서 CRUD 작업을 수행할 수 있는 권한을 함수에 부여하는 새로운 IAM 정책입니다.

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

완성된 최종 AWS SAM 템플릿은 다음과 같아야 합니다.

```
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
```

AWS SAM을 사용하여 서버리스 애플리케이션을 배포합니다(선택 사항).

Application Composer에서 방금 생성한 템플릿을 사용하여 AWS SAM을 서버리스 애플리케이션을 배포하는 데 사용하려면 먼저 AWS SAM 및 CLI를 설치해야 합니다. 그러려면 AWS SAM CLI 설치하기의 지침을 따릅니다.

애플리케이션을 배포하기 전에 Application Composer가 템플릿과 함께 저장한 함수 코드도 업데이트해야 합니다. 현재 애플리케이션 컴포저가 저장한 `lambda_function.py` 파일에는 함수를 생성할 때 Lambda가 제공한 기본 'Hello world' 코드만 포함되어 있습니다.

함수 코드를 업데이트하려면 다음 코드를 복사하여 Application Composer가 로컬 빌드 컴퓨터에 저장한 `lambda_function.py` 파일에 붙여넣습니다. 로컬 동기화 모드를 활성화할 때 Application Composer에서 이 파일을 저장할 디렉토리를 지정했습니다.

이 코드는 애플리케이션 컴포저에서 생성한 Amazon SQS 대기열의 메시지에 있는 키 값 쌍을 수락합니다. 키와 값이 모두 문자열인 경우 코드는 이를 사용하여 템플릿에 정의된 DynamoDB 테이블에 항목을 작성합니다.

Python 함수 코드 업데이트

```
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
```

서버리스 애플리케이션 배포하기

AWS SAM 및 CLI를 사용하여 애플리케이션을 배포하려면 다음 단계를 수행하세요. 함수를 올바르게 빌드하고 배포하려면 Python 버전 3.11이 빌드 머신과 PATH에 설치되어 있어야 합니다.

1. Application Composer가 `template.yaml` 및 `lambda_function.py` 파일을 저장한 디렉터리에서 다음 명령을 실행합니다.

   ```
   sam build
   ```

   이 명령은 응용 프로그램의 빌드 아티팩트를 수집하여 배포에 적합한 형식과 위치에 배치합니다.

2. 애플리케이션을 배포하고 AWS SAM 템플릿에 지정된 Lambda, Amazon SQS 및 DynamoDB 리소스를 생성하려면 다음 명령을 실행합니다.

   ```
   sam deploy --guided
   ```

   `--guided` 플래그를 사용하면 배포 AWS SAM 프로세스를 안내하는 메시지가 표시됩니다. 이 배포의 경우 Enter를 눌러 기본 옵션을 수락합니다.

배포 프로세스 중에 AWS SAM은 다음 리소스를 AWS 계정 위치에 생성합니다.

- AWS CloudFormation 스택의 이름은 `sam-app`입니다.
- `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`* 이름 형식으로 된 Lambda 함수의 이름입니다.
- `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`* 이름 형식을 사용하는 Amazon SQS 대기열
- `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`* 이름 형식이 지정된 DynamoDB 테이블

AWS SAM은 또한 Lambda 함수가 Amazon SQS 대기열에서 메시지를 읽고 DynamoDB 테이블에서 CRUD 작업을 수행할 수 있도록 필요한 IAM 역할 및 정책을 생성합니다.

AWS SAM을 사용하여 서버리스 애플리케이션을 배포하는 방법에 대해 자세히 알아보려면 the section called “다음 단계” 섹션의 리소스를 참조하세요.

배포된 애플리케이션 테스트(선택 사항)

서버리스 애플리케이션이 올바르게 배포되었는지 확인하려면 키 값 쌍이 포함된 메시지를 Amazon SQS 대기열로 전송하고 Lambda가 이 값을 사용하여 DynamoDB 테이블에 항목을 기록하는지 확인합니다.

서버리스 애플리케이션 테스트하기

1. Amazon SQS 콘솔의 대기열 페이지를 열고 AWS SAM가 템플릿에서 생성한 대기열을 선택합니다. 이름의 형식은 `sam-app-LambdaIaCQueue-xL87VeKsGiIo`입니다.
2. 메시지 전송 및 수신을 선택하고 다음 JSON을 메시지 전송 섹션의 메시지 본문에 붙여넣습니다.

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

3. 메시지 전송을 선택합니다.

   메시지를 대기열로 보내면 Lambda가 이벤트 소스 매핑을 통해 AWS SAM 템플릿에 정의된 함수를 간접 호출합니다. Lambda에서 예상대로 함수를 간접 호출했는지 확인하려면 항목이 DynamoDB 테이블에 추가되었는지 확인합니다.

4. DynamoDB 콘솔의 표 페이지를 열고 표를 선택합니다. 이름의 형식은 `sam-app-LambdaIaCTable-CN0S66C0VLNV`입니다.
5. 테이블 항목 탐색을 선택합니다. 반환된 항목 창에 ID가 `myKey`이고 값이 `myValue`인 항목이 표시되어야 합니다.

다음 단계

AWS SAM 및 AWS CloudFormation과 함께 애플리케이션 컴포저를 사용하는 방법에 대해 자세히 알아보려면 AWS CloudFormation 및 AWS SAM과 함께Application Composer 사용에서 시작합니다.

Application Composer에서 디자인한 서버리스 애플리케이션을 배포하는 데 AWS SAM을 사용하는 가이드 자습서를 보려면 AWS Application Composer자습서를 AWS 서버리스 패턴 워크숍에서 수행하는 것도 좋습니다.

AWS SAM은 명령줄 인터페이스(CLI)를 제공하며, 이는 AWS SAM 템플릿 및 지원되는 서드 파티 통합과 함께 사용하여 서버리스 애플리케이션을 빌드하고 실행할 수 있습니다. AWS SAM 및 CLI를 사용하여 애플리케이션을 빌드 및 배포하고, 로컬 테스트 및 디버깅을 수행하며, CI/CD 파이프라인을 구성하는 등의 작업을 수행할 수 있습니다. AWS SAM 및 CLI 사용에 대한 자세한 내용은 AWS SAM시작하기를 AWS Serverless Application Model개발자 안내서에서 참조하세요.

AWS SAM 템플릿로 AWS CloudFormation콘솔을 사용하여 서버리스 애플리케이션을 배포하는 방법을 알아보려면 AWS CloudFormation콘솔 사용을 AWS CloudFormation사용 설명서에서 시작하세요.

Application Composer와의 Lambda 통합이 지원되는 지역

Application Composer와의 Lambda 통합은 다음 AWS 리전에서 지원됩니다.

- 미국 동부(버지니아 북부)
- 미국 동부(오하이오)
- 미국 서부(캘리포니아 북부)
- 미국 서부(오레곤)
- 아프리카(케이프타운)
- 아시아 태평양(홍콩)
- 아시아 태평양(하이데라바드)
- 아시아 태평양(자카르타)
- 아시아 태평양(멜버른)
- 아시아 태평양(뭄바이)
- 아시아 태평양(오사카)
- 아시아 태평양(서울)
- 아시아 태평양(싱가포르)
- 아시아 태평양(시드니)
- 아시아 태평양(도쿄)
- 캐나다(중부)
- 유럽(프랑크푸르트)
- 유럽(취리히)
- 유럽(아일랜드)
- Europe (London)
- 유럽(스톡홀름)

- 중동(UAE)

VPC를 사용한 프라이빗 네트워킹

Amazon Virtual Private Cloud(Amazon VPC)는 AWS 계정 전용 AWS 클라우드의 가상 네트워크입니다. Amazon VPC를 사용하여 데이터베이스, 캐시 인스턴스 또는 내부 서비스 등과 같은 리소스에 대한 프라이빗 네트워크를 생성할 수 있습니다. Amazon VPC에 대한 자세한 내용은 Amazon VPC가 무엇인가요?를 참조하세요.

Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. Lambda는 이 VPC 네트워크 액세스 및 보안 규칙을 적용하고 Lambda는 VPC를 자동으로 유지 관리 및 모니터링합니다. Lambda 함수가 계정 VPC의 리소스에 액세스해야 하는 경우 VPC에 액세스하도록 함수를 구성합니다. Lambda는 Lambda 함수가 Lambda VPC에서 계정 VPC의 ENI(탄력적 네트워크 인터페이스)로 연결하는 데 사용하는 Hyperplane ENI라는 관리형 리소스를 제공합니다.

VPC 또는 Hyperplane ENI의 사용에 따르는 추가 요금은 없습니다. NAT 게이트웨이와 같은 일부 VPC 구성 요소에 대해 요금이 부과됩니다. 자세한 내용은 Amazon VPC 요금을 참조하세요.

주제

VPC 네트워크 요소

Amazon VPC 네트워크에는 다음과 같은 네트워크 요소가 포함됩니다.

- 탄력적 네트워크 인터페이스 - 탄력적 네트워크 인터페이스는 VPC에서 가상 네트워크 카드를 나타내는 논리적 네트워킹 구성 요소입니다.
- 서브넷 — VPC의 IP 주소 범위입니다. 지정된 서브넷으로 AWS 리소스를 추가할 수 있습니다. 인터넷에 연결되어야 하는 리소스에는 퍼블릭 서브넷을 사용하고, 인터넷에 연결되지 않는 리소스에는 프라이빗 서브넷을 사용합니다.

- 보안 그룹 – 보안 그룹을 사용하여 각 서브넷의 AWS 리소스에 대한 액세스를 제어합니다.
- 액세스 제어 목록(ACL) — 네트워크 ACL을 사용하여 서브넷에 추가 보안을 제공합니다. 기본 서브넷 ACL은 인바운드와 아웃바운드 트래픽을 모두 허용합니다.
- 라우팅 테이블 — VPC에 대한 네트워크 트래픽을 지시하는 데 AWS에서 사용하는 경로 집합을 포함합니다. 서브넷을 특정 라우팅 테이블과 명시적으로 연결할 수 있습니다. 기본적으로 서브넷은 기본 라우팅 테이블과 연결됩니다.
- 경로 — 라우팅 테이블의 각 경로는 IP 주소 범위와 Lambda가 해당 범위에 대한 트래픽을 전송하는 대상을 지정합니다. 또한 경로는 트래픽을 전송할 게이트웨이, 네트워크 인터페이스 또는 연결인 대상을 지정합니다.
- NAT 게이트웨이 — 프라이빗 VPC 프라이빗 서브넷에서 인터넷으로의 액세스를 제어하는 AWS 네트워크 주소 변환(NAT) 서비스입니다.
- VPC 엔드포인트 – Amazon VPC 엔드포인트를 사용하여 인터넷이나 NAT 장치, VPN 연결 또는 AWS Direct Connect 연결을 통해 액세스할 필요 없이 AWS에서 호스팅되는 서비스에 대한 프라이빗 연결을 생성할 수 있습니다. 자세한 내용은 AWS PrivateLink 및 VPC 엔드포인트를 참조하십시오.

> **Tip**
>
> VPC 및 서브넷에 액세스하도록 Lambda 함수를 구성하려면 Lambda 콘솔 또는 API를 사용하면 됩니다.
> 함수를 구성하려면 CreateFunction의 `VpcConfig` 섹션을 참조하세요. 자세한 단계는 Lambda 함수를 AWS 계정의 Amazon VPC에 연결하기 섹션을 참조하세요.

Amazon VPC 네트워킹 정의에 대한 자세한 내용은 Amazon VPC 개발자 가이드의 Amazon VPC 작동 방식과 Amazon VPC 자주 묻는 질문을 참조하십시오.

Lambda 함수를 VPC에 연결

Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. 기본적으로 Lambda 함수는 사용자 계정의 VPC에 연결되지 않습니다. 계정에서 VPC에 함수를 연결할 때 VPC가 액세스 권한을 제공하지 않으면 함수는 인터넷에 액세스할 수 없습니다.

Lambda는 Hyperplane ENI를 사용하여 VPC에서 리소스에 액세스합니다. Hyperplane ENI는 VPC-VPC 간 NAT(V2N)를 사용하여 Lambda VPC에서 계정 VPC로 NAT 기능을 제공합니다. V2N은 Lambda VPC에서 계정 VPC로의 연결을 제공하지만 다른 방향으로는 제공하지 않습니다.

Lambda 함수를 생성(혹은 VPC 설정을 업데이트)하면 Lambda는 함수의 VPC 구성에서 각 서브넷에 대해 Hyperplane ENI를 할당합니다. 함수가 동일한 서브넷과 보안 그룹을 공유하는 경우 여러 Lambda 함수가 네트워크 인터페이스를 공유할 수 있습니다.

다른 AWS 서비스에 연결하려면 VPC와 지원되는 AWS 서비스 간의 프라이빗 통신용 VPC 엔드포인트를 사용할 수 있습니다. 다른 접근 방식은 NAT 게이트웨이를 사용하여 아웃바운드 트래픽을 다른 AWS 서비스로 라우팅하는 것입니다.

함수에 인터넷 액세스 권한을 부여하려면 아웃바운드 트래픽을 퍼블릭 서브넷의 NAT 게이트웨이로 라우팅합니다. NAT 게이트웨이는 퍼블릭 IP 주소가 있으므로 VPC의 인터넷 게이트웨이를 통해 인터넷에 연결할 수 있습니다. 자세한 설명은 VPC 연결 Lambda 함수에 대한 인터넷 액세스 활성화을 참조하세요.

공유 서브넷

VPC 공유를 통해 여러 AWS 계정이 중앙에서 관리되는 공유 Virtual Private Cloud(VPC)에서 Amazon EC2 인스턴스 및 Lambda 함수와 같은 애플리케이션 리소스를 생성할 수 있습니다. 이 모델에서 VPC를 소유한 계정(소유자)은 동일한 AWS 조직에 속한 다른 계정(참가자)과 하나 이상의 서브넷을 공유합니다.

프라이빗 리소스에 액세스하려면 함수를 VPC의 프라이빗 공유 서브넷에 연결합니다. 서브넷 소유자는 서브넷에 함수를 연결하기 전에 서브넷을 공유해야 합니다. 서브넷 소유자는 나중에 서브넷 공유를 해제하여 연결을 제거할 수도 있습니다. 공유 서브넷에서 VPC 리소스를 공유, 공유 해제 및 관리하는 방법에 대한 자세한 내용은 Amazon VPC 가이드에서 다른 계정과 VPC 공유를 참조하세요.

Lambda Hyperplane ENI

Hyperplane ENI는 Lambda 서비스에서 생성 및 관리하는 관리형 네트워크 리소스입니다. Lambda VPC의 여러 실행 환경에서는 Hyperplane ENI를 사용하여 계정의 VPC 내부의 리소스에 안전하게 액세스할 수 있습니다. Hyperplane ENI는 Lambda VPC에서 계정 VPC로 NAT 기능을 제공합니다.

각 서브넷에 있어 Lambda는 각 고유한 보안 그룹 세트에 대해 네트워크 인터페이스를 생성합니다. 동일한 서브넷 및 보안 그룹 조합을 공유하는 계정의 함수는 동일한 네트워크 인터페이스를 사용합니다. 하이퍼플레인 계층을 통해 이루어진 연결은 보안 그룹 구성에 추적하도록 별도로 설정되어 있지 않더라도 자동으로 추적됩니다. 설정된 연결과 일치하지 않는 VPC의 인바운드 패킷은 하이퍼플레인 계층에서 삭제됩니다. 자세한 내용은 Amazon EC2 사용 설명서의 보안 그룹 연결 추적을 참조하세요.

계정의 함수가 ENI 리소스를 공유하므로 ENI 수명 주기는 다른 Lambda 리소스보다 복잡합니다. 다음 섹션에서는 ENI 수명 주기에 대해 설명합니다.

ENI 수명 주기

- ENI 생성
- ENI 관리
- ENI 삭제

ENI 생성

Lambda는 새로 생성된 VPC 지원 함수 또는 기존 함수에 대한 VPC 구성 변경을 위해 Hyperplane ENI 리소스를 생성할 수 있습니다. Lambda가 필요한 리소스를 생성하는 동안 함수는 보류 중인 상태로 유지됩니다. Hyperplane ENI가 준비되면 함수가 활성 상태로 전환되고 ENI를 사용할 수 있게 됩니다. Lambda가 Hyperplane ENI를 생성하는 데 몇 분이 소요될 수 있습니다.

새로 생성된 VPC 지원 함수의 경우 함수 상태가 활성으로 전환될 때까지 함수에서 작동하는 호출이나 기타 API 작업이 실패합니다.

기존 함수에 대한 VPC 구성 변경의 경우 모든 함수 호출은 함수 상태가 활성으로 전환될 때까지 이전 서브넷 및 보안 그룹 구성과 연결된 Hyperplane ENI를 계속 사용합니다.

Lambda 함수가 연속으로 30일 동안 유휴 상태로 유지되면 Lambda는 사용되지 않는 Hyperplane ENI를 회수하고 함수 상태를 유휴 상태로 설정합니다. 다음 호출은 Lambda가 유휴 함수를 다시 활성화하도록 합니다. 호출이 실패하고 Lambda가 Hyperplane ENI의 생성 또는 할당을 완료할 때까지 함수가 보류 상태로 전환됩니다.

함수 상태에 대한 자세한 내용은 Lambda 함수 상태 단원을 참조하세요.

ENI 관리

Lambda는 함수의 실행 역할에 있는 권한을 사용하여 네트워크 인터페이스를 생성하고 관리합니다. Lambda는 계정에서 VPC 지원 함수에 대한 고유한 서브넷과 보안 그룹 조합을 정의할 때 Hyperplane ENI를 생성합니다. Lambda는 계정에서 동일한 서브넷과 보안 그룹 조합을 사용하는 다른 VPC 지원 함수에 대해 Hyperplane ENI를 재사용합니다.

동일한 Hyperplane ENI를 사용할 수 있는 Lambda 함수 수에는 할당량이 없습니다. 그러나 각 Hyperplane ENI는 최대 65,000개의 연결/포트를 지원합니다. 연결 수가 65,000개를 초과하는 경우 Lambda는 추가 연결을 제공하기 위해 새 Hyperplane ENI를 생성합니다.

다른 VPC에 액세스하도록 함수 구성을 업데이트하면 Lambda가 이전 VPC의 Hyperplane ENI에 대한 연결을 종료합니다. 새 VPC 대한 연결을 업데이트하는 프로세스는 몇 분 정도 걸릴 수 있습니다. 이 시간 동안 함수에 대한 호출은 이전 VPC를 계속 사용합니다. 업데이트가 완료되면 새 VPC의

Hyperplane ENI를 사용하여 새 호출이 시작됩니다. 이 시점에서 Lambda 함수는 더 이상 이전 VPC와 연결되지 않습니다.

ENI 삭제

VPC 구성을 제거하기 위해 함수를 업데이트하는 경우 Lambda에서 연결된 Hyperplane ENI를 삭제하는 데 최대 20분이 소요됩니다. Lambda는 다른 함수(또는 게시된 함수 버전)가 해당 Hyperplane ENI를 사용하고 있지 않은 경우에만 ENI를 삭제합니다.

Lambda는 함수 실행 역할의 권한에 따라 Hyperplane ENI를 삭제합니다. Lambda가 Hyperplane ENI를 삭제하기 전에 실행 역할을 삭제하면 Lambda가 Hyperplane ENI를 삭제할 수 없습니다. 수동으로 삭제를 수행할 수 있습니다.

Lambda는 계정의 함수 또는 함수 버전에서 사용 중인 네트워크 인터페이스를 삭제하지 않습니다. Lambda ENI Finder를 사용하여 Hyperplane ENI를 사용하는 함수 또는 함수 버전을 식별할 수 있습니다. 더 이상 필요하지 않은 함수 또는 함수 버전의 경우 Lambda가 Hyperplane ENI를 삭제하도록 VPC 구성을 제거할 수 있습니다.

연결

Lambda는 TCP(전송 제어 프로토콜)와 UDP(사용자 데이터그램 프로토콜)의 두 가지 유형의 연결을 지원합니다.

VPC를 생성하면 Lambda는 자동으로 DHCP 옵션 세트를 생성하여 VPC와 연결합니다. VPC에 대해 고유한 DHCP 옵션 세트를 구성할 수 있습니다. 자세한 내용은 Amazon VPC DHCP 옵션을 참조하세요.

Amazon은 VPC용 DNS 서버(Amazon Route 53 Resolver)를 제공합니다. 자세한 내용은 VPC에 대한 DNS 지원을 참조하세요.

IPv6 지원

Lambda는 Lambda의 퍼블릭 듀얼 스택 엔드포인트에 대한 인바운드 연결과 IPv6를 통한 듀얼 스택 VPC 서브넷으로의 아웃바운드 연결을 지원합니다.

인바운드

IPv6를 통해 함수를 호출하려면 Lambda의 퍼블릭 듀얼 스택 엔드포인트를 사용하세요. IPv4 및 IPv6를 모두 지원하는 듀얼 스택 엔드포인트입니다. Lambda 듀얼 스택 엔드포인트는 다음 구문을 사용합니다.

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

Lambda 함수 URL를 사용하여 IPv6를 통해 함수를 호출할 수도 있습니다. 함수 URL 엔드포인트는 다음 형식을 취합니다.

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

아웃바운드

함수는 IPv6를 통해 듀얼 스택 VPC 서브넷의 리소스에 연결할 수 있습니다. 이 옵션은 기본적으로 꺼져 있습니다. 아웃바운드 IPv6 트래픽을 허용하려면 콘솔을 사용하거나 create-function 또는 update-function-configuration 명령과 함께 `--vpc-config Ipv6AllowedForDualStack=true` 옵션을 사용하세요.

> **Note**
>
> VPC에서 아웃바운드 IPv6 트래픽을 허용하려면 함수에 연결된 모든 서브넷이 듀얼 스택 서브넷이어야 합니다. Lambda는 VPC의 IPv6 전용 서브넷을 위한 아웃바운드 IPv6 연결, VPC에 연결되지 않은 함수를 위한 아웃바운드 IPv6 연결 또는 VPC 엔드포인트(AWS PrivateLink)를 사용하는 인바운드 IPv6 연결을 지원하지 않습니다.

IPv6를 통해 서브넷 리소스에 명시적으로 연결하도록 함수 코드를 업데이트할 수 있습니다. 다음 Python 예제는 소켓을 열고 IPv6 서버에 연결합니다.

Example - IPv6 서버에 연결

```
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()
```

보안

AWS는 VPC의 보안을 강화하기 위해 보안 그룹과 네트워크 ACL을 제공합니다. 보안 그룹은 리소스용 인바운드 및 아웃바운드 트래픽을 제어하고, 네트워크 ACL은 서브넷용 인바운드 및 아웃바운드 트래픽을 제어합니다. 보안 그룹은 대부분의 서브넷에 대해 충분한 액세스 제어를 제공합니다. VPC에 대한 추가 보안 계층을 원하는 경우 네트워크 ACL을 사용할 수 있습니다. 자세한 내용은 Amazon VPC의 인터네트워크 트래픽 개인 정보 보호를 참조하세요. 생성된 각 서브넷에는 자동으로 VPC의 기본 네트워크 ACL이 연결됩니다. 연결된 네트워크 ACL 및 기본 네트워크 ACL의 내용을 변경할 수 있습니다.

일반 보안 모범 사례는 VPC 보안 모범 사례를 참조하세요. IAM을 사용하여 Lambda API 및 리소스에 대한 액세스를 관리하는 방법에 대한 자세한 내용은 AWS Lambda사용 권한을 참조하세요.

VPC 설정에 Lambda 특정 조건 키를 사용하여 Lambda 함수에 대한 추가 권한 제어를 제공할 수 있습니다. VPC 조건 키에 대한 자세한 내용은 VPC 설정에 IAM 조건 키 사용을 참조하세요.

> **Note**
>
> Lambda 함수는 퍼블릭 인터넷 또는 AWS PrivateLink 엔드포인트에서 간접적으로 호출할 수 있습니다. 퍼블릭 인터넷을 통해서만 함수 URL에 액세스할 수 있습니다. Lambda 함수는 AWS PrivateLink를 지원하지만 함수 URL은 지원하지 않습니다.

관찰성

VPC 흐름 로그를 사용하여 VPC의 네트워크 인터페이스에서 송수신되는 IP 트래픽에 대한 정보를 캡처할 수 있습니다. 흐름 로그 데이터는 Amazon CloudWatch Logs 또는 Amazon S3에 게시할 수 있습니다. 흐름 로그를 생성한 다음 선택된 대상의 데이터를 가져와 확인할 수 있습니다.

참고: 함수를 VPC에 연결하면 CloudWatch 로그 메시지는 VPC 경로를 사용하지 않습니다. Lambda는 이를 로그에 대한 일반 라우팅을 사용하여 전송합니다.

Lambda 함수에 대한 명령 세트 아키텍처 구성

Lambda 함수의 명령 세트 아키텍처에 따라 Lambda가 함수를 실행하는 데 사용하는 컴퓨터 프로세서의 유형이 결정됩니다. Lambda는 다음과 같은 명령 세트 아키텍처를 선택할 수 있는 옵션을 제공합니다.

- arm64 - AWS Graviton2 프로세서에 사용되는 64비트 ARM 아키텍처입니다.
- x86_64 - x86 기반 프로세서에 사용되는 64비트 x86 아키텍처입니다.

> **Note**
>
> arm64 아키텍처는 대부분의AWS 리전에서 사용할 수 있습니다. 자세한 내용은 AWS Lambda 요금을 참조하세요. 메모리 가격표에서 Arm Price(ARN 가격) 탭을 선택한 다음 Region(리전) 드롭다운 목록을 열어 Lambda에서 arm64를 지원하는 AWS 리전을 확인합니다.
> arm64 아키텍처를 사용하여 함수를 생성하는 방법의 예는 AWS Lambda Graviton2 프로세서로 구동되는 AWS 함수를 참조하세요.

주제

arm64 아키텍처를 사용하는 데 따른 이점

arm64 아키텍처(AWS Graviton2 프로세서)를 사용하는 Lambda 함수는 x86_64 아키텍처에서 실행되는 동일한 함수보다 훨씬 우수한 가격 대비 성능을 실현할 수 있습니다. 고성능 컴퓨팅, 비디오 인코딩 및 시뮬레이션 워크로드와 같은 컴퓨팅 집약적 애플리케이션에는 arm64를 사용하는 것이 좋습니다.

Graviton2 CPU는 Neoverse N1 코어를 사용하며 Armv8.2(CRC 및 암호화 확장 포함)와 기타 여러 아키텍처 확장을 지원합니다.

Graviton2는 vCPU당 더 큰 L2 캐시를 제공하여 메모리 읽기 시간을 줄입니다. 따라서 웹 및 모바일 백엔드, 마이크로서비스 및 데이터 처리 시스템의 대기 시간 성능이 향상됩니다. 또한 Graviton2는 향상

된 암호화 성능을 제공하며 CPU 기반 기계 학습 추론의 대기 시간을 개선하는 명령 세트를 지원합니다.

AWS Graviton2에 대한 자세한 내용은 AWS Graviton 프로세서를 참조하세요.

arm64 아키텍처로 마이그레이션을 위한 요구 사항

arm64 아키텍처로 마이그레이션할 Lambda 함수를 선택할 때, 원활한 마이그레이션을 위해 함수가 다음 요구 사항을 충족하는지 확인합니다.

- 함수가 현재 Lambda Amazon Linux 2 런타임을 사용합니다.
- 배포 패키지에 사용자가 제어하는 오픈 소스 구성 요소 및 소스 코드만 포함되어 있어 사용자가 마이그레이션에 필요한 모든 업데이트를 수행할 수 있습니다.
- 함수 코드에 서드 파티 종속 구성 요소가 포함된 경우 각 라이브러리 또는 패키지가 arm64 버전을 제공합니다.

arm64 아키텍처와의 함수 코드의 호환성

Lambda 함수 코드는 함수의 명령 세트 아키텍처와 호환되어야 합니다. 함수를 arm64 아키텍처로 마이그레이션하기 전에 현재 함수 코드와 관련해 다음 사항에 유의합니다.

- 내장된 코드 편집기를 사용하여 함수 코드를 추가한 경우 코드는 수정 없이 두 아키텍처 중 하나에서 실행될 수 있습니다.
- 함수 코드를 업로드한 경우에는 대상 아키텍처와 호환되는 새 코드를 업로드해야 합니다.
- 함수가 계층을 사용하는 경우 각 계층이 새로운 아키텍처와 호환되는지 확인합니다. 계층이 호환되지 않는 경우 함수를 편집하여 현재 계층 버전을 호환되는 계층 버전으로 바꿉니다.
- 함수가 Lambda 확장을 사용하는 경우 각 확장이 새로운 아키텍처와 호환되는지 확인합니다.
- 함수가 컨테이너 이미지 배포 패키지 유형을 사용하는 경우 함수의 아키텍처와 호환되는 새 컨테이너 이미지를 만들어야 합니다.

arm64 아키텍처로 마이그레이션하는 방법

Lambda 함수를 arm64 아키텍처로 마이그레이션하려면 다음 단계를 수행하는 것이 좋습니다.

1. 애플리케이션 또는 워크로드의 종속 구성 요소 목록을 작성합니다. 일반적인 종속 구성 요소는 다음과 같습니다.

- 함수가 사용하는 모든 라이브러리와 패키지
- 컴파일러, 테스트 제품군, 지속적 통합 및 지속적 전달(CI/CD) 파이프라인, 프로비저닝 도구, 스크립트 등, 함수를 빌드, 배포 및 테스트하는 데 사용하는 도구
- 프로덕션 환경에서 함수를 모니터링하는 데 사용하는 Lambda 확장 및 서드 파티 도구

2. 각 종속 구성 요소별로 버전을 확인한 다음 arm64 버전을 사용할 수 있는지 확인합니다.
3. 새 환경을 빌드하여 애플리케이션을 마이그레이션합니다.
4. 애플리케이션을 부트스트랩합니다.
5. 애플리케이션을 테스트하고 디버깅합니다.
6. arm64 함수의 성능을 테스트합니다. x86_64 버전과 성능을 비교합니다.
7. arm64 Lambda 함수를 지원하도록 인프라 파이프라인을 업데이트합니다.
8. 배포를 프로덕션 환경에 스테이징합니다.

   예를 들어 별칭 라우팅 구성을 사용하여 함수의 x86 버전과 arm64 버전 간에 트래픽을 분할하고 성능과 대기 시간을 비교합니다.

Java, Go, .NET 및 Python에 대한 언어별 정보를 포함하여 arm64 아키텍처용 코드 환경을 만드는 방법에 대한 자세한 내용은 AWS Graviton 시작하기 GitHub 리포지토리를 참조하세요.

명령 세트 아키텍처 구성

Lambda 콘솔, AWS SDK, AWS Command Line Interface(AWS CLI) 또는 AWS CloudFormation을 사용하여 신규 및 기존 Lambda 함수의 명령 세트 아키텍처를 구성할 수 있습니다. 다음 단계에 따라 콘솔에서 기존 Lambda 함수의 명령 세트 아키텍처를 변경합니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 명령 세트 아키텍처를 구성할 함수의 이름을 선택합니다.
3. 기본 코드 탭의 런타임 설정 섹션에서 편집을 선택합니다.
4. Architecture(아키텍처)에서 함수에 사용할 명령 세트 아키텍처를 선택합니다.
5. Save(저장)를 선택합니다.

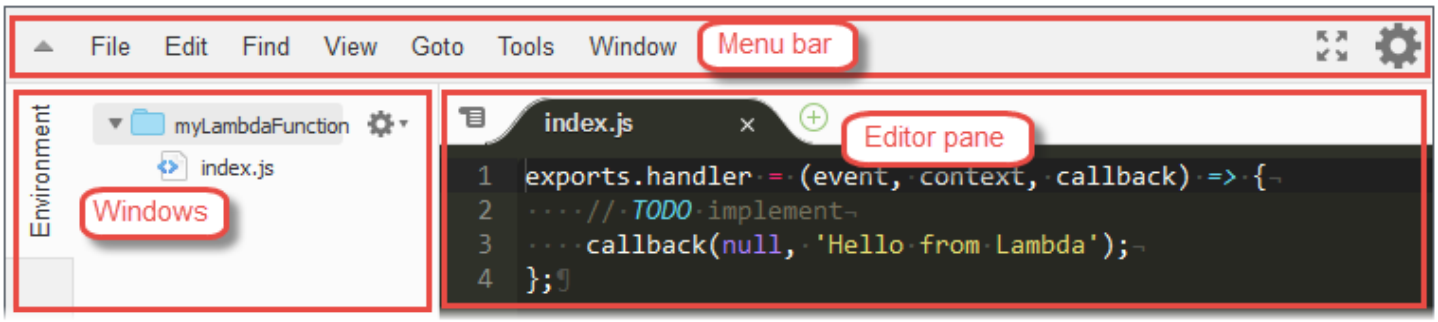
> **Note**
>
> 모든 Amazon Linux 2 런타임에서 x86_64 및 ARM CPU 아키텍처를 모두 지원합니다.

Go 1.x와 같이 Amazon Linux 2를 사용하지 않는 런타임은 arm64 아키텍처를 지원하지 않습니다. Go 1.x에서 arm64 아키텍처를 사용하려면 제공된 .al2 런타임에서 함수를 실행합니다. 자세한 내용은 .zip 패키지와 컨테이너 이미지의 배포 지침을 참조하세요.

Lambda 콘솔 편집기를 사용하여 코드 편집

Lambda 콘솔의 코드 편집기를 사용하여 Lambda 함수 코드를 작성하고 테스트하고 실행 결과를 볼 수 있습니다. Lambda 콘솔은 Node.js와 Python과 같이 컴파일이 필요하지 않은 언어를 지원합니다. 코드 편집기는 .zip 파일 아카이브 배포 패키지만을 지원하며 배포 패키지의 크기는 3MB 미만이어야 합니다.

코드 편집기에는 메뉴 모음, 창 및 편집기 창이 포함되어 있습니다.



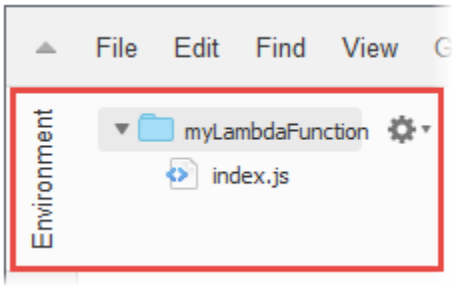
명령이 수행하는 작업 목록은 AWS Cloud9 사용 설명서의 menu bar commands reference를 참조하세요. 이 참조에 나와 있는 일부 명령은 코드 편집기에서 사용할 수 없습니다.

주제

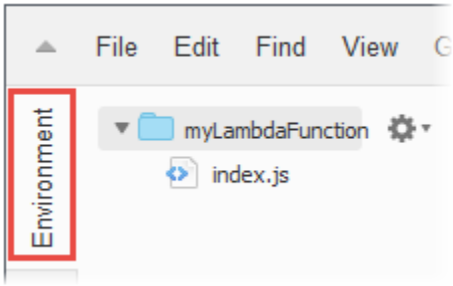
- 파일 및 폴더 작업
- 코드 작업
- 전체 화면 모드에서 작업
- 기본 설정 작업

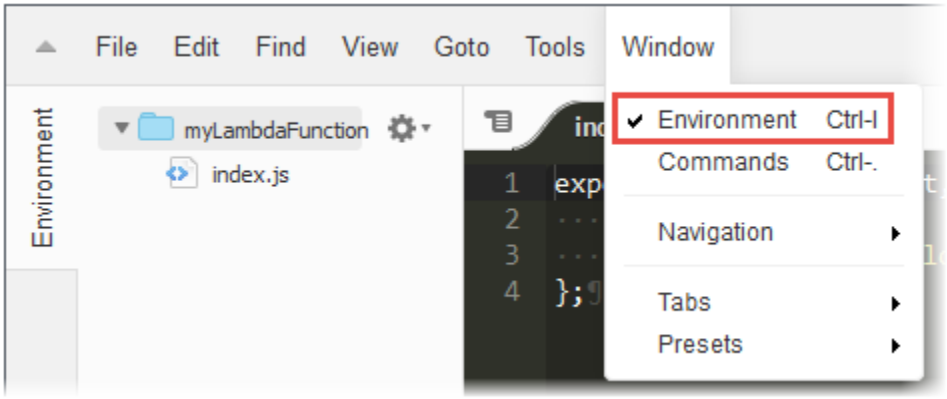
파일 및 폴더 작업

코드 편집기에서 [Environment] 창을 사용하여 함수의 파일을 생성하고, 열고, 관리할 수 있습니다.



[Environment] 창을 표시하거나 숨기려면 [Environment] 버튼을 선택합니다. [Environment] 버튼이 표시되지 않으면 메뉴 모음에서 [Window], [Environment]를 선택합니다.



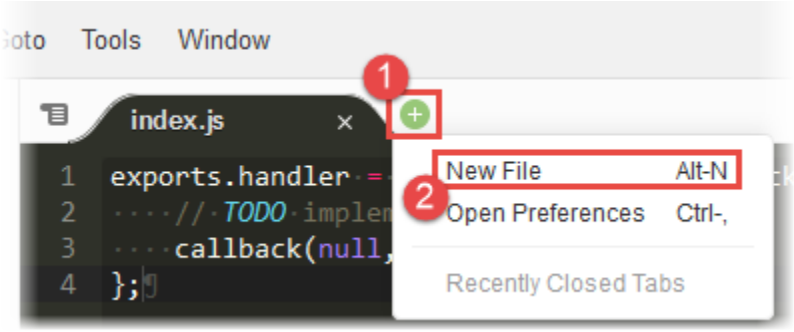


단일 파일을 열고 해당 콘텐츠를 편집기 창에 표시하려면 [Environment] 창에서 해당 파일을 두 번 클릭합니다.

여러 파일을 열고 해당 콘텐츠를 편집기 창에 표시하려면 [Environment] 창에서 해당 파일을 선택합니다. 선택한 파일을 마우스 오른쪽 버튼으로 클릭하고 [Open]을 선택합니다.

새 파일을 생성하려면 다음 중 하나를 수행합니다.

- [Environment] 창에서 새 파일을 추가할 폴더를 마우스 오른쪽 버튼으로 클릭하고 [New File]을 선택합니다. 파일 이름과 확장자를 입력한 다음 Enter 키를 누릅니다.
- 메뉴 모음에서 [File], [New File]을 선택합니다. 파일을 저장할 준비가 되면 메뉴 모음에서 [File], [Save] 또는 [File], [Save As]를 선택합니다. 그런 다음 [Save As] 대화 상자가 표시되면 파일 이름을 지정하고 파일을 저장할 위치를 선택합니다.
- 편집기 창의 탭 버튼 모음에서 [+] 버튼을 선택한 다음 [New File]을 선택합니다. 파일을 저장할 준비가 되면 메뉴 모음에서 [File], [Save] 또는 [File], [Save As]를 선택합니다. 그런 다음 [Save As] 대화 상자가 표시되면 파일 이름을 지정하고 파일을 저장할 위치를 선택합니다.



새 폴더를 생성하려면 [Environment] 창에서 새 폴더를 저장할 폴더를 마우스 오른쪽 버튼으로 클릭하고 [New Folder]를 선택합니다. 폴더 이름을 입력한 다음 Enter 키를 누릅니다.

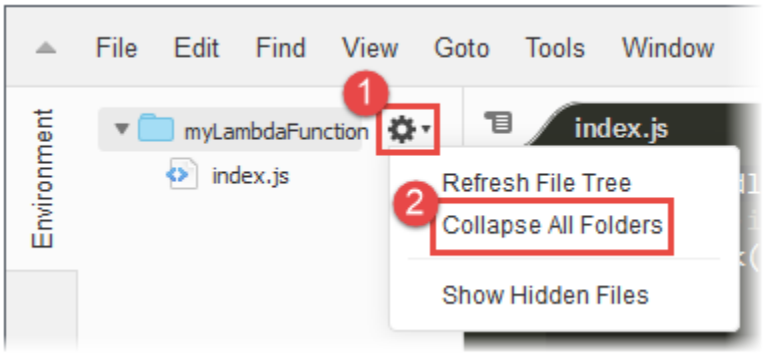
파일을 저장하려면 파일을 열고 해당 콘텐츠를 편집기 창에 표시한 다음 메뉴 모음에서 [File], [Save]를 선택합니다.

파일이나 폴더의 이름을 바꾸려면 [Environment] 창에서 파일이나 폴더를 마우스 오른쪽 버튼으로 클릭합니다. 바꿀 이름을 입력한 다음 Enter 키를 누릅니다.

파일이나 폴더를 삭제하려면 [Environment] 창에서 파일이나 폴더를 선택합니다. 선택한 파일이나 폴더를 마우스 오른쪽 버튼으로 클릭하고 [Delete]를 선택합니다. 그런 다음 [Yes](단일 개체를 선택한 경우) 또는 [Yes to All]을 선택하여 삭제를 확인합니다.

파일이나 폴더를 잘라내기, 복사, 붙여넣기 또는 복제하려면 [Environment] 창에서 파일이나 폴더를 선택합니다. 선택 항목을 마우스 오른쪽 버튼으로 클릭한 다음 각각 잘라내기, 복사, 붙여넣기 또는 복제를 선택합니다.

폴더를 축소하려면 [Environment] 창에서 기어 모양 아이콘을 선택한 다음 [Collapse All Folders]를 선택합니다.



숨김 파일을 표시하거나 숨기려면 [Environment] 창에서 기어 모양 아이콘을 선택한 다음 [Show Hidden Files]를 선택합니다.

함수에 대해 구성된 환경 변수를 보려면 다음을 수행합니다.

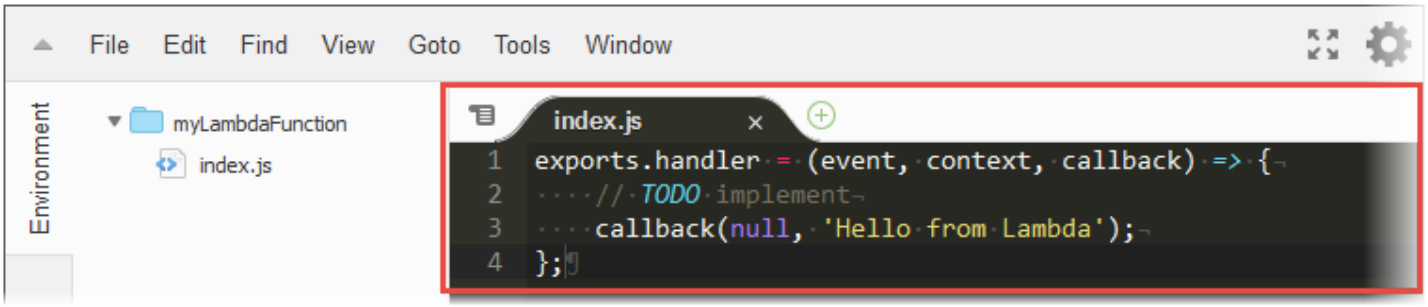
1. 코드 탭을 선택합니다.
2. 환경 변수 탭을 선택합니다.
3. 도구, 환경 변수 표시를 선택합니다.

콘솔 코드 편집기에 나열되어 있는 환경 변수는 암호화된 상태로 유지됩니다. 전송 중 암호화에 대해 암호화 도우미를 활성화한 경우 해당 설정은 변경되지 않습니다. 자세한 내용은 Lambda 환경 변수 보안 단원을 참조하십시오.

환경 변수 목록은 읽기 전용이며 Lambda 콘솔에서만 사용할 수 있습니다. 이 파일은 함수의 .zip 파일 아카이브를 다운로드할 때 포함되지 않으며, 이 파일을 업로드하여 환경 변수를 추가할 수 없습니다.

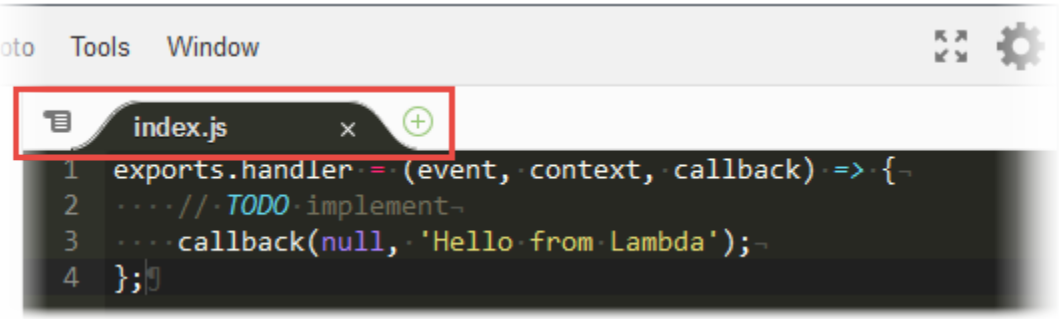
코드 작업

코드 편집기의 편집기 창을 사용하여 코드를 보고 작성할 수 있습니다.



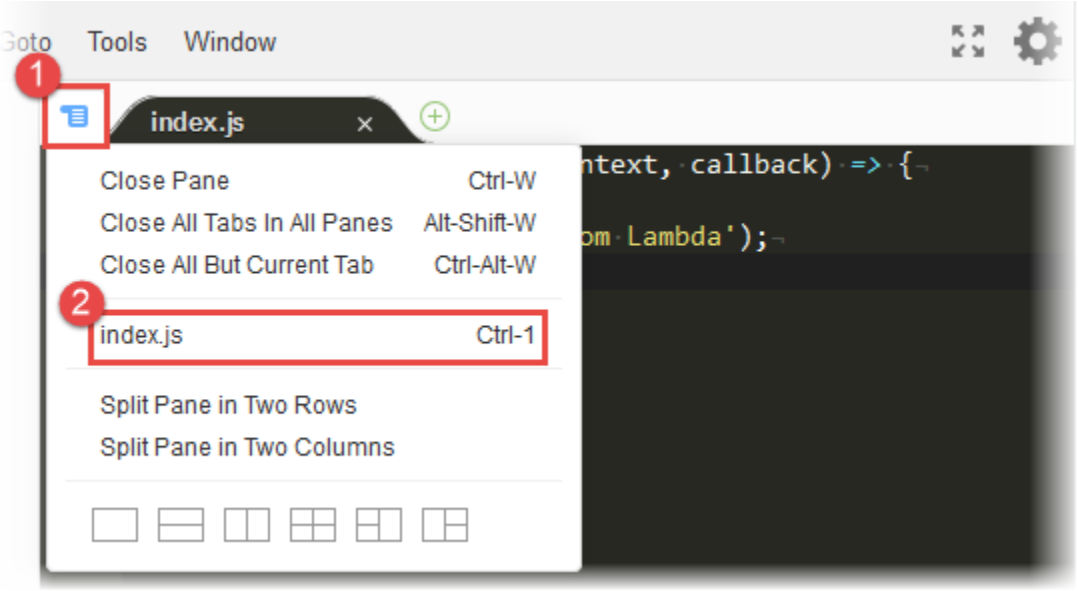
탭 버튼 작업

탭 버튼 모음을 사용하여 파일을 선택하고, 보고, 생성할 수 있습니다.



열려 있는 파일의 콘텐츠를 표시하려면 다음 중 하나를 수행합니다.

- 파일의 탭을 선택합니다.
- 탭 버튼 모음에서 드롭다운 메뉴 버튼을 선택한 다음 파일 이름을 선택합니다.



열려 있는 파일을 닫으려면 다음 중 하나를 수행합니다.

- 파일의 탭에서 [X] 아이콘을 선택합니다.
- 파일의 탭을 선택합니다. 그런 다음 탭 버튼 모음에서 드롭다운 메뉴 버튼을 선택하고 [Close Pane]을 선택합니다.

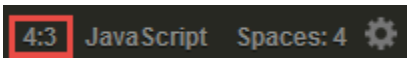
열려 있는 여러 파일을 닫으려면 탭 버튼 모음에서 드롭다운 메뉴를 선택한 다음 필요에 따라 [Close All Tabs in All Panes] 또는 [Close All But Current Tab]을 선택합니다.

새 파일을 생성하려면 탭 버튼 모음에서 [+] 버튼을 선택한 다음 [New File]을 선택합니다. 파일을 저장할 준비가 되면 메뉴 모음에서 [File], [Save] 또는 [File], [Save As]를 선택합니다. 그런 다음 [Save As] 대화 상자가 표시되면 파일 이름을 지정하고 파일을 저장할 위치를 선택합니다.

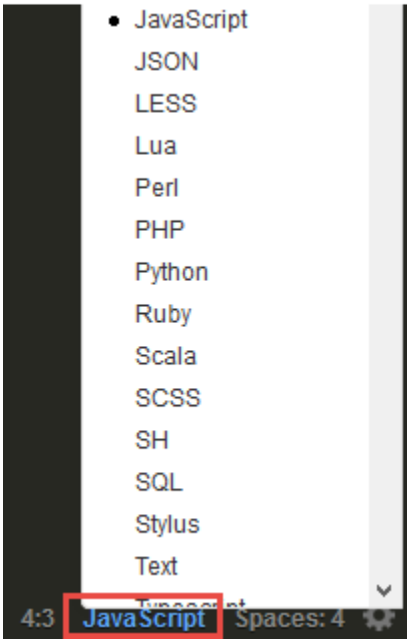
상태 표시줄 작업

상태 표시줄을 사용하여 활성 파일의 특정 행으로 빠르게 이동하고 코드 표시 방법을 변경할 수 있습니다.

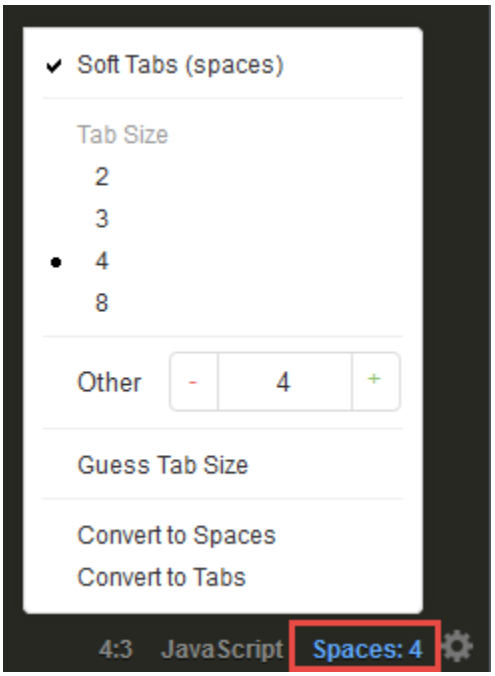
활성 파일의 특정 줄로 빠르게 이동하려면 줄 선택기를 선택하고 이동할 줄 번호를 입력한 다음 Enter 키를 누릅니다.



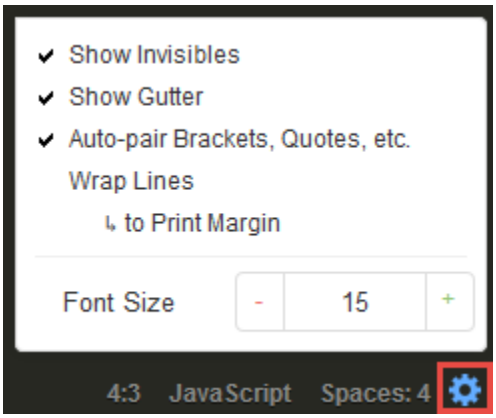
활성 파일의 코드 색상 체계를 변경하려면 코드 색상 체계 선택기를 선택한 다음 새 코드 색상 체계를 선택합니다.



활성 파일에서 소프트 탭을 사용할지 공백을 사용할지에 대한 설정, 탭 크기 설정, 공백이나 탭으로 변환할지에 대한 설정을 변경하려면 공백 및 탭 선택기를 선택한 다음 새 설정을 선택합니다.



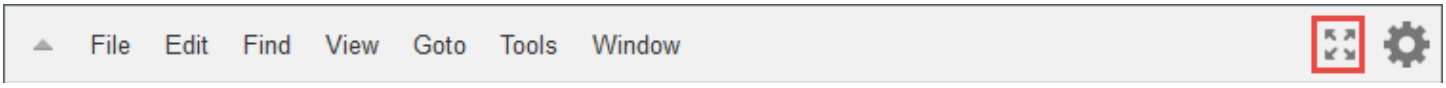
모든 파일에 대해 표시되지 않는 문자나 여백을 표시할지 또는 숨길지에 대한 설정, 괄호 또는 따옴표 쌍 자동 완성 설정, 줄 바꿈 또는 글꼴 크기에 대한 설정을 변경하려면 기어 모양 아이콘을 선택한 다음 새 설정을 선택합니다.



전체 화면 모드에서 작업

코드 편집기를 확장하여 코드 작업 공간을 추가로 확보할 수 있습니다.

코드 편집기를 웹 브라우저 창의 가장자리까지 확장하려면 메뉴 모음에서 [Toggle fullscreen] 버튼을 선택합니다.



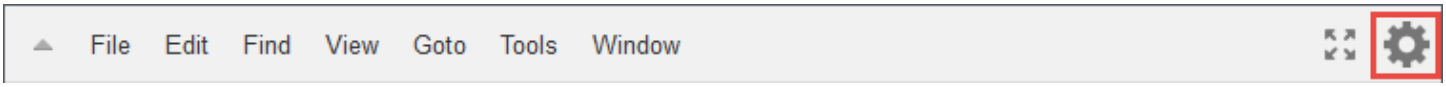
코드 편집기를 원래 크기로 축소하려면 [Toggle fullscreen] 버튼을 다시 선택합니다.

전체 화면 모드에서는 메뉴 모음에 저장과 테스트라는 두 가지 옵션이 추가로 표시됩니다. [Save]를 선택하면 함수 코드가 저장됩니다. [Test] 또는 [Configure Events]를 선택하면 함수의 테스트 이벤트를 생성하거나 편집할 수 있습니다.

기본 설정 작업

표시할 코딩 힌트와 경고, 코드 접기 동작, 코드 자동 완성 동작 등의 다양한 코드 편집기 설정을 변경할 수 있습니다.

코드 편집기 설정을 변경하려면 메뉴 모음에서 [Preferences] 기어 모양 아이콘을 선택합니다.



설정이 수행하는 작업의 목록은 AWS Cloud9 사용 설명서에서 다음 참조를 참조하세요.

- 사용자가 변경할 수 있는 프로젝트 설정
- 사용자가 변경할 수 있는 사용자 설정

이러한 참조에 나와 있는 일부 설정은 코드 편집기에서 사용할 수 없습니다.

추가 Lambda 기능

Lambda는 함수 관리 및 호출을 위한 관리 콘솔 및 API를 제공합니다. 표준 기능 세트를 지원하는 런타임을 제공하므로 필요에 따라 언어와 프레임워크 간에 손쉽게 전환할 수 있습니다. 함수 외에 버전, 별칭, 계층 및 사용자 지정 런타임을 생성할 수도 있습니다.

고급 기능

- 스케일링
- 동시성 제어
- 함수 URL
- 비동기식 호출
- 이벤트 소스 매핑
- 대상
- 함수 블루프린트
- 테스트 및 배포 도구
- 애플리케이션 템플릿

스케일링

Lambda는 코드를 실행하는 인프라를 관리하고 수신 요청에 대한 응답으로 자동 확장됩니다. 함수 인스턴스 하나의 이벤트 처리 속도보다 빠르게 함수를 호출하는 경우 Lambda는 추가 인스턴스를 실행하여 규모를 확장합니다. 트래픽이 감소하면 비활성 인스턴스가 고정되거나 중지됩니다. 함수가 이벤트를 초기화하거나 처리하는 시간에 대해서만 비용을 지불합니다.

자세한 내용은 Lambda 함수 규모 조정 이행 단원을 참조하십시오.

동시성 제어

동시성 설정을 사용하여 프로덕션 애플리케이션이 높은 가용성과 높은 응답성을 유지하도록 보장합니다.

함수가 너무 많은 동시성을 사용하지 못하게 하고 계정의 사용 가능한 동시성 중 일부를 함수에 예약하려면 예약된 동시성을 사용합니다. 예약된 동시성은 사용 가능한 동시성 풀을 하위 집합으로 분할합니다. 예약된 동시성이 있는 함수는 전용 하위 집합에서의 동시성만 사용합니다.

지연 시간 변동 없이 함수를 확장할 수 있도록 하려면 프로비저닝된 동시성을 사용합니다. 초기화하는 데 오랜 시간이 걸리거나 모든 호출에 대해 지연 시간이 매우 짧은 함수의 경우, 프로비저닝된 동시성을 사용하면 함수의 인스턴스를 사전 초기화하고 이들을 항상 실행 상태로 유지할 수 있습니다. Lambda는 Application Auto Scaling을 통합하여 사용률을 바탕으로 프로비저닝된 동시성에 대해 자동 크기 조정을 지원합니다.

자세한 내용은 함수에 대해 예약된 동시성 구성 단원을 참조하십시오.

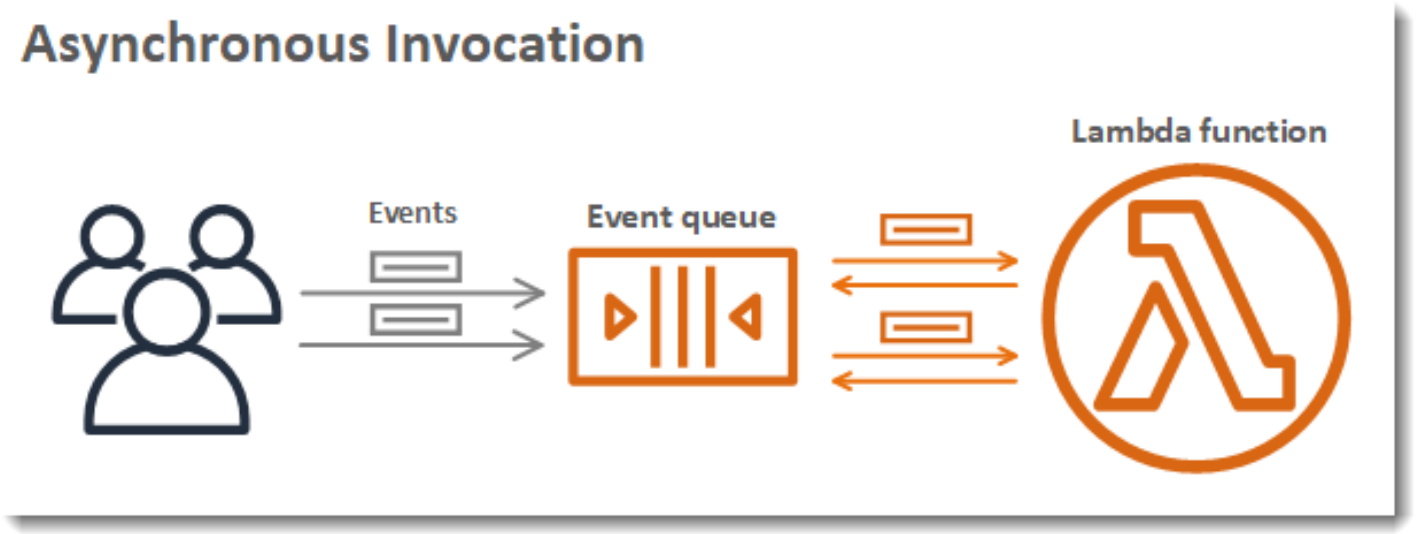
함수 URL

Lambda는 함수 URL을 통해 기본 HTTP(S) 엔드포인트 지원을 제공합니다. 함수 URL을 사용하면 Lambda 함수에 전용 HTTP 엔드포인트를 할당할 수 있습니다. 함수 URL이 구성되면 웹 브라우저, curl, Postman 또는 모든 HTTP 클라이언트를 통해 함수를 호출하는 데 사용할 수 있습니다.

기존 함수에 함수 URL을 추가하거나 함수 URL을 사용하여 새 함수를 생성할 수 있습니다. 자세한 내용은 Lambda 함수 URL 호출 단원을 참조하십시오.

비동기식 호출

함수를 호출할 때 동기식으로 호출할 것인지 비동기식으로 호출할 것인지 선택할 수 있습니다. 동기식 호출의 경우 함수가 이벤트를 처리하여 응답을 반환하기를 기다립니다. 비동기식 호출의 경우, Lambda는 처리를 위해 이벤트를 대기열에 저장하고 즉시 응답을 반환합니다.

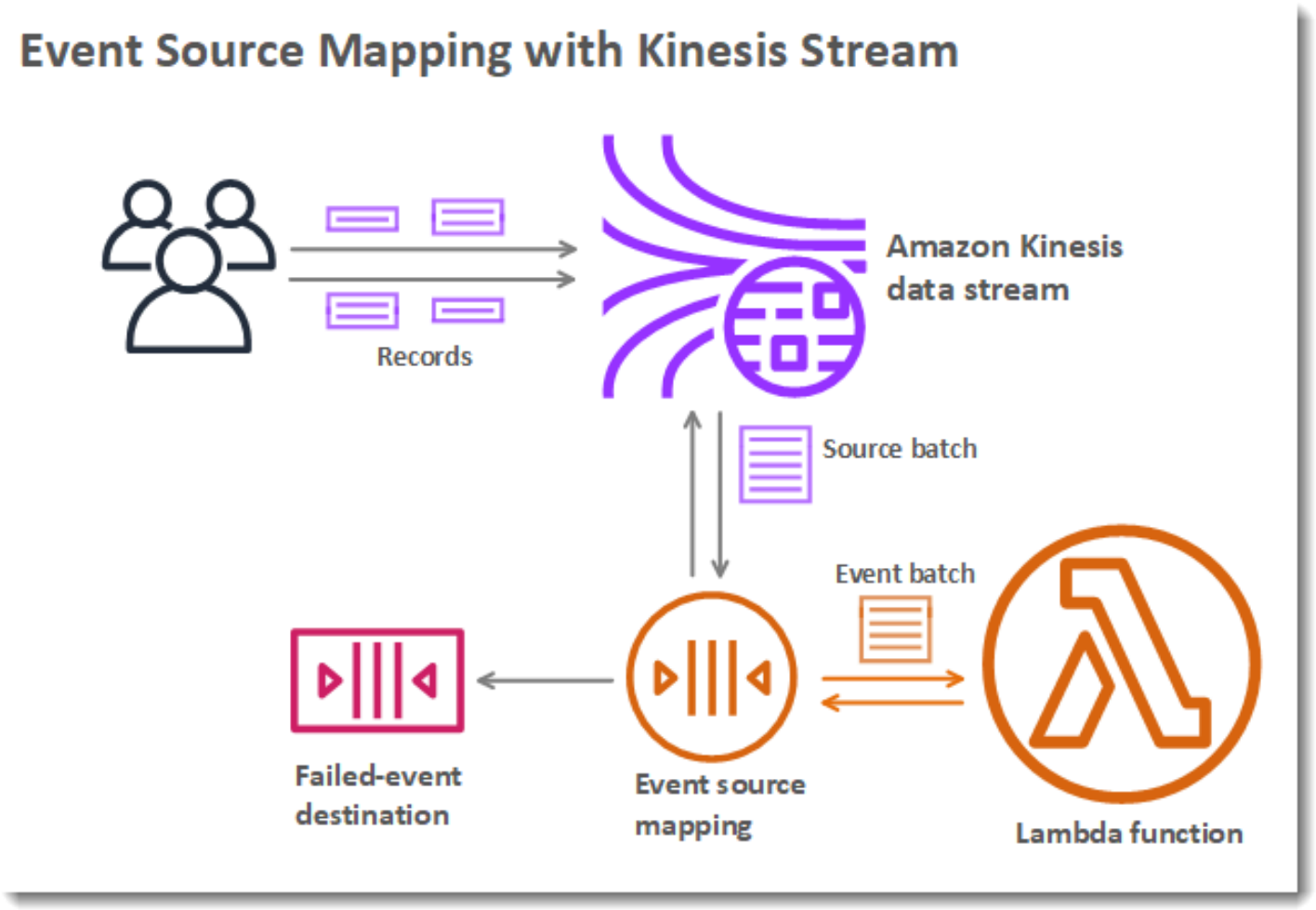


비동기식 호출에서는 함수가 오류를 반환하거나 병목 중인 경우 Lambda가 재시도를 처리합니다. 이 동작을 사용자 지정하기 위해 함수, 버전 또는 별칭에 대한 오류 처리 설정을 구성할 수 있습니다. 처리에 실패한 이벤트를 배달 못한 편지 대기열로 보내거나, 호출 레코드를 대상에 보내도록 Lambda를 구성할 수도 있습니다.

자세한 내용은 비동기식 호출 섹션을 참조하세요.

이벤트 소스 매핑

스트림 또는 대기열에서 항목을 처리하려면 이벤트 소스 매핑을 생성하면 됩니다. 이벤트 소스 매핑은 Amazon Simple Queue Service(Amazon SQS) 대기열, Amazon Kinesis 스트림 또는 Amazon DynamoDB 스트림에서 항목을 읽어 배치로 함수에 전송하는 Lambda 내 리소스입니다. 함수가 처리하는 각 이벤트에는 수백 또는 수천 개의 항목이 포함될 수 있습니다.

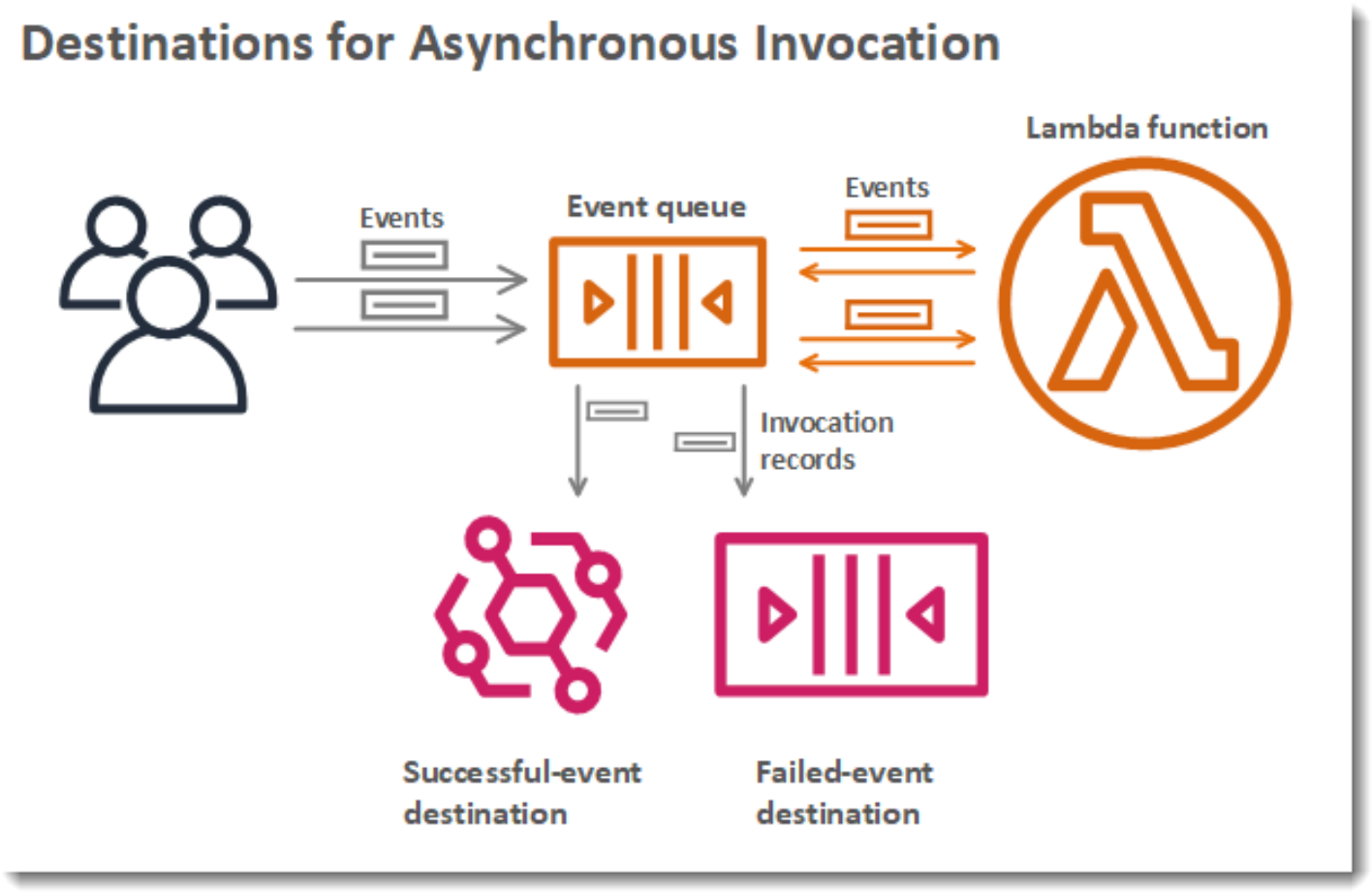


이벤트 소스 매핑은 처리되지 않은 항목의 로컬 대기열을 유지 관리하고, 함수가 오류를 반환하거나 병목 중인 경우 재시도를 처리합니다. 배치 처리 동작 및 오류 처리를 사용자 정의하거나 처리에 실패한 항목의 레코드를 대상으로 보내도록 이벤트 소스 매핑을 구성할 수 있습니다.

자세한 내용은 Lambda가 스트림 및 대기열 기반 이벤트 소스의 레코드를 처리하는 방법 단원을 참조하십시오.

대상

대상은 함수에 대한 호출 레코드를 수신하는 AWS 리소스입니다. 비동기식 호출의 경우, 호출 레코드를 대기열, 주제, 함수 또는 이벤트 버스에 보내도록 Lambda를 구성할 수 있습니다. 성공적인 호출과 처리에 실패한 이벤트에 대해 별도의 대상을 구성할 수 있습니다. 호출 레코드에는 이벤트, 함수의 응답 및 레코드가 전송된 이유에 대한 세부 정보가 포함되어 있습니다.



스트림에서 읽기를 수행하는 이벤트 소스 매핑의 경우, 처리에 실패한 배치 레코드를 대기열이나 주제로 보내도록 Lambda를 구성할 수 있습니다. 이벤트 소스 매핑에 대한 실패 레코드에는 배치에 대한 메타데이터가 포함되어 있으며 스트림의 항목을 가리킵니다.

자세한 내용은 비동기식 호출에 대한 대상 구성 단원과 Amazon DynamoDB에서 AWS Lambda 사용 및 Lambda가 Amazon Kinesis Data Streams의 레코드를 처리하는 방법의 오류 처리 섹션을 참조하세요.

함수 블루프린트

Lambda 콘솔에서 함수를 생성할 때 처음부터 시작하거나, 청사진을 사용하거나, 컨테이너 이미지를 사용하도록 선택할 수 있습니다. 블루프린트는 Lambda를 AWS 서비스 또는 인기 있는 타사 애플리케이션과 함께 사용하는 방법을 보여주는 샘플 코드를 제공합니다. 블루프린트에는 Node.js 및 Python 런타임에 대한 샘플 코드 및 함수 구성 사전 설정이 포함되어 있습니다.

블루프린트는 Amazon 소프트웨어 라이선스에 따라 사용할 수 있도록 제공됩니다. 이들은 Lambda 콘솔에서만 사용할 수 있습니다.

테스트 및 배포 도구

Lambda는 코드를 있는 그대로 또는 컨테이너 이미지로 배포하는 것을 지원합니다. AWS 서비스 및 Docker 명령줄 인터페이스(CLI)와 같은 널리 사용되는 커뮤니티 도구를 사용하여 Lambda 함수를 작성, 빌드 및 배포할 수 있습니다. Docker CLI를 설정하려면 Docker Docs 웹 사이트에서 Get Docker를 참조하세요. AWS에서 Docker를 사용하는 방법에 대한 소개는 Amazon Elastic 컨테이너 레지스트리 사용 설명서에서 AWS CLI를 사용하여 Amazon ECR 시작하기를 참조하세요.

AWS CLI 및 AWS SAM CLI는 Lambda 애플리케이션 스택을 관리하기 위한 명령줄 도구입니다. AWS CloudFormation API를 사용해 애플리케이션 스택을 관리하기 위한 명령 이외에 AWS CLI는 배포 패키지 업로드, 템플릿 업데이트 등의 작업을 간소화하는 상위 수준 명령을 지원합니다. AWS SAM CLI는 템플릿 확인, 로컬 테스트 및 CI/CD 시스템 통합을 포함하여 추가 기능을 제공합니다.

- AWS SAM CLI 설치
- AWS SAM 사용을 통한 서버리스 애플리케이션 테스트 및 디버깅
- AWS SAM 사용을 통한 CI/CD 시스템 사용 서버리스 애플리케이션 배포

애플리케이션 템플릿

Lambda 콘솔을 사용하여 지속적 전달 파이프라인에서 애플리케이션을 생성할 수 있습니다. Lambda 콘솔의 애플리케이션 템플릿에는 하나 이상의 함수에 대한 코드, 함수를 정의하고 AWS 리소스를 지원하는 애플리케이션 템플릿,AWS CodePipeline 파이프라인을 정의하는 인프라 템플릿이 포함되어 있습니다. 파이프라인에는 포함된 Git 리포지토리에 변경 사항을 푸시할 때마다 실행되는 빌드 및 배포 단계가 있습니다.

애플리케이션 템플릿은 MIT No Attribution 라이선스에 따라 제공됩니다. 이들은 Lambda 콘솔에서만 사용할 수 있습니다.

자세한 내용은 AWS Lambda 콘솔에서 애플리케이션 관리 단원을 참조하십시오.

서버리스 솔루션을 빌드하는 방법을 알아보세요.

> **Tip**
>
> 서버리스 솔루션을 빌드하는 방법을 알아보려면 서버리스 개발자 안내서를 확인하세요.

Lambda 런타임

Lambda는 런타임을 사용하여 여러 언어를 지원합니다. 런타임은 Lambda와 함수 간에 호출 이벤트, 컨텍스트 정보 및 응답을 릴레이하는 언어별 환경을 제공합니다. Lambda에서 제공하는 런타임을 사용하거나 나만의 런타임을 빌드할 수 있습니다.

각 주요 프로그래밍 언어 릴리스에는 `nodejs20.x` 또는 `python3.12`와 같은 고유한 런타임 식별자를 가진 별도의 런타임이 있습니다. 새 메이저 언어 버전을 사용하도록 함수를 구성하려면 런타임 식별자를 변경해야 합니다. AWS Lambda는 메이저 버전 간에 이전 버전과의 호환성을 보장할 수 없으므로 이 작업은 고객이 수행해야 합니다.

컨테이너 이미지로 정의된 함수의 경우 컨테이너 이미지를 생성할 때 런타임 및 Linux 배포판을 선택합니다. 런타임을 변경하려면 새 컨테이너 이미지를 생성합니다.

배포 패키지에 .zip 파일 아카이브를 사용하는 경우 함수를 생성할 때 런타임을 선택합니다. 런타임을 변경하려면 함수의 구성을 업데이트할 수 있습니다. 런타임은 Amazon Linux 배포판 중 하나와 페어링됩니다. 기반 실행 환경은 함수 코드에서 액세스할 수 있는 추가 라이브러리와 환경 변수를 제공합니다.

Lambda는 실행 환경에서 함수를 호출합니다. 실행 환경은 함수를 실행하는 데 필요한 리소스를 관리하는 안전하고 격리된 런타임 환경을 제공합니다. Lambda는 사용 가능한 경우 이전 호출에서 실행 환경을 다시 사용하거나 새 실행 환경을 만들 수 있습니다.

Lambda에서 Go 또는 Rust와 같은 다른 언어를 사용하려면 OS 전용 런타임을 사용합니다. Lambda 실행 환경은 호출 이벤트를 받고 응답을 보내기 위한 런타임 인터페이스를 제공합니다. 사용자 지정 런타임을 함수 코드와 함께 배포하거나 하나의 계층에서 구현하여 다른 언어를 배포할 수 있습니다.

지원되는 런타임

다음 테이블에는 지원되는 Lambda 런타임 및 예상되는 사용 중단 날짜가 나와 있습니다. 런타임이 지원 중단된 후에도 제한된 기간에 함수를 생성하고 업데이트할 수 있습니다. 자세한 내용은 the section called “지원 중단 이후 런타임 사용” 단원을 참조하십시오. 이 테이블에서는 현재 예상되는 런타임 지원 중단 날짜를 제공합니다. 이 날짜는 계획 수립을 위해 제공되며 변경될 수 있습니다.

지원되는 런타임

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024년 6월 12일	2025년 2월 28일	2025년 3월 31일
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	2024년 10월 14일	2025년 2월 28일	2025년 3월 31일
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			

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
.NET 8	`dotnet8`	Amazon Linux 2023			
.NET 6	`dotnet6`	Amazon Linux 2	2024년 11월 12일	2025년 2월 28일	2025년 3월 31일
Ruby 3.3	`ruby3.3`	Amazon Linux 2023			
Ruby 3.2	`ruby3.2`	Amazon Linux 2			
OS 전용 런타임	`provided.al2023`	Amazon Linux 2023			
OS 전용 런타임	`provided.al2`	Amazon Linux 2			

> **Note**
>
> 새 리전의 경우, Lambda는 향후 6개월 내에 지원 중단될 런타임은 지원하지 않습니다.

Lambda는 패치를 통해 관리형 런타임 및 해당하는 컨테이너 기본 이미지를 최신 상태로 유지하고 마이너 버전 릴리스를 지원합니다. 자세한 내용은 Lambda 런타임 업데이트를 참조하세요.

Lambda는 Go 1.x 런타임이 지원 중단된 후에도 Go 프로그래밍 언어를 계속 지원합니다. 자세한 내용은 AWS 컴퓨팅 블로그에서 Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2를 참조하세요.

지원되는 모든 Lambda 런타임은 x86_64 아키텍처와 arm64 아키텍처를 모두 지원합니다.

새 런타임 릴리스

Lambda는 릴리스가 언어의 릴리스 주기에서 장기 지원(LTS) 단계에 도달한 경우에만 새 언어 버전의 관리형 런타임을 제공합니다. 예를 들어, Node.js 릴리스 주기의 경우 릴리스가 활성 LTS 단계에 도달하는 경우가 이에 해당합니다.

릴리스가 장기 지원 단계에 도달하기 전까지는 개발 상태이며, 여전히 주요 변경 사항이 적용될 수 있습니다. Lambda는 기본적으로 런타임 업데이트를 자동으로 적용하므로 런타임 버전을 크게 변경하면 함수가 예상대로 작동하지 않을 수 있습니다.

Lambda는 LTS 릴리스가 예정되지 않은 언어 버전에 대해 관리형 런타임을 제공하지 않습니다.

다음 목록에서는 예정된 Lambda 런타임의 목표 출시 월을 보여줍니다. 이 날짜는 참고용일 뿐이기 때문에 변경될 수 있습니다.

- Python 3.13 - 2024년 11월
- Node.js 22 - 2024년 11월

런타임 사용 중단 정책

Lambda 런타임.zip 파일 아카이브에 대한 은 유지 관리 및 보안 업데이트가 적용되는 운영 체제, 프로그래밍 언어 및 소프트웨어 라이브러리의 조합을 기반으로 구축됩니다. Lambda의 표준 지원 중단 정책은 런타임의 주요 구성 요소가 커뮤니티 장기 지원(LTS) 종료에 도달하여 보안 업데이트를 더 이상 사용할 수 없는 경우 런타임을 지원 중단하는 것입니다. 대부분의 경우 이 런타임이 언어 런타임이지만 운영 체제(OS)가 LTS 끝에 도달하여 런타임이 더 이상 사용되지 않는 경우도 있습니다.

런타임이 사용 중단되면 AWS에서 더 이상 해당 런타임에 보안 패치 또는 업데이트를 적용하지 않을 수 있으며 해당 런타임을 사용하는 함수는 더 이상 기술 지원을 받을 수 없습니다. 이러한 사용 중단된 런타임은 어떠한 보증 없이 '있는 그대로' 제공되며 버그, 오류, 결함 또는 기타 취약성을 포함할 수 있습니다.

런타임 업그레이드 및 지원 중단 관리에 대한 자세한 내용은 다음 섹션 및 AWS 컴퓨팅 블로그에서 Managing AWS Lambda runtime upgrades를 참조하세요.

> **Important**
>
> Lambda는 때때로 해당 런타임이 지원하는 언어 버전의 지원 종료일 이후 제한된 기간 동안 Lambda 런타임의 지원 중단을 지연합니다. 이 기간 동안 Lambda는 런타임 OS에만 보안 패치

를 적용합니다. Lambda는 지원 종료일에 도달한 후에는 프로그래밍 언어 런타임에 보안 패치를 적용하지 않습니다.

Node.js 16에 대한 런타임 지원 중단

고객 피드백에 따라 AWS에서는 커뮤니티 LTS 종료 후 9개월로 Node.js 16 런타임의 지원 중단을 지연하고 있습니다. Node.js 16 런타임은 지원되는 런타임 표에 제시된 날짜에 지원이 중단됩니다. 앞서 언급한 바와 같이, LTS가 종료되는 2023년 9월 11일부터 지원 중단일 사이에 Lambda는 OS 패치만 런타임에 적용합니다. 이 기간 동안에는 언어 런타임에 대한 보안 패치가 적용되지 않습니다.

Node.js 16의 지원 중단을 지연하면 이 런타임을 사용하는 고객이 Node.js 18을 건너뛰고 해당 함수를 Node.js 20으로 직접 마이그레이션할 수 있습니다.

공동 책임 모델

Lambda는 지원되는 모든 관리형 런타임과 컨테이너 기본 이미지에 대한 보안 업데이트를 큐레이팅하고 게시하는 역할을 담당합니다. Lambda는 기본적으로 관리형 런타임을 사용하는 함수에 런타임 업데이트를 자동으로 적용합니다. 기본 자동 런타임 업데이트 설정이 변경된 경우 런타임 관리 제어 공동 책임 모델을 참조하세요. 컨테이너 이미지를 사용하여 배포된 함수의 경우 사용자는 최신 기본 이미지에서 함수의 컨테이너 이미지를 다시 빌드하고 컨테이너 이미지를 재배포해야 합니다.

런타임이 지원 중단되면 관리형 런타임 및 컨테이너 기본 이미지 업데이트에 대한 Lambda의 책임이 중단됩니다. 지원되는 런타임 또는 기본 이미지를 사용하도록 함수를 업그레이드할 책임은 사용자에게 있습니다.

모든 경우에 종속성을 포함한 함수 코드에 업데이트를 적용하는 것은 사용자의 책임입니다. 공동 책임 모델에 따른 사용자의 책임은 다음 표에 요약되어 있습니다.

런타임 수명 주기 단계	Lambda의 책임	귀하의 책임
지원되는 관리형 런타임	보안 패치 및 기타 업데이트와 함께 정기적인 런타임 업데이트를 제공합니다. 기본적으로 런타임 업데이트를 자동으로 적용합니다(기본이 아닌 동작은 the section called “런타임 관리 제어” 참조).	종속성을 포함한 함수 코드를 업데이트하여 보안 취약성을 해결합니다.

런타임 수명 주기 단계	Lambda의 책임	귀하의 책임
지원되는 컨테이너 이미지	보안 패치 및 기타 업데이트와 함께 정기적인 컨테이너 기본 이미지 업데이트를 제공합니다.	종속성을 포함한 함수 코드를 업데이트하여 보안 취약성을 해결합니다. 최신 기본 이미지를 사용하여 컨테이너 이미지를 정기적으로 다시 빌드하고 배포합니다.
지원 중단에 가까워진 관리형 런타임	런타임 지원 중단 전에 설명서, AWS Health Dashboard, 이메일, Trusted Advisor 등을 통해 고객에게 알립니다. 런타임 업데이트에 대한 책임은 지원 중단과 동시에 종료됩니다.	Lambda 설명서, AWS Health Dashboard, 이메일 또는 Trusted Advisor에서 런타임 지원 중단 정보를 모니터링합니다. 이전 런타임이 지원 중단되기 전에 지원되는 런타임으로 함수를 업그레이드합니다.
지원 중단에 가까워진 컨테이너 이미지	컨테이너 이미지를 사용하는 함수에 대한 지원 중단 알림은 제공되지 않습니다. 컨테이너 기본 이미지 업데이트에 대한 책임은 지원 중단과 동시에 종료됩니다.	지원 중단 일정을 숙지하고 이전 이미지가 지원 중단되기 전에 지원되는 기본 이미지로 함수를 업그레이드합니다.

지원 중단 이후 런타임 사용

런타임이 사용 중단되면 AWS에서 더 이상 해당 런타임에 보안 패치 또는 업데이트를 적용하지 않을 수 있으며 해당 런타임을 사용하는 함수는 더 이상 기술 지원을 받을 수 없습니다. 이러한 사용 중단된 런타임은 어떠한 보증 없이 '있는 그대로' 제공되며 버그, 오류, 결함 또는 기타 취약성을 포함할 수 있습니다. 지원 중단된 런타임을 사용하는 함수는 성능이 저하되거나 인증서 만료와 같은 기타 문제가 발생하여 제대로 작동하지 않을 수 있습니다.

런타임이 지원 중단된 후 최소 30일 동안은 해당 런타임을 사용하여 새 Lambda 함수를 생성할 수 있습니다. 지원 중단 후 30일이 지나면 Lambda는 새 함수 생성을 차단하기 시작합니다.

런타임이 지원 중단된 후 최소 60일 동안은 기존 함수의 함수 코드 및 구성을 업데이트할 수 있습니다. 지원 중단 후 60일이 지나면 Lambda는 기존 함수의 함수 코드 및 구성 업데이트를 차단하기 시작합니다.

> **Note**
>
> 일부 런타임의 경우 AWS는 지원 중단 후 일반적인 30일 및 60일 이후로 block-function-create 및 block-function-update 날짜를 늦춥니다. AWS는 고객 피드백을 반영하여 함수를 업그레이드할 수 있는 시간을 더 확보하기 위해 이러한 변경을 수행했습니다. 런타임 날짜를 확인하려면 the section called “지원되는 런타임” 및 the section called “더 이상 사용되지 않는 런타임”의 표를 참조하세요.

런타임이 지원 중단된 후 지원되는 최신 런타임을 무기한으로 사용하도록 함수를 업데이트할 수 있습니다. 60일이 지나면 지원 중단된 런타임으로 되돌릴 수 없게 되므로 프로덕션 환경에 런타임 변경 사항을 적용하기 전에 함수가 새 런타임에서 작동하는지 테스트해야 합니다. 롤백을 통해 안전하게 배포하려면 함수 버전 및 별칭을 사용하는 것이 좋습니다.

함수를 계속 생성하고 업데이트할 수 있는 정확한 기간은 고정되어 있지 않습니다. 이 기간은 지원 중단 및 AWS 리전마다 다를 수 있습니다. 함수 생성 및 업데이트를 차단하는 공식 날짜는 이 페이지 첫 번째 섹션의 지원되는 런타임 테이블에 나와 있습니다. Lambda는 이 테이블에 나와 있는 날짜 이전에 함수 생성 또는 업데이트 차단을 시작하지 않습니다.

런타임이 지원 중단된 후에도 무기한으로 함수를 계속 간접 호출할 수 있습니다. 그러나 AWS에서는 함수가 보안 패치를 계속 수신하고 기술 지원을 받을 수 있도록 함수를 지원되는 런타임으로 마이그레이션하는 것을 권장합니다.

런타임 지원 중단 알림 수신

런타임의 지원 중단 날짜가 가까워지면 Lambda는 AWS 계정의 함수가 해당 런타임을 사용하는 경우 이메일 알림을 보냅니다. 알림은 AWS Health Dashboard 및 AWS Trusted Advisor에도 표시됩니다.

- 이메일 알림 수신:

  Lambda는 런타임이 지원 중단되기 최소 180일 전까지 이메일 알림을 보냅니다. 이 이메일에는 런타임을 사용하는 모든 함수의 $LATEST 버전이 나열되어 있습니다. 영향을 받는 함수 버전의 전체 목록을 보려면 Trusted Advisor를 사용하거나 the section called “지원 중단된 런타임을 사용하는 함수 나열” 섹션을 참조하세요.

Lambda는 AWS 계정의 기본 계정 연락처로 이메일 알림을 전송합니다. 계정의 이메일 주소 보기 또는 업데이트에 대한 자세한 내용은 AWS 참조 가이드에서 Updating contact information을 참조하세요.

- AWS Health Dashboard을 통해 알림 수신:

  AWS Health Dashboard에서는 런타임이 지원 중단되기 최소 180일 전까지 알림을 표시합니다. 알림은 계정 상태 페이지의 기타 알림에 표시됩니다. 알림의 영향을 받는 리소스 탭에는 런타임을 사용하는 모든 함수의 $LATEST 버전이 나열됩니다.

  > **Note**
  >
  > 영향을 받는 함수 버전의 전체 최신 목록을 보려면 Trusted Advisor를 사용하거나 the section called “지원 중단된 런타임을 사용하는 함수 나열” 섹션을 참조하세요.

  AWS Health Dashboard 알림은 영향을 받는 런타임이 지원 중단되고 90일 후에 만료됩니다.

- AWS Trusted Advisor 사용하기

  Trusted Advisor에서는 런타임이 지원 중단되기 180일 전까지 알림을 표시합니다. 알림은 보안 페이지에 표시됩니다. 영향을 받는 함수 목록은 지원 중단된 런타임을 사용하는 AWS Lambda 함수 아래에 표시됩니다. 이 함수 목록에는 $LATEST 버전 및 게시된 버전이 모두 표시되며 함수의 현재 상태를 반영하도록 자동으로 업데이트됩니다.

  Trusted Advisor 콘솔의 기본 설정 페이지에서 Trusted Advisor가 제공하는 주간 이메일 알림을 켤 수 있습니다.

지원 중단된 런타임을 사용하는 함수 나열

Trusted Advisor를 사용하여 예정된 런타임 지원 중단의 영향을 받는 함수의 실시간 목록을 보는 것 외에도 AWS Command Line Interface(AWS CLI) 또는 AWS SDK 중 하나를 사용하여 특정 런타임을 사용하는 모든 함수 버전을 나열할 수 있습니다.

AWS CLI를 사용하여 목록을 생성하려면 다음 명령을 실행합니다. `RUNTIME_IDENTIFIER`를 지원 중단되는 런타임 이름으로 바꾸고 AWS 리전을 선택합니다. $LATEST 함수 버전만 나열하려면 명령에서 `--function-version ALL`을 생략합니다.

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

> **Tip**
>
> 명령 예제에서는 us-east-1 리전에서 특정 AWS 계정에 대한 함수를 나열합니다. 각 AWS 계정 및 계정이 함수를 보유한 각 기전에서 이 명령을 반복해야 합니다.

AWS를 사용하여 ListFunctions 작업으로 함수를 나열하는 방법에 대해 자세히 알아보려면 원하는 프로그래밍 언어에 대한 SDK 설명서를 참조하세요. 또한 AWS SDK 중 하나를 사용하여 DescribeLogStreams 및 GetMetricStatistics 작업으로 가장 많이 간접적으로 호출된 함수와 가장 최근에 간접적으로 호출된 함수에 대한 통계를 수집할 수 있습니다.

AWS Config 고급 쿼리 기능을 사용하여 영향을 받는 런타임을 사용하는 모든 함수를 나열할 수도 있습니다. 이 쿼리는 함수의 $LATEST 버전만 반환하지만, 단일 명령으로 모든 리전과 여러 AWS 계정에서 함수를 나열하도록 쿼리를 집계할 수 있습니다. 자세한 내용은 AWS Config 개발자 안내서에서 Querying the Current Configuration State of AWS Auto Scaling Resources를 참조하세요.

더 이상 사용되지 않는 런타임

다음 런타임은 지원 종료 시점에 도달했습니다.

더 이상 사용되지 않는 런타임

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
.NET 7(컨테이너만 해당)	dotnet7	Amazon Linux 2	2024년 5월 14일		
Java 8	java8	Amazon Linux	2024년 1월 8일	2024년 2월 8일	2025년 2월 28일
Go 1.x	go1.x	Amazon Linux	2024년 1월 8일	2024년 2월 8일	2025년 2월 28일
OS 전용 런타임	provided	Amazon Linux	2024년 1월 8일	2024년 2월 8일	2025년 2월 28일

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Ruby 2.7	ruby2.7	Amazon Linux 2	2023년 12월 7일	2024년 1월 9일	2025년 2월 28일
Node.js 14	nodejs14.x	Amazon Linux 2	2023년 12월 4일	2024년 1월 9일	2025년 2월 28일
Python 3.7	python3.7	Amazon Linux	2023년 12월 4일	2024년 1월 9일	2025년 2월 28일
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	2023년 4월 3일	2023년 4월 3일	2023년 5월 3일
Node.js 12	nodejs12.x	Amazon Linux 2	2023년 3월 31일	2023년 3월 31일	2023년 4월 30일
Python 3.6	python3.6	Amazon Linux	2022년 7월 18일	2022년 7월 18일	2022년 8월 29일
.NET 5(컨테이너만 해당)	dotnet5.0	Amazon Linux 2	2022년 5월 10일		
.NET Core 2.1	dotnetcore2.1	Amazon Linux	2022년 1월 5일	2022년 1월 5일	2022년 4월 13일
Node.js 10	nodejs10.x	Amazon Linux 2	2021년 7월 30일	2021년 7월 30일	2022년 2월 14일
Ruby 2.5	ruby2.5	Amazon Linux	2021년 7월 30일	2021년 7월 30일	2022년 3월 31일
Python 2.7	python2.7	Amazon Linux	2021년 7월 15일	2021년 7월 15일	2022년 5월 30일
Node.js 8.10	nodejs8.10	Amazon Linux	2020년 3월 6일		2020년 3월 6일

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Node.js 4.3	`nodejs4.3`	Amazon Linux	2020년 3월 5일		2020년 3월 5일
Node.js 4.3 엣지	`nodejs4.3-edge`	Amazon Linux	2020년 3월 5일		2019년 4월 30일
Node.js 6.10	`nodejs6.10`	Amazon Linux	2019년 8월 12일	2019년 8월 12일	
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	2019년 6월 27일		2019년 7월 30일
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	2019년 5월 30일		2019년 5월 30일
Node.js 0.10	`nodejs`	Amazon Linux			2016년 10월 31일

거의 모든 경우에 언어 버전 또는 운영 체제의 수명 종료는 미리 공개됩니다. 아래 링크는 Lambda가 관리형 런타임으로 지원하는 각 언어의 수명 종료 일정을 제공합니다.

언어 및 프레임워크 지원 정책

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com 및 Corretto FAQ
- Go – golang.org
- .NET – dotnet.microsoft.com

Lambda 런타임 업데이트

Lambda는 보안 업데이트, 버그 수정, 새로운 기능, 성능 개선 사항 및 마이너 버전 릴리스에 대한 지원을 통해 각 관리형 런타임을 최신 상태로 유지합니다. 이러한 런타임 업데이트는 런타임 버전으로 게시됩니다. Lambda는 이전 런타임 버전에서 새 런타임 버전으로 함수를 마이그레이션하여 함수에 런타임 업데이트를 적용합니다.

관리형 런타임을 사용하는 함수의 경우 Lambda는 기본적으로 런타임 업데이트를 자동으로 적용합니다. 자동 런타임 업데이트를 통해, Lambda는 런타임 버전에 패치를 적용하는 데 따른 운영 부담을 덜어줍니다. 대부분의 고객은 자동 업데이트를 선택하는 것이 좋습니다. 자세한 설명은 런타임 관리 제어 섹션을 참조하세요.

또한 Lambda는 각각의 새 런타임 버전을 컨테이너 이미지로 게시합니다. 컨테이너 기반 함수의 런타임 버전을 업데이트하려면 업데이트된 기본 이미지에서 새 컨테이너 이미지를 생성하고 함수를 재배포해야 합니다.

각 런타임 버전은 버전 번호 및 ARN(Amazon 리소스 이름)과 연결됩니다. 런타임 버전 번호는 프로그래밍 언어에서 사용하는 버전 번호에 관계없이 Lambda가 정의하는 번호 지정 스키마를 사용합니다. 런타임 버전 ARN은 각 런타임 버전의 고유 식별자입니다.

함수 로그의 `INIT_START` 줄과 Lambda 콘솔에서 함수의 현재 런타임 버전 ARN을 확인할 수 있습니다.

런타임 버전을 런타임 식별자와 혼동해서는 안 됩니다. 각 런타임에는 `python3.9` 또는 `nodejs18.x`와 같은 고유한 런타임 식별자가 있습니다. 이 식별자는 각각의 메이저 프로그래밍 언어 릴리스에 해당합니다. 런타임 버전은 개별 런타임의 패치 버전을 설명합니다.

> **Note**
>
> 동일한 런타임 버전 번호의 ARN은 AWS 리전 및 CPU 아키텍처마다 다를 수 있습니다.

주제

- 공동 책임 모델
- 규정 준수 요구 사항이 높은 애플리케이션

런타임 관리 제어

Lambda는 기존 함수와 호환되는 런타임 업데이트를 제공하기 위해 노력합니다. 하지만 소프트웨어 패치와 마찬가지로, 드물지만 런타임 업데이트가 기존 함수에 부정적인 영향을 미칠 수 있는 경우도 있습니다. 예를 들어 보안 패치는 이전의 안전하지 않은 동작에 의존하는 기존 함수의 근본적인 문제를 노출시킬 수 있습니다. Lambda 런타임 관리 제어는 드물게 런타임 버전 비호환성이 발생하는 경우 워크로드에 영향을 미칠 위험을 줄이는 데 도움이 됩니다. 각 함수 버전($LATEST 또는 게시된 버전)에 대해 다음 런타임 업데이트 모드 중 하나를 선택할 수 있습니다.

- 자동(기본값) - 2단계 런타임 버전 롤아웃을 사용하여 가장 최신의 안전한 런타임 버전으로 자동 업데이트합니다. 런타임 업데이트의 이점을 항상 누릴 수 있도록 대부분의 고객에게 이 모드를 권장합니다.
- 함수 업데이트 - 함수를 업데이트하면 최신의 안전한 런타임 버전으로 업데이트합니다. 함수를 업데이트하면 Lambda는 함수의 런타임 버전을 최신의 안전한 런타임 버전으로 업데이트합니다. 이 방식은 런타임 업데이트를 함수 배포와 동기화하여 Lambda가 런타임 업데이트를 적용하는 시점을 사용자가 제어할 수 있도록 합니다. 이 모드를 사용하면 드물게 발생하는 런타임 업데이트 비호환성을 조기에 감지하여 해결할 수 있습니다. 이 모드를 사용하는 경우 런타임을 최신 상태로 유지하려면 함수를 정기적으로 업데이트해야 합니다.
- 수동 — 런타임 버전을 수동으로 업데이트합니다. 사용자가 함수 구성에서 런타임 버전을 지정합니다. 이 런타임 버전이 함수에 무기한으로 사용됩니다. 드문 경우지만 새 런타임 버전이 기존 함수와 호환되지 않는 경우 이 모드를 사용하여 함수를 이전 런타임 버전으로 롤백할 수 있습니다. 배포 전반에서 런타임 일관성을 유지하는 데 수동 모드를 사용하지 않는 것이 좋습니다. 자세한 설명은 런타임 버전 롤백 섹션을 참조하세요.

함수에 런타임 업데이트를 적용할 책임의 여부는 선택한 런타임 업데이트 모드에 따라 달라집니다. 자세한 설명은 공동 책임 모델 섹션을 참조하세요.

2단계 런타임 버전 롤아웃

Lambda는 다음 순서로 새 런타임 버전을 도입합니다.

1. 첫 번째 단계에서 Lambda는 사용자가 함수를 생성하거나 업데이트할 때마다 새 런타임 버전을 적용합니다. UpdateFunctionCode또는 UpdateFunctionConfigurationAPI 작업을 호출하면 함수가 업데이트됩니다.

2. 두 번째 단계에서 Lambda는 자동 런타임 업데이트 모드를 사용하고 아직 새 런타임 버전으로 업데이트되지 않은 모든 함수를 업데이트합니다.

롤아웃 프로세스의 전체 기간은 런타임 업데이트에 포함된 보안 패치의 심각도를 비롯한 여러 요인에 따라 달라집니다.

함수를 적극적으로 개발 및 배포하는 경우 첫 번째 단계에서 새 런타임 버전을 선택할 가능성이 높습니다. 이렇게 하면 런타임 업데이트가 함수 업데이트와 동기화됩니다. 드물지만 최신 런타임 버전이 애플리케이션에 부정적인 영향을 미치는 경우, 이 방법을 사용하면 즉각적인 시정 조치를 취할 수 있습니다. 현재 개발 중이 아닌 함수는 여전히 두 번째 단계에서 자동 런타임 업데이트라는 운영상의 이점을 누릴 수 있습니다.

이 방법은 함수 업데이트 모드 또는 수동 모드로 설정된 함수에는 영향을 미치지 않습니다. 함수 업데이트 모드를 사용하는 함수의 경우 생성하거나 업데이트할 때만 최신 런타임 업데이트가 적용됩니다. 수동 모드를 사용하는 함수의 경우 런타임 업데이트가 적용되지 않습니다.

Lambda는 여러 AWS 리전에 새로운 런타임 버전을 점진적, 순차적으로 게시합니다. 함수가 자동 또는 함수 업데이트 모드로 설정된 경우 여러 리전에 동시에 배포되거나 동일한 리전에서 서로 다른 시간에 배포된 함수에 서로 다른 런타임 버전이 적용될 수 있습니다. 여러 환경에서 런타임 버전 일관성을 보장해야 하는 고객은 컨테이너 이미지를 사용하여 Lambda 함수를 배포해야 합니다. 수동 모드는 드물게 런타임 버전이 함수와 호환이 안 되는 경우에 런타임 롤백을 지원하기 위한 임시 문제 해결 조치로 설계되었습니다.

런타임 버전 롤백

드물지만 새 런타임 버전이 기존 함수와 호환되지 않는 경우 해당 런타임 버전을 이전 버전으로 롤백할 수 있습니다. 이렇게 하면 애플리케이션이 계속 작동하고 중단을 최소화하여 최신 런타임 버전으로 돌아가기 전에 비호환성을 해결할 시간을 확보할 수 있습니다.

Lambda는 특정 런타임 버전을 사용할 수 있는 기간에 대한 시간 제한을 두지 않습니다. 하지만 최신 보안 패치, 성능 개선 사항 및 기능을 활용하려면 가능한 한 빨리 최신 런타임 버전으로 업데이트하는 것이 좋습니다. Lambda는 드물지만 런타임 업데이트 호환성 문제가 발생하는 경우를 대비하여 일시적인 문제 해결 수단으로만 이전 런타임 버전으로 롤백하는 옵션을 제공합니다. 이전 런타임 버전을 장기간 사용하는 함수는 결국 성능이 저하되거나 인증서 만료와 같은 문제가 발생하여 제대로 작동하지 않을 수 있습니다.

런타임 버전을 롤백하는 방법은 다음과 같습니다.

- 수동 런타임 업데이트 모드 사용

- 게시된 함수 버전 사용

자세한 내용은 AWS 컴퓨팅 블로그에서 Introducing AWS Lambda runtime management controls(런타임 관리 제어 소개)를 참조하세요.

수동 런타임 업데이트 모드를 사용하여 런타임 버전 롤백

자동 런타임 버전 업데이트 모드를 사용하거나 $LATEST 런타임 버전을 사용하는 경우 수동 모드를 사용하여 런타임 버전을 롤백할 수 있습니다. 롤백하려는 함수 버전에 대해 런타임 버전 업데이트 모드를 수동으로 변경하고 이전 런타임 버전의 ARN을 지정합니다. 이전 런타임 버전의 ARN 찾는 방법에 대한 자세한 내용은 런타임 버전 변경 확인 섹션을 참조하세요.

> **Note**
>
> 함수의 $LATEST 버전이 수동 모드를 사용하도록 구성된 경우 함수가 사용하는 CPU 아키텍처 또는 런타임 버전을 변경할 수 없습니다. 이러한 변경 작업을 수행하려면 자동 또는 함수 업데이트 모드로 변경해야 합니다.

게시된 함수 버전을 사용하여 런타임 버전 롤백

게시된 함수 버전은 생성한 시점의 $LATEST 함수 코드 및 구성을 보여주는 변경 불가능한 스냅샷입니다. 자동 모드에서는 Lambda가 런타임 버전 롤아웃의 2단계에서 게시된 함수 버전의 런타임 버전을 자동으로 업데이트합니다. 함수 업데이트 모드에서는 Lambda가 게시된 함수 버전의 런타임 버전을 업데이트하지 않습니다.

따라서 함수 업데이트 모드를 사용하여 게시된 함수 버전은 함수 코드, 구성 및 런타임 버전의 정적 스냅샷을 생성합니다. 함수 버전과 함께 함수 업데이트 모드를 사용하면 런타임 업데이트를 배포와 동기화할 수 있습니다. 트래픽을 이전에 게시된 함수 버전으로 리디렉션하여 코드, 구성 및 런타임 버전의 롤백을 조정할 수도 있습니다. 이 접근 방식을 지속적 통합 및 지속적 전달(CI/CD)에 통합하여 런타임 업데이트가 호환되지 않는 드문 경우에 완전 자동 롤백을 실행할 수 있습니다. 이 방법을 사용할 때는 정기적으로 함수를 업데이트하고 새 함수 버전을 게시하여 최신 런타임 업데이트를 적용해야 합니다. 자세한 설명은 공동 책임 모델 섹션을 참조하세요.

런타임 버전 변경 확인

런타임 버전 번호와 ARN은 INIT_START 로그 라인에 기록되며, Lambda는 새 실행 환경을 생성할 때마다 Lambda가 CloudWatch Logs로 내보냅니다. 실행 환경은 모든 함수 호출에 대해 동일한 런타임 버전을 사용하므로 Lambda는 init 단계를 실행할 때만 INIT_START 로그 라인을 내보냅니다. Lambda

는 각 함수 호출에 대해 이 로그 줄을 내보내지 않습니다. Lambda는 로그 라인을 CloudWatch Logs로 내보내지만 콘솔에는 표시되지 않습니다.

Example INIT_START 로그 줄의 예

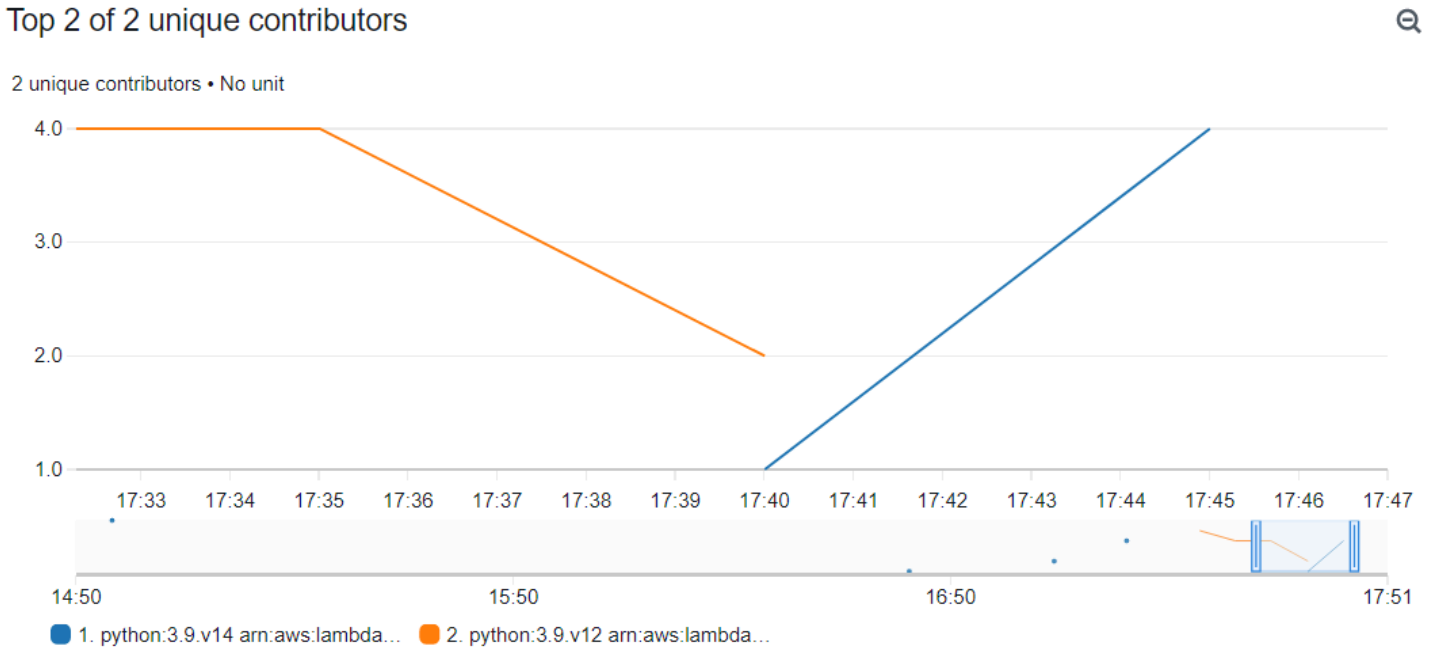
```
INIT_START Runtime Version: python:3.9.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

로그를 직접 처리하는 대신 Amazon CloudWatch Contributor Insights를 사용하여 런타임 버전 간 전환을 식별할 수 있습니다. 다음 규칙은 각 `INIT_START` 로그 줄에서 개별 런타임 버전의 수를 계산합니다. 이 규칙을 사용하려면 예제 로그 그룹 이름 `/aws/lambda/*`를 함수 또는 함수 그룹에 적합한 접두사로 바꿉니다.

```
{
  "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"
  }
```

```
}
```

다음 CloudWatch Contributor Insights 보고서는 이전 규칙에 의해 캡처된 런타임 버전 전환의 예를 보여줍니다. 주황색 선은 이전 런타임 버전(python:3.9.v12)의 실행 환경 초기화를 나타내며, 파란색 선은 새 런타임 버전(python:3.9.v14)의 실행 환경 초기화를 나타냅니다.



런타임 관리 설정 구성

Lambda 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 런타임 관리 설정을 구성할 수 있습니다.

> **Note**
>
> 각 함수 버전마다 런타임 관리 설정을 별도로 구성할 수 있습니다.

Lambda가 런타임 버전을 업데이트하는 방법을 구성하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. Code(코드) 탭의 Runtime settings(런타임 설정)에서 Edit runtime management configuration(런타임 관리 구성 편집)을 선택합니다.
4. Runtime management configuration(런타임 관리 구성)에서 다음 중 하나를 선택합니다.

- 함수가 자동으로 최신 런타임 버전으로 업데이트되도록 하려면 Auto(자동)를 선택합니다.
- 사용자가 함수를 변경할 때 함수가 최신 런타임 버전으로 업데이트되도록 하려면 Function update(함수 업데이트)를 선택합니다.
- 런타임 버전 ARN을 변경할 때만 함수가 최신 런타임 버전으로 업데이트되도록 하려면 Manual(수동)을 선택합니다.

> **Note**
>
> 런타임 버전 ARN은 Runtime management configuration(런타임 관리 구성)에서 확인할 수 있습니다. 함수 로그의 `INIT_START` 줄에서도 ARN을 확인할 수 있습니다.

5. 저장을 선택합니다.

Lambda가 런타임 버전을 업데이트하는 방법을 구성하려면(AWS CLI)

함수에 대한 런타임 관리를 구성하려면 런타임 업데이트 모드와 함께 `put-runtime-management-config` AWS CLI 명령을 사용하면 됩니다. `Manual` 모드를 사용할 때는 런타임 버전 ARN도 제공해야 합니다.

```
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
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

공동 책임 모델

Lambda는 지원되는 모든 관리형 런타임과 컨테이너 이미지에 대한 보안 업데이트를 큐레이팅하고 게시하는 역할을 담당합니다. 최신 런타임 버전을 사용하도록 기존 함수를 업데이트할 책임의 여부는 사용하는 런타임 업데이트 모드에 따라 달라집니다.

Lambda는 자동 런타임 업데이트 모드를 사용하도록 구성된 모든 함수에 런타임 업데이트를 적용하는 역할을 담당합니다.

함수 업데이트 런타임 업데이트 모드로 구성된 함수의 경우 사용자가 함수를 정기적으로 업데이트해야 합니다. Lambda는 사용자가 해당 업데이트를 수행할 때 런타임 업데이트를 적용하는 역할을 담당합니다. 사용자가 함수를 업데이트하지 않으면 Lambda는 런타임을 업데이트하지 않습니다. 사용자가 함수를 정기적으로 업데이트하지 않을 경우 보안 업데이트를 계속 받을 수 있도록 자동 런타임 업데이트로 구성하는 것이 좋습니다.

수동 런타임 업데이트 모드를 사용하도록 구성된 함수의 경우, 최신 런타임 버전을 사용하도록 사용자가 함수를 업데이트해야 합니다. 런타임 업데이트가 호환되지 않는 드문 경우를 대비하여 런타임 버전을 롤백하는 용도로만 이 모드를 사용하는 것이 좋습니다. 또한 함수가 패치되지 않는 시간을 최소화하려면 최대한 빨리 자동 모드로 변경하는 것이 좋습니다.

컨테이너 이미지를 사용하여 함수를 배포하는 경우 Lambda는 업데이트된 기본 이미지를 게시하는 역할을 담당합니다. 이 경우 사용자는 최신 기본 이미지에서 함수의 컨테이너 이미지를 다시 빌드하고 컨테이너 이미지를 재배포해야 합니다.

이 내용은 다음 표에 요약되어 있습니다.

Deployment mode(배포 모드)	Lambda의 책임	고객의 책임
관리형 런타임, 자동 모드	최신 패치가 포함된 새 런타임 버전을 게시합니다. 기존 함수에 런타임 패치를 적용합니다.	드물게 런타임 업데이트 호환성 문제가 발생하는 경우 이전 런타임 버전으로 롤백합니다.
관리형 런타임, 함수 업데이트 모드	최신 패치가 포함된 새 런타임 버전을 게시합니다.	함수를 정기적으로 업데이트하여 최신 런타임 버전을 적용합니다. 함수를 정기적으로 업데이트하지 않을 경우 함수를 자동 모드로 전환합니다. 드물게 런타임 업데이트 호환성 문제가 발생하는 경우 이전 런타임 버전으로 롤백합니다.

Deployment mode(배포 모드)	Lambda의 책임	고객의 책임
관리형 런타임, 수동 모드	최신 패치가 포함된 새 런타임 버전을 게시합니다.	런타임 업데이트 호환성 문제가 발생하는 드문 경우에 임시 런타임 롤백을 수행하는 데에만 이 모드를 사용합니다. 함수를 자동 또는 함수 업데이트 모드로 전환하고 최대한 빨리 최신 런타임 버전으로 전환하합니다.
컨테이너 이미지	최신 패치가 포함된 새 컨테이너 이미지를 게시합니다.	최신 컨테이너 기본 이미지를 사용하여 정기적으로 함수를 재배포함으로써 최신 패치를 적용합니다.

AWS의 공동 책임에 대한 자세한 내용은 AWS 클라우드 보안 사이트에서 공동 책임 모델을 참조하세요.

규정 준수 요구 사항이 높은 애플리케이션

Lambda 고객은 패치 요구 사항을 충족하기 위해 일반적으로 자동 런타임 업데이트를 사용합니다. 애플리케이션에 엄격한 패치 업데이트 요구 사항이 적용되는 경우 이전 런타임 버전의 사용을 제한해야 할 수 있습니다. AWS Identity and Access Management(IAM) 을 사용하여 AWS 계정 내 사용자의 API 작업 액세스를 거부함으로써 Lambda의 런타임 관리 제어를 제한할 수 있습니다. PutRuntimeManagementConfig 이 작업은 함수의 런타임 업데이트 모드를 선택하는 데 사용됩니다. 이 작업에 대한 액세스를 거부하면 모든 함수가 기본적으로 자동 모드로 설정됩니다. 서비스 제어 정책(SCP)을 사용하여 조직 전체에 이 제한을 적용할 수 있습니다. 함수를 이전 런타임 버전으로 롤백해야 하는 경우, 정책 예외를 기준으로 허용할 수 있습니다. case-by-case

런타임 환경 수정

내부 익스텐션을 사용하여 런타임 프로세스를 수정할 수 있습니다. 내부 익스텐션은 별도의 프로세스가 아니라 런타임 프로세스의 일부로 실행됩니다.

Lambda는 런타임에 옵션과 도구를 추가하도록 설정할 수 있는 언어별 환경 변수를 제공합니다. Lambda는 또한 Lambda가 런타임 시작 동작을 스크립트에 위임할 수 있도록 래퍼 스크립트를 제공합니다. 래퍼 스크립트를 생성하여 런타임 시작 동작을 사용자 지정할 수 있습니다.

언어별 환경 변수

Lambda는 다음 언어별 환경 변수를 통해 함수 초기화 중에 코드를 미리 로드할 수 있는 구성 전용 방법을 지원합니다.

- `JAVA_TOOL_OPTIONS` – Java에서 Lambda는 Lambda에서 추가 명령줄 변수를 설정할 수 있도록 이 환경 변수를 지원합니다. 이 환경 변수를 사용하면 `agentlib` 또는 `javaagent` 옵션을 사용하여 네이티브 또는 Java 프로그래밍 언어 에이전트를 시작하는 것을 포함하여 도구의 초기화를 지정할 수 있습니다. 자세한 내용은 `JAVA_TOOL_OPTIONS` 환경 변수를 참조하세요.
- `NODE_OPTIONS` - Node.js 런타임에서 사용할 수 있습니다.
- `DOTNET_STARTUP_HOOKS` – .NET Core 3.1 이상에서 이 환경 변수는 Lambda가 사용할 수 있는 어셈블리(dll)에 대한 경로를 지정합니다.

언어별 환경 변수를 사용하는 것이 시작 속성을 설정하는 권장 방법입니다.

래퍼 스크립트

래퍼 스크립트를 생성하여 Lambda 함수의 런타임 시작 동작을 사용자 지정할 수 있습니다. 래퍼 스크립트를 사용하면 언어별 환경 변수를 통해 설정할 수 없는 구성 파라미터를 설정할 수 있습니다.

> **Note**
>
> 래퍼 스크립트가 런타임 프로세스를 성공적으로 시작하지 않으면 호출이 실패할 수 있습니다.

래퍼 스크립트는 모든 네이티브 Lambda 런타임에서 지원됩니다. 래퍼 스크립트는 OS 전용 런타임(`provided` 런타임 제품군)에서 지원되지 않습니다.

함수에 래퍼 스크립트를 사용하면 Lambda Lambda는 스크립트를 사용하여 런타임을 시작합니다. Lambda는 표준 런타임 시작을 위해 인터프리터 경로와 모든 원본 인수를 스크립트에 보냅니다. 스크

립트는 프로그램의 시작 동작을 확장하거나 변형할 수 있습니다. 예를 들어 스크립트는 인수를 삽입 및 변경하거나, 환경 변수를 설정하거나, 지표, 오류 및 기타 진단 정보를 캡처할 수 있습니다.

스크립트를 지정하려면 AWS_LAMBDA_EXEC_WRAPPER 환경 변수의 값을 실행 바이너리 또는 스크립트의 파일 시스템 경로로 설정합니다.

예제: Python 3.8에서 래퍼 스크립트 생성 및 사용

다음 예제에서는 `-X importtime` 옵션을 사용하여 Python 인터프리터를 시작하는 래퍼 스크립트를 생성합니다. 함수를 실행하면 Lambda가 각 가져오기에 소요된 기간을 표시하는 로그 항목을 생성합니다.

Python 3.8에서 래퍼 스크립트를 생성하고 사용하려면

1. 래퍼 스크립트를 생성하려면 다음 코드를 `importtime_wrapper` 파일에 붙여 넣습니다.

```
#!/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. 스크립트에 실행 권한을 부여하려면 명령줄에서 `chmod +x importtime_wrapper`를 입력합니다.
3. 스크립트를 Lambda 계층으로 배포합니다.
4. Lambda 콘솔을 사용하여 함수를 생성합니다.
   a. Lambda 콘솔을 엽니다.
   b. 함수 생성을 선택합니다.
   c. 기본 정보(Basic information)에서 함수 이름(Function name)에 **`wrapper-test-function`**을 입력합니다.

d. 런타임(Runtime)에서 Python 3.8을 선택합니다.

e. 함수 생성(Create function)을 선택합니다.

5. 함수에 계층을 추가합니다.

a. 기능을 선택한 다음 코드(Code)를 선택합니다(아직 선택되어 있지 않은 경우).

b. [Add a layer]를 선택합니다.

c. 계층 선택(Choose a layer)에서 앞서 생성한 호환되는 계층의 이름(Name) 및 버전(Version)을 선택합니다.

d. 추가를 선택합니다.

6. 코드와 환경 변수를 함수에 추가합니다.

a. 함수 코드 편집기(code editor)에서 다음 함수 코드를 붙여 넣습니다.

```
import json

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

b. 저장을 선택합니다.

c. Environment variables(환경 변수)에서 편집을 선택합니다.

d. Add environment variable(환경 변수 추가)을 선택합니다.

e. 키(Key)에 `AWS_LAMBDA_EXEC_WRAPPER`를 입력합니다.

f. 값에 `/opt/importtime_wrapper`을(를) 입력합니다.

g. 저장을 선택합니다.

7. 함수를 실행하려면 테스트(Test)를 선택합니다.

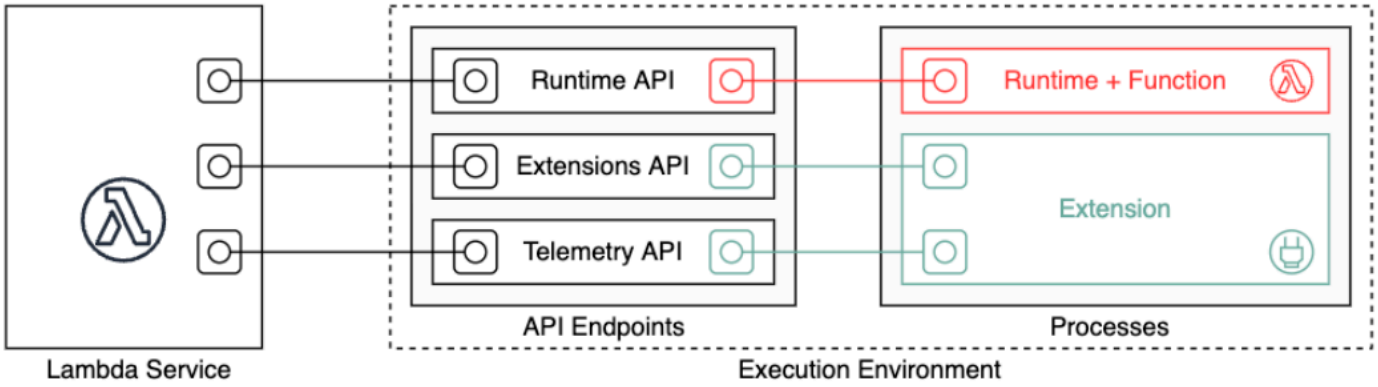
래퍼 스크립트가 `-X importtime` 옵션을 사용하여 Python 인터프리터를 시작했기 때문에 로그에는 각 가져오기에 필요한 시간이 표시됩니다. 예:

```
...
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 런타임 API

AWS Lambda은 사용자 지정 런타임에 HTTP API를 제공하여 Lambda에서 호출 이벤트를 수신하고 Lambda 실행 환경 내에서 응답 데이터를 다시 전송합니다.



런타임 API 버전 2018-06-01에 대한 OpenAPI 사양은 runtime-api.zip에서 사용할 수 있습니다.

API 요청 URL을 만들려면 런타임에서 `AWS_LAMBDA_RUNTIME_API` 환경 변수에서 API 엔드포인트를 가져오고 API 버전을 추가한 다음 원하는 리소스 경로를 추가합니다.

Example 요청

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

API 메서드

- 다음 호출
- 호출 응답
- 초기화 오류
- 호출 오류

다음 호출

경로 – `/runtime/invocation/next`

메서드 – GET

런타임에서는 호출 이벤트를 요청하기 위해 이 메시지를 Lambda로 보냅니다. 응답 본문에는 호출의 페이로드가 포함되어 있습니다. 이 페이로드는 함수 트리거의 이벤트 데이터를 포함하는 JSON 문서입니다. 응답 헤더에는 호출에 대한 추가 데이터가 포함되어 있습니다.

응답 헤더

- `Lambda-Runtime-Aws-Request-Id` – 함수 호출을 트리거한 요청을 식별하는 요청 ID입니다.

  예: `8476a536-e9f4-11e8-9739-2dfe598c3fcd`.
- `Lambda-Runtime-Deadline-Ms` – 함수가 Unix 시간 형식에 따른 밀리초 단위의 시간 제한에 도달하는 날짜입니다.

  예: `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – 호출에 지정된 Lambda 함수, 버전 또는 별칭의 ARN입니다.

  예: `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.
- `Lambda-Runtime-Trace-Id` – AWS X-Ray 추적 헤더입니다.

  예: `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.
- `Lambda-Runtime-Client-Context` – AWS Mobile SDK 호출에서 클라이언트 애플리케이션 및 디바이스에 관한 데이터입니다.
- `Lambda-Runtime-Cognito-Identity` – AWS Mobile SDK 호출에서 Amazon Cognito 자격 증명 공급자에 관한 데이터입니다.

응답이 지연될 수 있으므로 `GET` 요청에 시간 제한을 설정하지 마세요. Lambda가 런타임을 부트스트랩하는 시점과 런타임에 반환할 이벤트가 있는 시점 사이에 런타임 프로세스가 몇 초 동안 동결될 수 있습니다.

요청 ID는 Lambda 내에서 호출을 추적합니다. 응답을 전송할 때에는 이 ID를 사용하여 호출을 지정합니다.

트레이스 헤더에는 추적 ID, 상위 ID 및 샘플링 결정이 포함되어 있습니다. 요청이 샘플링될 경우, 이 요청은 Lambda 또는 업스트림 서비스에 의해 샘플링된 것입니다. 런타임은 헤더의 값을 사용해 `_X_AMZN_TRACE_ID`을 설정해야 합니다. X-Ray SDK는 이 값을 판독하여 ID를 가져온 다음, 요청을 추적할지 여부를 결정합니다.

호출 응답

경로 – `/runtime/invocation/AwsRequestId/response`

메서드 – POST

함수가 완료될 때까지 실행되면 런타임은 호출 응답을 Lambda로 보냅니다. 동기식 호출의 경우, Lambda는 응답을 클라이언트로 보냅니다.

Example 성공 요청

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

초기화 오류

함수가 오류를 반환하거나 런타임에서 초기화 중에 오류가 발생하면 런타임에서는 이 메서드를 사용하여 Lambda에 오류를 보고합니다.

경로 – `/runtime/init/error`

메서드 – POST

헤더

`Lambda-Runtime-Function-Error-Type` – 런타임에서 발생한 오류 유형입니다. 필수 항목 여부: 아니요

이 헤더는 문자열 값으로 구성됩니다. Lambda는 모든 문자열을 허용하지만 <category.reason> 형식을 사용하는 것이 좋습니다. 예:

- 런타임. NoSuchHandler
- 런타임.api KeyNotFound
- 런타임. ConfigInvalid
- 런타임. UnknownReason

본문 파라미터

ErrorRequest – 오류에 대한 정보입니다. 필수 항목 여부: 아니요

이 필드는 다음과 같은 구조의 JSON 객체입니다.

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

Lambda는 errorType에 대한 모든 값을 허용합니다.

다음 예제에서는 Lambda 함수가 호출에 제공된 이벤트 데이터를 구문 분석할 수 없는 함수 오류 메시지를 보여 줍니다.

Example 함수 오류

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

응답 본문 파라미터

- StatusResponse – 문자열. 상태 정보, 202 응답 코드와 함께 전송됨.
- ErrorResponse – 오류 응답 코드와 함께 전송되는 추가 오류 정보입니다. ErrorResponse 오류 유형과 오류 메시지가 들어 있습니다.

응답 코드

- 202 - 수락됨
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 런타임을 신속히 종료해야 합니다.

Example 초기화 오류 요청

```
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"
```

호출 오류

함수가 오류를 반환하거나 런타임에 오류가 발생하면 런타임에서는 이 메서드를 사용하여 오류를 Lambda에 보고합니다.

경로 – `/runtime/invocation/AwsRequestId/error`

메서드 – POST

헤더

`Lambda-Runtime-Function-Error-Type` – 런타임에서 발생한 오류 유형입니다. 필수 항목 여부: 아니요

이 헤더는 문자열 값으로 구성됩니다. Lambda는 모든 문자열을 허용하지만 <category.reason> 형식을 사용하는 것이 좋습니다. 예:

- 런타임. NoSuchHandler
- 런타임.api KeyNotFound
- 런타임. ConfigInvalid
- 런타임. UnknownReason

본문 파라미터

`ErrorRequest` – 오류에 대한 정보입니다. 필수 항목 여부: 아니요

이 필드는 다음과 같은 구조의 JSON 객체입니다.

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

Lambda는 `errorType`에 대한 모든 값을 허용합니다.

다음 예제에서는 Lambda 함수가 호출에 제공된 이벤트 데이터를 구문 분석할 수 없는 함수 오류 메시지를 보여 줍니다.

Example 함수 오류

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

응답 본문 파라미터

- `StatusResponse` – 문자열. 상태 정보, 202 응답 코드와 함께 전송됨.
- `ErrorResponse` – 오류 응답 코드와 함께 전송되는 추가 오류 정보입니다. ErrorResponse 오류 유형과 오류 메시지가 들어 있습니다.

응답 코드

- 202 - 수락됨
- 400 - 잘못된 요청
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 런타임을 신속히 종료해야 합니다.

Example 오류 요청

```
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"
```

Lambda의 OS 전용 런타임을 사용해야 하는 경우

Lambda는 Java, Python, Node.js, .NET 및 Ruby에 대한 관리형 런타임을 제공합니다. 관리형 런타임으로 사용할 수 없는 프로그래밍 언어로 Lambda 함수를 생성하려면 OS 전용 런타임(`provided` 런타임 제품군)을 사용합니다. OS 전용 런타임에는 세 가지 기본 사용 사례가 있습니다.

- Native Ahead-of-Time(AOT) 컴파일: Go, Rust, C++와 같은 언어는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. 이러한 언어에는 컴파일된 바이너리를 실행할 수 있는 OS 환경만 필요합니다. 또한 Lambda OS 전용 런타임을 사용하여 .NET 네이티브 AOT 및 Java GraalVM 네이티브로 컴파일된 바이너리를 배포할 수 있습니다.

  바이너리에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda 런타임 API를 직접 호출하여 함수 간접 호출을 검색한 후 함수 핸들러를 직접 호출합니다. Lambda는 Go, .NET Native AOT, C++, Rust(실험용)에 대한 런타임 인터페이스 클라이언트를 제공합니다.

  Linux 환경 및 함수에 사용하려는 것과 동일한 명령 세트 아키텍처(x86_64 또는 arm64)에 맞게 바이너리를 컴파일해야 합니다.
- 타사 런타임: PHP용 Bref 또는 Swift용 Swift AWS Lambda Runtime과 같은 상용 런타임을 사용하여 Lambda 함수를 실행할 수 있습니다.
- 사용자 지정 런타임: Lambda가 관리형 런타임을 제공하지 않는 언어 또는 언어 버전에 대한 자체 런타임을 구축할 수 있습니다(예: Node.js 19). 자세한 내용은 AWS Lambda에 대한 사용자 지정 런타임 빌드 단원을 참조하십시오. 이는 OS 전용 런타임의 경우 가장 흔하지 않은 사용 사례입니다.

Lambda는 다음과 같은 OS 전용 런타임을 지원합니다.

OS 전용

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
OS 전용 런타임	`provided.al2023`	Amazon Linux 2023			
OS 전용 런타임	`provided.al2`	Amazon Linux 2			

Amazon Linux 2023(`provided.al2023`) 런타임은 작은 배포 공간과 `glibc`와 같이 업데이트된 라이브러리 버전을 포함하여 Amazon Linux 2에 비해 여러 가지 이점을 제공합니다.

`provided.al2023` 런타임은 Amazon Linux 2의 기본 패키지 관리자인 `yum` 대신 `dnf`를 패키지 관리자로 사용합니다. `provided.al2023` 및 `provided.al2` 간의 차이점에 대한 자세한 내용은 AWS Lambda을 위한 Amazon Linux 2023 런타임 소개를 AWS Compute 블로그에서 참조하십시오.

AWS Lambda에 대한 사용자 지정 런타임 빌드

AWS Lambda 런타임은 모든 프로그래밍 언어로 구현할 수 있습니다. 런타임은 함수가 호출될 때 Lambda 함수의 핸들러 메서드를 실행하는 프로그램입니다. 런타임은 함수의 배포 패키지 또는 계층에 포함될 수 있습니다. Lambda 함수를 생성할 때 OS 전용 런타임(`provided` 런타임 제품군)을 선택합니다.

> **Note**
>
> 사용자 지정 런타임을 생성하는 작업은 고급 사용 사례입니다. 네이티브 바이너리로 컴파일하거나 타사 상용 런타임을 사용하는 방법에 대한 자세한 내용은 Lambda의 OS 전용 런타임을 사용해야 하는 경우 섹션을 참조하세요.

사용자 지정 런타임 배포 프로세스에 대한 자세한 내용은 자습서: 사용자 지정 런타임 빌드 섹션을 참조하세요. GitHub에서 C++로 구현된 사용자 지정 런타임을 awslabs/aws-lambda-cpp에서 탐색할 수도 있습니다.

주제

- 요구 사항
- 사용자 지정 런타임에서 응답 스트리밍 구현

요구 사항

사용자 지정 런타임에서는 특정 초기화 및 처리 작업을 완료해야 합니다. 런타임은 함수의 설정 코드를 실행하고, 환경 변수에서 핸들러 이름을 읽고, Lambda 런타임 API에서 호출 이벤트를 읽습니다. 런타임은 이벤트 데이터를 함수 핸들러에 전달하고 핸들러의 응답을 Lambda에 다시 게시합니다.

초기화 작업

초기화 작업은 함수의 인스턴스당 한 번씩 실행되어 호출을 처리할 수 있는 환경을 준비합니다.

- 설정 검색 – 환경 변수를 읽어 함수 및 환경에 관한 세부 정보를 확인합니다.
  - _HANDLER – 함수의 구성에서 핸들러에 대한 위치입니다. 표준 형식은 *file*.*method*이며, 여기서 file은 확장명이 없는 파일의 이름이고 method는 파일에 정의된 메서드 또는 함수의 이름입니다.
  - LAMBDA_TASK_ROOT – 함수 코드가 포함된 디렉터리입니다.
  - AWS_LAMBDA_RUNTIME_API – 런타임 API의 호스트 및 포트입니다.

  사용 가능한 변수의 전체 목록은 정의된 런타임 환경 변수 섹션을 참조하세요.
- 함수 초기화 – 핸들러 파일을 로드하고 이 파일에 포함된 전역적 또는 정적 코드를 실행합니다. 함수는 SDK 클라이언트 및 데이터베이스 연결과 같은 정적 리소스를 한 번 생성해야 하며, 다중 호출 시 그러한 리소스를 다시 사용해야 합니다.
- 오류 처리 – 오류가 발생하면 초기화 오류 API를 호출하고 즉시 종료하세요.

초기화 횟수는 청구 대상인 실행 시간 및 시간 초과에 반영됩니다. 실행으로 인해 새 함수 인스턴스의 초기화가 트리거되면 로그 및 AWS X-Ray 추적에서 초기화 시간을 볼 수 있습니다.

Example 로그

```
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
```

작업 처리

런타임은 실행 중에 Lambda 런타임 인터페이스를 사용하여 수신 이벤트를 관리하고 오류를 보고합니다. 초기화 작업을 완료한 후, 런타임은 수신 이벤트를 루프에서 처리합니다. 런타임 코드에서 다음 단계를 순서대로 수행합니다.

- 이벤트 가져오기 – 다음 이벤트를 가져오려면 다음 호출 API를 호출합니다. 응답 본문에는 이벤트 데이터가 포함됩니다. 응답 헤더에는 요청 ID 및 기타 정보가 포함됩니다.
- 트레이스 헤더 전파 – API 응답의 Lambda-Runtime-Trace-Id 헤더에서 X-Ray 트레이스 헤더를 가져옵니다. _X_AMZN_TRACE_ID 환경 변수를 동일한 값으로 로컬로 설정합니다. X-Ray SDK는 이 값을 사용하여 서비스 간에 추적 데이터를 연결합니다.
- 컨텍스트 객체 생성 – API 응답에서 환경 변수 및 헤더의 컨텍스트 정보를 사용하여 객체를 생성합니다.
- 함수 핸들러 호출 – 이벤트 및 컨텍스트 객체를 핸들러에 전달합니다.
- 응답 처리 – 호출 응답 API를 호출하여 핸들러의 응답을 게시합니다.

- 오류 처리 – 오류가 발생하면 호출 오류 API를 호출합니다.
- 정리 – 사용하지 않은 리소스를 릴리스하거나 다른 서비스로 데이터를 전송하거나 혹은 다음 이벤트를 가져오기 전에 추가 작업을 수행하세요.

진입점

사용자 지정 런타임의 진입점은 bootstrap이라는 이름의 실행 파일입니다. 부트스트랩 파일은 런타임일 수 있으며 혹은 런타임을 생성하는 다른 파일을 호출할 수도 있습니다. 배포 패키지의 루트에 bootstrap 파일이 없는 경우 Lambda는 함수 계층에서 파일을 찾습니다. bootstrap 파일이 없거나 실행할 수 없는 경우, 함수는 간접 호출 시 Runtime.InvalidEntrypoint 오류를 반환합니다.

다음은 번들 버전의 Node.js를 사용하여 별도의 runtime.js 파일에서 JavaScript 런타임을 실행하는 bootstrap 파일 예제입니다.

Example 부트스트랩

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

사용자 지정 런타임에서 응답 스트리밍 구현

응답 스트리밍 함수의 경우, 런타임이 클라이언트에 부분 응답을 스트리밍하고 페이로드를 청크 단위로 반환할 수 있도록 response 및 error 엔드포인트의 동작이 약간 수정되었습니다. 특정 동작에 대한 자세한 내용은 다음 섹션을 참조하세요.

- /runtime/invocation/AwsRequestId/response - 런타임에서 Content-Type 헤더를 전파하여 클라이언트로 전송합니다. Lambda는 HTTP/1.1 청크 분할 전송 인코딩을 통해 응답 페이로드를 청크로 반환합니다. 응답 스트림의 최대 크기는 20MiB일 수 있습니다. Lambda로 응답을 스트리밍하려면 런타임에서 다음을 수행해야 합니다.
  - Lambda-Runtime-Function-Response-Mode HTTP 헤더를 streaming로 설정합니다.
  - Transfer-Encoding 헤더를 chunked로 설정합니다.
  - HTTP/1.1 청크 분할 전송 인코딩 사양을 준수하는 응답을 작성합니다.
  - 응답을 성공적으로 작성한 후 기본 연결을 닫습니다.
- /runtime/invocation/AwsRequestId/error - 이 런타임은 이 엔드포인트를 사용하여 Transfer-Encoding 헤더도 허용하는 Lambda에 함수 또는 런타임 오류를 보고할 수 있습니다. 이 엔드포인트는 런타임이 호출 응답 전송을 시작하기 전에만 호출할 수 있습니다.

- `/runtime/invocation/AwsRequestId/response`의 오류 트레일러를 사용하여 미드스트림 오류 보고 – 이 런타임은 호출 응답 작성을 시작한 후 발생하는 오류를 보고하기 위해 선택적으로 `Lambda-Runtime-Function-Error-Type` 및 `Lambda-Runtime-Function-Error-Body`라는 HTTP 후행 헤더를 연결할 수 있습니다. Lambda는 이를 성공적인 응답으로 간주하고 런타임이 클라이언트에 제공하는 오류 메타데이터를 전달합니다.

  > **Note**
  >
  > 후행 헤더를 첨부하려면 런타임이 HTTP 요청 시작 시 `Trailer` 헤더 값을 설정해야 합니다. 이는 HTTP/1.1 청크 분할 전송 인코딩 사양의 요구 사항입니다.

  - `Lambda-Runtime-Function-Error-Type` – 런타임에서 발생한 오류 유형입니다. 이 헤더는 문자열 값으로 구성됩니다. Lambda는 모든 문자열을 허용하지만 *`<category.reason>`* 형식을 사용하는 것이 좋습니다. 예: `Runtime.APIKeyNotFound`
  - `Lambda-Runtime-Function-Error-Body` - 오류에 대한 base64로 인코딩된 정보입니다.

자습서: 사용자 지정 런타임 빌드

이 자습서에서는 사용자 지정 런타임이 있는 Lambda 함수를 생성합니다. 먼저 함수의 배포 패키지에 런타임을 포함시킵니다. 그런 다음, 이 런타임을 함수와는 별도로 관리하는 계층으로 마이그레이션합니다. 마지막으로 리소스 기반 권한 정책을 업데이트하여 런타임 계층을 다른 모든 사용자와 공유합니다.

사전 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

Lambda 함수를 생성하려면 IAM 역할이 필요합니다. 역할에는 로그를 Logs로 CloudWatch 보내고 함수가 사용하는 AWS 서비스에 액세스할 수 있는 권한이 필요합니다. 함수 개발을 위한 역할이 없는 경우 이 역할을 하나 생성합니다.

실행 역할을 만들려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성(Create role)을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.

   - 신뢰할 수 있는 엔터티 – Lambda.
   - 권한 — AWSLambdaBasicExecutionRole.
   - 역할 이름 – **`lambda-role`**.

   AWSLambdaBasicExecutionRole정책에는 함수가 로그를 로그에 기록하는 데 필요한 권한이 있습니다. CloudWatch

함수 생성

사용자 지정 런타임이 있는 Lambda 함수를 생성합니다. 이 예제는 두 개의 파일 즉, 런타임 `bootstrap` 파일과 함수 핸들러를 포함합니다. 두 파일은 모두 Bash에서 구현됩니다.

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

   ```
   mkdir runtime-tutorial
   ```

```
cd runtime-tutorial
```

2. bootstrap라는 파일을 새로 생성합니다. 사용자 지정 런타임입니다.

Example 부트스트랩

```
#!/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
```

런타임은 배포 패키지에서 함수 스크립트를 로드합니다. 런타임은 두 변수를 사용하여 스크립트를 찾습니다. LAMBDA_TASK_ROOT는 패키지가 추출된 위치를 알려주며 _HANDLER에는 스크립트의 이름이 포함됩니다.

런타임은 함수 스크립트를 로딩한 후 런타임 API를 사용하여 Lambda에서 간접 호출 이벤트를 검색하고 이벤트를 핸들러로 전달하며 응답을 Lambda에 다시 게시합니다. 요청 ID를 가져오기 위해 런타임은 API 응답의 헤더를 임시 파일에 저장하고 이 파일에서 Lambda-Runtime-Aws-Request-Id 헤더를 읽습니다.

> **Note**
>
> 런타임은 오류 처리를 포함한 추가적인 책임이 있으며, 핸들러에 컨텍스트 정보를 제공합니다. 자세한 내용은 요구 사항 단원을 참조하세요.

3. 함수에 대해 스크립트를 생성합니다. 다음 스크립트 예는 이벤트 데이터를 취하여 이를 `stderr`에 기록한 후 반환하는 핸들러 함수를 정의합니다.

   Example function.sh

   ```
   function handler () {
     EVENT_DATA=$1
     echo "$EVENT_DATA" 1>&2;
     RESPONSE="Echoing request: '$EVENT_DATA'"

     echo $RESPONSE
   }
   ```

   `runtime-tutorial` 디렉터리는 이제 다음과 같아야 합니다.

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

4. 실행 파일들을 생성하여 이를 .zip 파일 아카이브에 추가합니다. 이것이 배포 패키지입니다.

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

5. `bash-runtime`이라는 이름의 함수를 생성합니다. `--role`에 Lambda 실행 역할의 ARN을 입력합니다.

   ```
   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. 함수를 호출합니다.

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

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음과 같은 응답이 표시되어야 합니다.

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

7. 응답을 확인합니다.

```
cat response.txt
```

다음과 같은 응답이 표시되어야 합니다.

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

계층 생성

함수 코드에서 런타임 코드를 분리하려면 런타임만 포함하는 계층을 생성하세요. 계층을 사용하면 함수의 종속 항목들을 독립적으로 개발할 수 있으며, 여러 함수가 있는 동일한 계층을 사용할 때 스토리지 사용량을 줄일 수 있습니다. 자세한 설명은 계층으로 Lambda 종속성 관리 섹션을 참조하세요.

1. `bootstrap` 파일을 포함하는 .zip 파일을 생성합니다.

```
zip runtime.zip bootstrap
```

2. publish-layer-version 명령을 사용하여 계층을 생성합니다.

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

그러면 해당 계층의 첫 번째 버전이 생성됩니다.

함수 업데이트

함수에 런타임 계층을 사용하려면 이 계층을 사용하도록 함수를 구성하고 함수에서 런타임 코드를 제거하세요.

1. 계층을 가져 오도록 함수 구성을 업데이트하세요.

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

이 작업으로 /opt 디렉터리의 함수에 런타임이 추가됩니다. Lambda가 계층의 런타임을 사용하도록 하려면 다음 두 단계에 표시된 것처럼 함수의 배포 패키지에서 boostrap을 제거해야 합니다.

2. 함수 코드를 포함하는 .zip 파일을 생성합니다.

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

3. 핸들러 스크립트만 포함하도록 함수 코드를 업데이트하세요.

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

4. 함수를 간접 호출하여 런타임 계층에서 작동하는지 확인합니다.

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

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음과 같은 응답이 표시되어야 합니다.

```
{
    "StatusCode": 200,
```

```
    "ExecutedVersion": "$LATEST"
}
```

5. 응답을 확인합니다.

```
cat response.txt
```

다음과 같은 응답이 표시되어야 합니다.

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

런타임 업데이트

1. 실행 환경에 관한 정보를 기록하려면 환경 변수를 출력하도록 런타임 스크립트를 업데이트하세요.

Example 부트스트랩

```
#!/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 파일의 새 버전을 포함하는 .zip 파일을 생성합니다.

```
zip runtime.zip bootstrap
```

3. bash-runtime 계층의 새로운 버전을 생성합니다.

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

4. 새 버전의 계층을 사용하도록 함수를 구성합니다.

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

계층 공유

다른 계정에 계층 사용 권한을 부여하려면 add-layer-version-permission 명령을 사용해 해당 계층 버전의 권한 정책에 명령문을 추가합니다. 각 명령문에서 단일 계정, 모든 계정 또는 조직을 대상으로 권한을 부여할 수 있습니다.

다음 예시에서는 111122223333 계정에 bash-runtime 계층의 버전 2에 대한 액세스 권한을 부여합니다.

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

다음과 유사한 출력 화면이 표시되어야 합니다.

```
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"}
```

권한은 하나의 계층 버전에만 적용됩니다. 새 계층 버전을 만들 때마다 해당 과정을 반복합니다.

정리

각 버전의 계층을 삭제합니다.

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

이 함수는 계층의 버전 2에 대한 참조가 있으므로 Lambda 내에 여전히 존재합니다. 함수는 계속 작동하며, 다만 삭제된 버전을 사용하도록 함수를 구성할 수는 없습니다. 함수의 계층 목록을 수정하려면 새 버전을 지정하거나 삭제된 계층을 생략해야 합니다.

delete-function 명령으로 함수를 삭제합니다.

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

Lambda에서 AVX2 벡터화 사용

Advanced Vector Extensions 2(AVX2)는 256비트 벡터에 대해 Single Instruction Multiple Data(SIMD) 명령을 수행할 수 있는 Intel x86 명령 세트의 벡터화 익스텐션입니다. 고도의 병렬 처리가 가능한 연산을 사용하는 벡터화 가능 알고리즘의 경우 AVX2를 사용하면 CPU 성능이 향상되어 지연 시간이 줄어들며 처리량이 향상됩니다. 기계 학습 추론, 멀티미디어 처리, 과학 시뮬레이션, 금융 모델링 애플리케이션 등과 같은 컴퓨팅 집약적인 워크로드에 AVX2 명령 세트를 사용합니다.

> **Note**
>
> Lambda arm64는 NEON SIMD 아키텍처를 사용하며 x86 AVX2 확장을 지원하지 않습니다.

Lambda 함수와 함께 AVX2를 사용하려면 함수 코드가 AVX2로 최적화된 코드에 액세스하는지 확인하세요. 일부 언어의 경우 AVX2 지원 버전의 라이브러리 및 패키지를 설치할 수 있습니다. 다른 언어의 경우 적절한 컴파일러 플래그 세트를 사용하여 코드와 종속 항목을 다시 컴파일할 수 있습니다(컴파일러가 자동 벡터화를 지원하는 경우). AVX2를 사용하여 수학 연산을 최적화하는 타사 라이브러리로 코드를 컴파일할 수도 있습니다. 예를 들어 Intel Math Kernel Library(Intel MKL), OpenBLAS(Basic Linear Algebra Subprograms), AMD BLAS-like Library Instantiation Software(BLIS) 등이 있습니다. Java와 같은 자동 벡터화 언어는 자동으로 계산에 AVX2를 사용합니다.

추가 비용 없이 새로운 Lambda 워크로드를 생성하거나 기존 AVX2 지원 워크로드를 Lambda로 이동할 수 있습니다.

AVX2에 대한 자세한 내용은 Wikipedia의 Advanced Vector Extensions 2를 참조하세요.

소스에서 컴파일

Lambda 함수가 C 또는 C++ 라이브러리를 사용하여 컴퓨팅 집약적인 벡터화 가능 연산을 수행하는 경우 적절한 컴파일러 플래그를 설정하고 함수 코드를 다시 컴파일할 수 있습니다. 그러면 컴파일러가 자동으로 코드를 벡터화합니다.

`gcc` 또는 `clang` 컴파일러의 경우 `-march=haswell`을 명령에 추가하거나 `-mavx2`를 명령 옵션으로 설정합니다.

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

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

특정 라이브러리를 사용하려면 라이브러리 설명서의 지침에 따라 라이브러리를 컴파일하고 빌드합니다. 예를 들어, TensorFlow 소스에서 빌드하려면 TensorFlow 웹 사이트의 설치 지침을 따를 수 있습니다. `-march=haswell` 컴파일 옵션을 사용해야 합니다.

Intel MKL용 AVX2 활성화

Intel MKL은 컴퓨팅 플랫폼이 AVX2 명령을 지원하는 경우 무조건적으로 AVX2 명령을 사용하는 최적화된 수학 연산 라이브러리입니다. 프레임워크는 기본적으로 인텔 MKL로 PyTorch 빌드하므로 AVX2를 활성화하지 않아도 됩니다.

와 같은 TensorFlow 일부 라이브러리는 빌드 프로세스에 인텔 MKL 최적화를 지정하는 옵션을 제공합니다. 예를 들어 TensorFlow, 와 함께 `--config=mkl` 옵션을 사용하십시오.

인텔 MKL을 사용하여 SciPy 및 와 NumPy 같은 인기 있는 과학 Python 라이브러리를 구축할 수도 있습니다. Intel MKL을 사용하여 이러한 라이브러리를 빌드하는 방법은 Intel 웹 사이트의 Numpy/Scipy with Intel MKL and Intel Compilers를 참조하세요.

인텔 MKL 및 유사 라이브러리에 대한 자세한 내용은 위키백과의 수학 커널 라이브러리, OpenBLAS 웹 사이트 및 AMD BLIS 리포지토리를 참조하십시오. GitHub

다른 언어의 AVX2 지원

C 또는 C++ 라이브러리를 사용하지 않으며 Intel MKL로 빌드하지 않는 경우에도 애플리케이션에서 어느 정도 AVX2 성능 향상을 얻을 수 있습니다. 실제 개선은 코드에서 AVX2 기능을 활용하는 컴파일러 또는 인터프리터의 능력에 달려 있습니다.

Python

Python 사용자는 일반적으로 컴퓨팅 집약적인 워크로드에 SciPy 및 NumPy 라이브러리를 사용합니다. 이러한 라이브러리를 컴파일하여 AVX2를 활성화하거나 Intel MKL 지원 버전의 라이브러리를 사용할 수 있습니다.

노드

컴퓨팅 집약적인 워크로드의 경우 필요한 라이브러리 AVX2 지원 또는 Intel MKL 지원 버전을 사용하세요.

Java

Java의 JIT 컴파일러는 AVX2 명령으로 실행되도록 코드를 자동 벡터화할 수 있습니다. 벡터화된 코드 감지에 대한 자세한 내용은 OpenJDK 웹 사이트의 Code vectorization in the JVM 프레젠테이션을 참조하세요.

Go

표준 Go 컴파일러는 현재 자동 벡터화를 지원하지 않지만 Go용 GCC 컴파일러인 gccgo를 사용할 수 있습니다. `-mavx2` 옵션을 설정하세요.

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

내장 함수

많은 언어에서 내장 함수를 사용하여 AVX2를 사용하도록 코드를 수동으로 벡터화할 수 있습니다. 그러나 이 방법은 권장하지 않습니다. 수동으로 벡터화된 코드를 작성하려면 상당한 노력이 필요합니다. 또한 자동 벡터화에 기반하는 코드를 사용하는 것보다 이러한 코드를 디버깅하고 유지 관리하는 것이 훨씬 어렵습니다.

AWS Lambda 함수 구성

Lambda API 또는 콘솔을 사용하여 Lambda 함수의 핵심 기능 및 옵션을 구성하는 방법을 알아봅니다.

메모리

함수 메모리를 늘리는 경우와 방법을 알아봅니다.

임시 스토리지

함수의 임시 스토리지 용량을 늘리는 경우와 방법을 알아봅니다.

제한 시간

함수의 제한 시간 값을 늘리는 경우와 방법을 알아봅니다.

환경 변수

환경 변수를 사용하여 함수 구성에 함수 코드를 저장하면, 함수 코드의 이식성을 높이고 코드에 보안 암호가 포함되지 않도록 할 수 있습니다.

아웃바운드 네트워킹

Amazon VPC의 AWS 리소스와 함께 Lambda 함수를 사용할 수 있습니다. 함수를 VPC에 연결하면 관계형 데이터베이스 및 캐시와 같은 프라이빗 서브넷의 리소스에 액세스할 수 있습니다.

인바운드 네트워킹

인터페이스 VPC 엔드포인트를 사용하여 공용 인터넷을 통과하지 않고 Lambda 함수를 호출할 수 있습니다.

파일 시스템

Lambda 함수를 사용하여 Amazon EFS를 로컬 디렉터리에 마운트할 수 있습니다. 파일 시스템을 사용하면 함수 코드가 안전하고 높은 동시성으로 공유 리소스에 액세스하여 수정할 수 있습니다.

별칭

클라이언트를 업데이트하는 대신, 별칭을 사용하여 특정 Lambda 함수 버전을 호출함으로써 클라이언트를 구성할 수 있습니다.

버전

함수의 버전을 게시하여 변경할 수 없는 별도의 리소스로 코드와 구성을 저장할 수 있습니다.

응답 스트리밍

응답 페이로드를 클라이언트로 다시 스트리밍하도록 Lambda 함수 URL을 구성할 수 있습니다. 응답 스트리밍은 첫 번째 바이트까지 시간(TTFB) 성능을 개선하여 지연 시간에 민감한 애플리케이션에 도움이 될 수 있습니다. 이는 부분 응답을 사용할 수 있게 되면 클라이언트에 다시 전송할 수 있기 때문입니다. 또한 응답 스트리밍을 사용하여 더 큰 페이로드를 반환하는 함수를 빌드할 수 있습니다.

Lambda 함수 메모리 구성

Lambda는 구성된 메모리 크기에 비례하여 CPU 처리능력을 할당합니다. 메모리는 런타임에 Lambda 함수가 사용할 수 있는 메모리 양입니다. 메모리 설정을 사용하면 함수에 할당된 메모리 및 CPU 성능을 늘리거나 줄일 수 있습니다. 메모리는 128MB에서 10,240MB 사이에서 1MB 단위로 구성할 수 있습니다. 1,769MB에서 함수는 하나의 vCPU와 동등한 값을 갖습니다(초당 하나의 vCPU-초 크레딧).

이 페이지에서는 Lambda 함수의 메모리 설정을 업데이트하는 경우와 방법을 설명합니다.

Sections

Lambda 함수에 대한 적절한 메모리 설정을 결정합니다.

메모리는 함수의 성능을 제어하는 주요 수단입니다. 기본 설정인 128MB는 가능한 가장 낮은 설정입니다. 이벤트를 변환하고 다른 AWS 서비스로 라우팅하는 함수와 같은 간단한 Lambda 함수에는 128MB만 사용하는 것이 좋습니다. 메모리를 더 많이 할당하면 가져온 라이브러리, Lambda 계층, Amazon Simple Storage Service(Amazon S3) 또는 Amazon Elastic File System(Amazon EFS)을 사용하는 함수의 성능이 향상될 수 있습니다. 메모리를 더 추가하면 그에 비례하여 CPU 양도 증가하여 사용 가능한 전체 컴퓨팅 파워가 증가합니다. 함수가 CPU, 네트워크 또는 메모리 바인딩인 경우 메모리 설정을 늘리면 성능이 크게 향상될 수 있습니다.

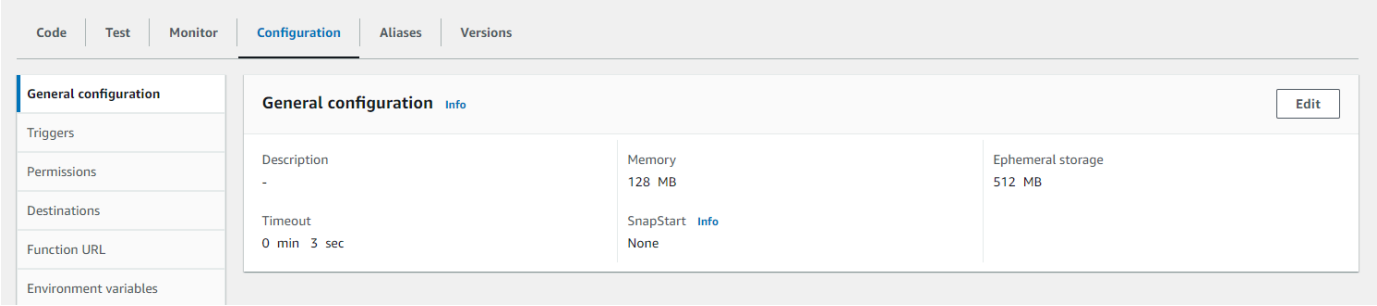
기능에 적합한 메모리 구성을 찾으려면 오픈 소스 AWS Lambda Power Tuning 도구를 사용하는 것이 좋습니다. 이 도구는 AWS Step Functions를 사용하여 서로 다른 메모리 할당에서 Lambda 함수의 여러 동시 버전을 실행하고 성능을 측정합니다. 입력 함수는 AWS 계정에서 실행되어 라이브 HTTP 호출 및 SDK 상호 작용을 수행하고 라이브 프로덕션 시나리오에서 예상 성능을 측정합니다. 또한 이 도구를 사용하도록 CI/CD 프로세스를 구현하여 배포한 새 기능의 성능을 자동으로 측정할 수 있습니다.

함수 메모리 구성(콘솔)

Lambda 콘솔에서 함수의 메모리를 구성할 수 있습니다.

함수의 메모리를 업데이트하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택한 다음 일반 구성을 선택합니다.



4. 일반 구성에서 편집을 선택합니다.
5. 메모리에 128MB에서 10,240MB 사이의 값을 설정합니다.
6. Save(저장)를 선택합니다.

함수 메모리 구성(AWS CLI)

update-function-configuration 명령을 사용하여 함수의 메모리를 구성할 수 있습니다.

Example

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

함수 메모리 구성(AWS SAM)

AWS Serverless Application Model을 사용하여 함수의 메모리를 구성할 수 있습니다. `template.yaml` 파일의 MemorySize 속성을 업데이트한 다음 sam deploy를 실행합니다.

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...
```

함수 메모리 권장 사항 수락(콘솔)

AWS Identity and Access Management(IAM)의 관리자 권한이 있는 경우 AWS Compute Optimizer에서 Lambda 함수 메모리 설정 권장 사항을 수신하도록 옵트인할 수 있습니다. 계정 또는 조직에 대한 메모리 권장 사항을 옵트인하는 방법에 대한 지침은 AWS Compute Optimizer 사용 설명서의 계정 옵트인을 참조하세요.

> **Note**
>
> Compute Optimizer는 x86_64 아키텍처를 사용하는 함수만 지원합니다.

옵트인한 상태에서 Lambda 함수가 Compute Optimizer 요구 사항을 충족하는 경우 Lambda 콘솔의 일반 구성에서 Compute Optimizer의 함수 메모리 권장 사항을 보고 수락할 수 있습니다.

Lambda 함수에 대한 임시 스토리지 구성

Lambda는 /tmp 디렉터리의 함수를 위한 임시 스토리지를 제공합니다. 이 스토리지는 임시이며 각 실행 환경에 고유합니다. 임시 스토리지 설정을 사용하여 함수에 할당되는 임시 스토리지의 양을 제어할 수 있습니다. 임시 스토리지는 512MB에서 10,240MB 사이에서 1MB 단위로 구성할 수 있습니다. /tmp에 저장된 모든 데이터는 AWS에서 관리되는 키를 사용하여 저장 시 암호화됩니다.

이 페이지에서는 일반적인 사용 사례와 Lambda 함수의 임시 스토리지를 업데이트하는 방법을 설명합니다.

Sections

증가된 임시 스토리지에 대한 일반적인 사용 사례

다음은 증가된 임시 스토리지가 이점이 되는 몇 가지 일반적인 사용 사례입니다.

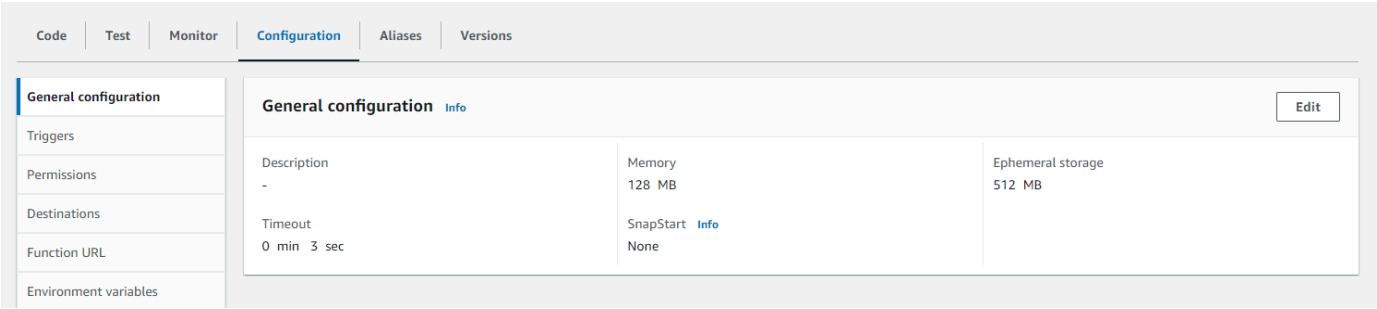
- ETL(추출, 변환 및 로드) 작업: 코드가 중간 계산을 수행하거나 처리를 완료하기 위해 다른 리소스를 다운로드하는 경우 임시 스토리지를 늘립니다. 임시 스토리지가 늘어나면 Lambda 함수에서 더 복잡한 ETL 작업을 실행할 수 있습니다.
- 기계 학습(ML) 추론: 많은 추론 작업은 라이브러리 및 모델을 포함한 대용량 참조 데이터 파일에 의존합니다. 임시 스토리지가 더 많으면 Amazon Simple Storage Service(Amazon S3)에서 /tmp로 더 큰 모델을 다운로드하여 처리에 사용할 수 있습니다.
- 데이터 처리: S3 이벤트에 대한 응답으로 Amazon S3에서 개체를 다운로드하는 워크로드의 경우 /tmp 스토리지가 많으면 인메모리 처리를 사용하지 않고도 더 큰 개체를 처리할 수 있습니다. PDF를 생성하거나 미디어를 처리하는 워크로드에도 더 많은 임시 스토리지가 이점이 될 수 있습니다.
- 그래픽 처리: 이미지 처리는 Lambda 기반 애플리케이션의 일반적인 사용 사례입니다. 대용량 TIFF 파일 또는 위성 이미지를 처리하는 워크로드의 경우 임시 스토리지가 많을수록 Lambda에서 라이브러리를 사용하고 계산을 수행하기가 더 쉬워집니다.

임시 스토리지(콘솔) 구성

Lambda 콘솔에서 임시 스토리지를 구성할 수 있습니다.

함수의 임시 스토리지를 수정하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택한 다음 일반 구성을 선택합니다.



4. 일반 구성에서 편집을 선택합니다.
5. 임시 스토리지 값을 512MB에서 10,240MB 사이에서 1MB 단위로 설정합니다.
6. Save(저장)를 선택합니다.

임시 스토리지 구성(AWS CLI)

update-function-configuration 명령을 사용하여 임시 스토리지를 구성할 수 있습니다.

Example

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

임시 스토리지 구성(AWS SAM)

AWS Serverless Application Model을 사용하여 함수의 임시 스토리지를 구성할 수 있습니다. `template.yaml` 파일의 EphemeralStorage 속성을 업데이트한 다음 sam deploy를 실행합니다.

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 함수 제한 시간 구성

Lambda는 제한 시간 이전에 설정된 시간 동안 코드를 실행합니다. 제한 시간은 Lambda 함수를 실행할 수 있는 최대 시간(초)입니다. 이 설정의 기본값은 3초이지만 1초 단위로 최대 900초(15분)까지 값을 조정할 수 있습니다.

이 페이지에서는 Lambda 함수의 제한 시간 설정을 업데이트하는 경우와 방법을 설명합니다.

Sections

Lambda 함수에 대한 적절한 제한 시간 값을 결정합니다.

제한 시간 값이 함수의 평균 지속 시간에 가까울수록 함수가 예기치 않게 시간 초과될 위험이 커집니다. 함수의 지속 시간은 데이터 전송량 및 처리량과 함수가 상호 작용하는 서비스의 지연 시간에 따라 달라질 수 있습니다. 시간 초과의 몇 가지 일반적인 원인은 다음과 같습니다.

- Amazon Simple Storage Service(Amazon S3)의 다운로드가 평균보다 크거나 시간이 더 걸립니다.
- 함수가 다른 서비스에 요청을 보내면 응답하는 데 시간이 더 오래 걸립니다.
- 함수에 제공되는 파라미터가 있으면 함수의 계산 복잡성이 커지므로 호출 시간이 더 오래 걸립니다.

애플리케이션을 테스트할 때 테스트에 데이터의 크기 및 양과 실제 파라미터 값이 정확하게 반영되는지 확인하십시오. 테스트에서는 편의상 작은 샘플을 사용하는 경우가 많지만 워크로드에 대해 합리적으로 예상되는 상한에 존재하는 데이터세트를 사용해야 합니다.

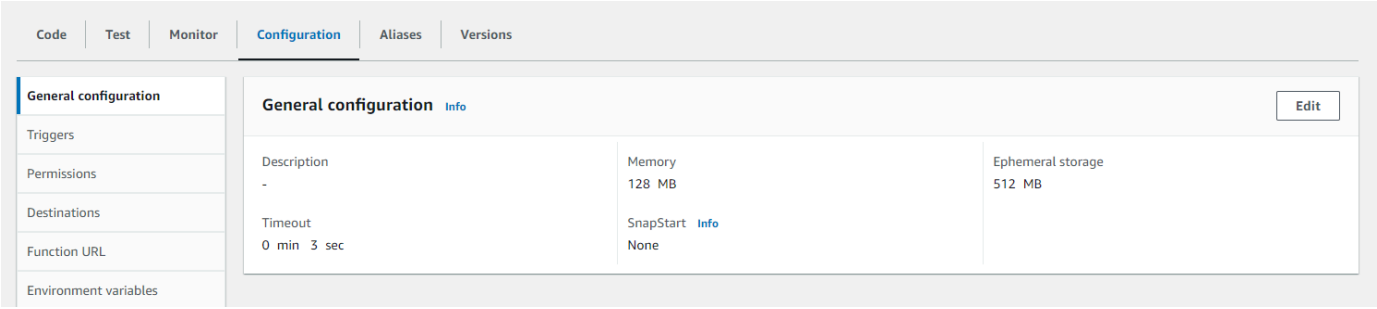
제한 시간 구성(콘솔)

Lambda 콘솔에서 함수 제한 시간을 구성할 수 있습니다.

함수에 대한 제한 시간을 수정하려면 다음을 수행합니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.

3. 구성 탭을 선택한 다음 일반 구성을 선택합니다.



4. 일반 구성에서 편집을 선택합니다.
5. 제한 시간에 1초부터 900초(15분) 사이의 값을 설정합니다.
6. Save(저장)를 선택합니다.

제한 시간 구성(AWS CLI)

update-function-configuration 명령을 사용하여 제한 시간 값을 초 단위로 구성할 수 있습니다. 다음 예제 명령에서는 함수 제한 시간을 120초(2분)로 늘립니다.

Example

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

제한 시간 구성(AWS SAM)

AWS Serverless Application Model을 사용하여 함수의 제한 시간 값을 구성할 수 있습니다. `template.yaml` 파일의 Timeout 속성을 업데이트한 다음 sam deploy를 실행합니다.

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 환경 변수를 사용하여 코드의 값 구성

환경 변수를 사용하면 코드를 업데이트하지 않고도 함수의 동작을 조정할 수 있습니다. 환경 변수는 함수의 버전별 구성에 저장된 문자열 쌍입니다. Lambda 런타임은 코드에 환경 변수를 사용할 수 있게 하고 함수 및 호출 요청에 대한 정보가 포함된 추가 환경 변수를 설정합니다.

> **Note**
>
> 보안을 강화하려면 환경 변수 대신 AWS Secrets Manager를 사용하여 데이터베이스 자격 증명과 API 키 또는 권한 부여 토큰과 같은 기타 민감한 정보를 저장하는 것이 좋습니다. 자세한 내용은 AWS Secrets Manager로 암호 생성 및 관리를 참조하세요.

환경 변수는 함수 간접 호출 전에 평가되지 않습니다. 정의한 모든 값은 리터럴 문자열로 간주되며 확장되지 않습니다. 함수 코드에서 변수 평가를 수행합니다.

Lambda 콘솔, AWS Command Line Interface(AWS CLI), AWS Serverless Application Model(AWS SAM) 또는 AWS SDK를 사용하여 Lambda에서 환경 변수를 구성할 수 있습니다.

Console

게시되지 않은 함수 버전에서 환경 변수를 정의합니다. 버전을 게시할 때 환경 변수는 다른 버전별 구성 설정과 함께 해당 버전에 대해 잠금 상태가 됩니다.

키와 값을 정의하여 함수에 환경 변수를 생성합니다. 함수는 키 이름을 사용하여 환경 변수 값을 검색합니다.

Lambda 콘솔에서 환경 변수를 설정하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성을 선택한 후 환경 변수를 선택합니다.
4. Environment variables(환경 변수)에서 편집을 선택합니다.
5. Add environment variable(환경 변수 추가)을 선택합니다.
6. 키와 값을 입력합니다.

   요구 사항

   - 키는 문자로 시작되며 최소 2자입니다.

- 키에는 문자, 숫자 및 밑줄(_)만 포함됩니다.
- 키는 Lambda에 의해 예약되지 않습니다.
- 모든 환경 변수의 총 크기는 4KB를 초과하지 않습니다.

7. Save(저장)를 선택합니다.

콘솔 코드 편집기에서 환경 변수 목록을 생성하려면

Lambda 코드 편집기에서 환경 변수 목록을 생성할 수 있습니다. 이는 코딩하는 동안 환경 변수를 빠르게 참조할 수 있는 방법입니다.

1. 코드 탭을 선택합니다.
2. 환경 변수 탭을 선택합니다.
3. 도구, 환경 변수 표시를 선택합니다.

콘솔 코드 편집기에 나열되어 있는 환경 변수는 암호화된 상태로 유지됩니다. 전송 중 암호화에 대해 암호화 도우미를 활성화한 경우 해당 설정은 변경되지 않습니다. 자세한 내용은 Lambda 환경 변수 보안 단원을 참조하십시오.

환경 변수 목록은 읽기 전용이며 Lambda 콘솔에서만 사용할 수 있습니다. 이 파일은 함수의 .zip 파일 아카이브를 다운로드할 때 포함되지 않으며, 이 파일을 업로드하여 환경 변수를 추가할 수 없습니다.

AWS CLI

다음 예제에서는 `my-function`라는 함수에서 두 개의 환경 변수를 설정합니다 .

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

`update-function-configuration` 명령을 사용하여 환경 변수를 적용하면 `Variables` 구조의 전체 내용이 바뀝니다. 새 환경 변수를 추가할 때 기존 환경 변수를 유지하려면 요청에 기존 값을 모두 포함시킵니다.

현재 구성을 가져오려면 `get-function-configuration` 명령을 사용합니다.

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

다음 결과가 표시됩니다.

```
{
    "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` 출력의 개정 ID를 파라미터로 `update-function-configuration`에 전달할 수 있습니다. 이렇게 하면 구성을 읽을 때와 업데이트할 때 사이에 값이 변경되지 않습니다.

함수의 암호화 키를 구성하려면 `KMSKeyARN` 옵션을 설정합니다.

```
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 Model을 사용하여 함수에 대한 환경 변수를 구성할 수 있습니다. `template.yaml` 파일의 환경 및 변수 속성을 업데이트한 다음 sam deploy를 실행합니다.

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 SDK를 사용하여 환경 변수를 관리하려면 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

자세한 내용은 선호하는 프로그래밍 언어에 대한 AWS SDK 설명서를 참조하세요.

정의된 런타임 환경 변수

Lambda 런타임은 초기화 중에 여러 환경 변수를 설정합니다. 대부분의 환경 변수는 함수 또는 런타임에 관한 정보를 제공합니다. 이러한 환경 변수의 키는 예약되어 있으며 함수 구성에서 설정할 수 없습니다.

예약된 환경 변수

- _HANDLER – 함수에 대해 구성된 핸들러 위치입니다.
- _X_AMZN_TRACE_ID – X-Ray 추적 헤더입니다. 이 환경 변수는 간접 호출할 때마다 변경됩니다.
  - 이 환경 변수는 OS 전용 런타임(provided 런타임 제품군)에서 정의되지 않았습니다. 다음 호출의 Lambda-Runtime-Trace-Id 응답 헤더를 사용하여 사용자 지정 런타임에 _X_AMZN_TRACE_ID을(를) 설정할 수 있습니다.

  - Java 런타임 버전 17 이상에서는 이 환경 변수가 사용되지 않습니다. 대신 Lambda는 `com.amazonaws.xray.traceHeader` 시스템 속성에 추적 정보를 저장합니다.
- `AWS_DEFAULT_REGION` – Lambda 함수가 실행되는 기본 AWS 리전입니다.
- `AWS_REGION` – Lambda 함수가 실행되는 AWS 리전입니다. 정의되면 이 값이 `AWS_DEFAULT_REGION`을 재정의합니다.
  - AWS SDK와 함께 AWS 리전 환경 변수를 사용하는 방법에 대한 자세한 내용은 AWS SDK 및 도구 참조 안내서의 AWS Region을 참조하세요.
- `AWS_EXECUTION_ENV` - 런타임 ID로서 앞에 `AWS_Lambda_`가 붙습니다(예: `AWS_Lambda_java8`). 이 환경 변수는 OS 전용 런타임(`provided` 런타임 제품군)에서 정의되지 않았습니다.
- `AWS_LAMBDA_FUNCTION_NAME` – 함수의 이름입니다.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – 함수에 사용 가능한 총 메모리 양(MB)입니다.
- `AWS_LAMBDA_FUNCTION_VERSION` – 실행할 함수의 버전입니다.
- `AWS_LAMBDA_INITIALIZATION_TYPE` – 함수의 초기화 유형이며, `on-demand`, `provisioned-concurrency` 또는 `snap-start`입니다. 자세한 내용은 프로비저닝된 동시성 구성 또는 Lambda SnapStart를 사용하여 시작 성능 개선 섹션을 참조하세요.
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` – 함수에 대한 Amazon CloudWatch Logs 그룹 및 스트림의 이름입니다. `AWS_LAMBDA_LOG_GROUP_NAME` 및 `AWS_LAMBDA_LOG_STREAM_NAME` 환경 변수는 Lambda SnapStart 함수에서 사용할 수 없습니다.
- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` - 함수의 실행 역할에서 가져온 액세스 키입니다.
- `AWS_LAMBDA_RUNTIME_API` – (사용자 지정 런타임) 런타임 API의 호스트 및 포트입니다.
- `LAMBDA_TASK_ROOT` – Lambda 함수 코드의 경로입니다.
- `LAMBDA_RUNTIME_DIR` – 런타임 라이브러리의 경로입니다.

다음 추가 환경 변수는 예약되어 있지 않으며 함수 구성에서 확장할 수 있습니다.

예약되지 않은 환경 변수

- `LANG` - 이것은 런타임의 로캘입니다(`en_US.UTF-8`).
- `PATH` - 실행 경로(`/usr/local/bin:/usr/bin/:/bin:/opt/bin`)입니다.
- `LD_LIBRARY_PATH` 시스템 라이브러리 경로(`/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib`)입니다.

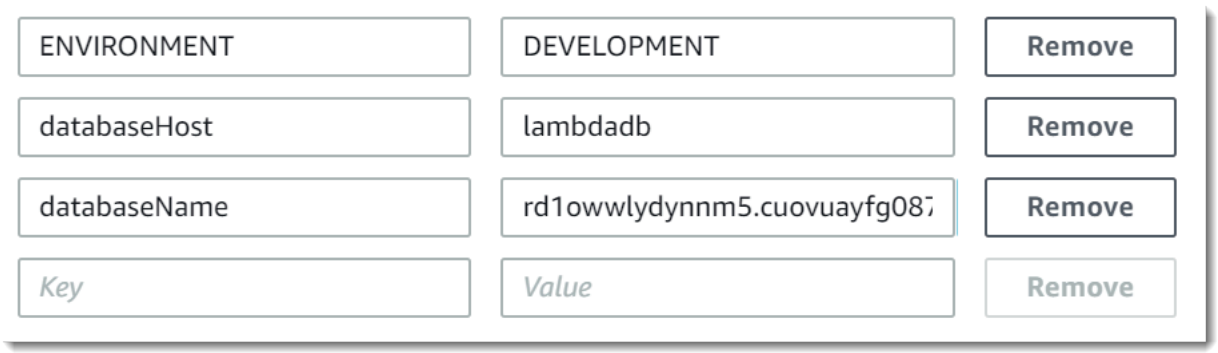
- NODE_PATH – (Node.js) Node.js 라이브러리 경로(/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules)입니다.
- PYTHONPATH – (Python 2.7, 3.6, 3.8) Python 라이브러리 경로($LAMBDA_RUNTIME_DIR)입니다.
- GEM_PATH – (Ruby) Ruby 라이브러리 경로($LAMBDA_TASK_ROOT/vendor/bundle/ruby/2.5.0:/opt/ruby/gems/2.5.0)입니다.
- AWS_XRAY_CONTEXT_MISSING – X-Ray 추적의 경우 Lambda는 X-Ray SDK에서 런타임 오류가 발생하지 않도록 LOG_ERROR로 설정합니다.
- AWS_XRAY_DAEMON_ADDRESS – X-Ray 추적의 경우 X-Ray 데몬의 포트 및 IP 주소입니다.
- AWS_LAMBDA_DOTNET_PREJIT – .NET 6 및 .NET 7 런타임의 경우 이 변수를 설정하여 .NET 특정 런타임 최적화를 활성화하거나 비활성화합니다. 값에는 always, never 및 provisioned-concurrency가 포함됩니다. 자세한 내용은 함수에 대해 프로비저닝된 동시성 구성 단원을 참조하십시오.
- TZ - 환경의 표준 시간대(UTC)입니다. 실행 환경에서는 NTP를 사용하여 시스템 클록을 동기화합니다.

표시된 샘플 값은 최신 런타임을 반영합니다. 특정 변수 또는 해당 값의 존재는 이전 런타임에 따라 다를 수 있습니다.

환경 변수에 대한 예제 시나리오

환경 변수를 사용하여 테스트 환경 및 프로덕션 환경에서 함수 동작을 사용자 지정할 수 있습니다. 예를 들어, 코드는 같지만 구성이 다른 두 개의 함수를 생성할 수 있습니다. 한 함수는 테스트 데이터베이스에 연결되고, 다른 함수는 프로덕션 데이터베이스에 연결됩니다. 이 경우 환경 변수를 사용하여 데이터베이스의 호스트 이름 및 기타 연결 세부 정보를 함수에 전달합니다.

다음 예는 환경 변수로 데이터베이스 호스트 및 데이터베이스 이름을 정의하는 방법을 보여줍니다.



테스트 환경에서 프로덕션 환경보다 더 많은 디버그 정보를 생성하도록 하려면 환경 변수를 설정하여 더 많은 상세 표시 로깅 또는 더 자세한 추적을 사용하도록 테스트 환경을 구성하면 됩니다.

Lambda 환경 변수 보안

환경 변수 보안을 위해 서버측 암호화를 이용해 저장 데이터를 보호하고 클라이언트측 암호화를 이용해 전송 중 데이터를 보호할 수 있습니다.

> **Note**
>
> 데이터베이스 보안을 강화하려면 환경 변수 대신 AWS Secrets Manager를 사용하여 데이터베이스 보안 인증 정보를 저장하는 것이 좋습니다. 자세한 내용은 Amazon RDS와 함께 AWS Lambda 사용 단원을 참조하십시오.

저장 중 보안

Lambda는 AWS KMS key를 이용해 항상 저장 중 서버 측 암호화를 제공합니다. 기본적으로 Lambda는 AWS 관리형 키를 사용합니다. 이 기본 동작이 워크플로에 적합한 경우 다른 작업을 설정할 필요가 없습니다. Lambda가 계정에 AWS 관리형 키를 생성하고 계정에 대한 허가를 관리합니다. AWS는 이 키를 사용하는 데 요금을 청구하지 않습니다.

원한다면 AWS KMS 고객 관리형 키를 대신 제공할 수 있습니다. KMS 키의 교체를 제어하거나 KMS 키를 관리하기 위한 조직의 요구 사항을 충족하기 위해 이 작업을 수행할 수 있습니다. 고객 관리형 키를 사용하면 KMS 키에 대한 액세스 권한이 있는 계정의 사용자만 함수에서 환경 변수를 보거나 관리할 수 있습니다.

고객 관리형 키에는 표준 AWS KMS 요금이 발생합니다. 자세한 내용은 AWS Key Management Service 요금을 참조하십시오.

전송 중 보안

보안을 강화하기 위해 전송 중 암호화를 위한 도우미를 사용하도록 설정하면 전송 중 보호를 위해 환경 변수가 클라이언트 측으로 암호화됩니다.

환경 변수 암호화를 구성하려면

1. AWS Key Management Service(AWS KMS)를 사용하여 서버 측 및 클라이언트 측 암호화에 사용할 Lambda에 대한 고객 관리형 키를 생성할 수 있습니다. 자세한 내용은 AWS Key Management Service 개발자 안내서에서 키 생성을 참조하세요.

2. Lambda 콘솔을 사용하여 환경 변수 편집 페이지로 이동합니다.

   a. Lambda 콘솔의 함수 페이지를 엽니다.

   b. 함수를 선택합니다.

   c. 구성을 선택한 다음 왼쪽 탐색 모음에서 환경 변수를 선택합니다.

   d. 환경 변수 섹션에서 편집을 선택합니다.

   e. Encryption configuration(암호화 구성)을 확장합니다.

3. (선택 사항) 클라이언트측 암호화를 사용하여 전송 중 데이터를 보호하도록 콘솔 암호화 도우미를 활성화합니다.

   a. 전송 중 암호화에서 전송 중 암호화에 대해 도우미 사용을 선택합니다.

   b. 콘솔 암호화 도우미를 사용 설정하려는 각 환경 변수에 대해 환경 변수 옆에 있는 암호화를 선택합니다.

   c. 전송 중 암호화할 AWS KMS key에서, 이 절차를 시작할 때 생성한 고객 관리형 키를 선택합니다.

   d. 실행 역할 정책을 선택하고 정책을 복사합니다. 이 정책은 환경 변수를 복호화할 수 있는 권한을 함수의 실행 역할에 부여합니다.

      이 정책을 저장하여 절차의 마지막 단계에서 사용합니다.

   e. 환경 변수를 복호화하는 함수에 코드를 추가합니다. 예제를 보려면 암호 코드 조각 해독을 선택합니다.

4. (선택 사항) 유휴 시 암호화를 위해 고객 관리형 키를 지정합니다.

   a. 고객 마스터 키 사용을 선택합니다.

   b. 이 절차를 시작할 때 생성한 고객 관리형 키를 선택합니다.

5. Save(저장)를 선택합니다.

6. 권한을 설정합니다.

   서버 측 암호화와 함께 고객 관리형 키를 사용하는 경우, 함수에서 환경 변수를 보거나 관리할 수 있는 사용자 또는 역할에 권한을 부여하세요. 자세한 내용은 서버 측 암호화 KMS 키에 대한 권한 관리 단원을 참조하십시오.

   전송 중 보안을 위해 클라이언트 측 암호화를 사용하도록 설정하는 경우, 함수가 `kms:Decrypt` API 작업을 호출하려면 권한이 필요합니다. 이전에 이 절차에서 저장한 정책을 함수의 실행 역할에 추가합니다.

서버 측 암호화 KMS 키에 대한 권한 관리

사용자 또는 함수의 실행 역할에 대한 AWS KMS 권한 없이도 기본 암호화 키를 사용할 수 있습니다. 고객 관리형 CMK를 사용하려면 키 사용 권한이 필요합니다. Lambda는 사용자의 권한을 사용하여 키에 대한 권한 부여를 생성합니다. 이를 통해 Lambda가 암호화에 해당 키를 사용할 수 있습니다.

- `kms:ListAliases` – Lambda 콘솔에서 키 보기.
- `kms:CreateGrant`, `kms:Encrypt` – 함수에 대한 고객 관리형 키 구성하기
- `kms:Decrypt` – 고객 관리형 키로 암호화된 환경 변수를 보고 관리하기

AWS 계정 또는 키의 리소스 기반 권한 정책에서 이러한 권한을 얻을 수 있습니다. `ListAliases`는 Lambda의 관리형 정책에서 제공합니다. 키 정책에 따라 키 사용자 그룹의 사용자는 나머지 권한을 받습니다.

`Decrypt` 권한이 없는 사용자도 기능을 관리할 수 있지만 Lambda 콘솔에서 환경 변수를 보거나 관리할 수는 없습니다. 사용자가 환경 변수를 볼 수 없도록 하려면 기본 키, 고객 관리형 키 또는 모든 키에 대한 액세스를 거부하는 사용자 권한에 문을 추가합니다.

Example IAM 정책 - 키 ARN별 액세스 거부

```
{
    "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"
        }
    ]
}
```

키 권한 관리에 대한 자세한 내용은 AWS Key Management Service 개발자 안내서의 Key policies in AWS KMS를 참조하세요.

Lambda 환경 변수 검색

함수 코드에서 환경 변수를 검색하려면 프로그래밍 언어에 대한 표준 메서드를 사용합니다.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> 경우에 따라 다음 형식을 사용해야 할 수 있습니다.
>
> ```
> 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
```

Lambda는 저장 시 환경 변수를 암호화하여 안전하게 저장합니다. 다른 암호화 키를 사용하도록 Lambda를 구성하거나, 클라이언트 측의 환경 변수 값을 암호화하거나, AWS Secrets Manager를 사용해 AWS CloudFormation 템플릿에서 환경 변수를 설정할 수 있습니다.

Lambda 함수에 Amazon VPC의 리소스에 대한 액세스 권한 부여

Amazon Virtual Private Cloud(VPC)를 사용하면 AWS 계정에 프라이빗 네트워크를 생성하여 Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스, Amazon Relational Database Service(RDS) 인스턴스 및 Amazon ElastiCache 인스턴스 같은 리소스를 호스팅할 수 있습니다. 리소스가 포함된 프라이빗 서브넷을 통해 함수를 VPC에 연결하여 Lambda 함수에 Amazon VPC에서 호스팅된 리소스에 대한 액세스 권한을 부여할 수 있습니다. Lambda 콘솔 AWS Command Line Interface(AWS CLI) 또는 AWS SAM을 사용하여 Amazon VPC에 Lambda 함수를 연결하려면 다음 섹션의 지침을 따릅니다.

> **Note**
>
> 모든 Lambda 함수는 Lambda 서비스가 소유하고 관리하는 VPC 내에서 실행됩니다. 이러한 VPC는 Lambda에 의해 자동으로 유지 관리되며 고객에게는 표시되지 않습니다. Amazon VPC의 다른 AWS 리소스에 액세스하도록 함수를 구성해도 함수가 내부에서 실행하는 Lambda 관리형 VPC에는 영향을 미치지 않습니다.

Sections

필수 IAM 권한

Lambda 함수를 AWS 계정의 Amazon VPC에 연결하려면 Lambda가 사용하는 네트워크 인터페이스를 생성하고 관리할 권한이 있어야 함수에 VPC의 리소스에 대한 액세스 권한을 부여할 수 있습니다.

Lambda가 생성하는 네트워크 인터페이스는 Hyperplane 탄력적 네트워크 인터페이스 또는 Hyperplane ENI라고 알려져 있습니다. 이러한 네트워크 인터페이스에 대한 자세한 내용은 the section called “Hyperplane 탄력적 네트워크 인터페이스(ENIs) 이해하기” 섹션을 참조하세요.

함수의 실행 역할에 AWS 관리형 정책 AWSLambdaVpcAccessExecutionRole을 연결하여 함수에 필요한 권한을 부여할 수 있습니다. Lambda 콘솔에서 새 함수를 생성하여 VPC에 연결하면 Lambda는 자동으로 이 권한 정책을 추가합니다.

자체 IAM 권한 정책을 생성하고 싶다면 다음 권한을 모두 추가해야 합니다.

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces— 이 작업은 모든 리소스에서 허용된 경우에만 작동합니다 (`"Resource": "*"`).
- ec2:DescribeSubnets
- ec2:DeleteNetworkInterface - 실행 역할에서DeleteNetworkInterface에 대한 리소스 ID를 지정하지 않은 경우 함수가 VPC 액세스하지 못할 수 있습니다. 고유한 리소스 ID를 지정하거나 모든 리소스 ID(예: `"Resource": "arn:aws:ec2:us-west-2:123456789012:*/*"`)를 포함합니다.
- ec2:AssignPrivateIpAddresses
- ec2:UnassignPrivateIpAddresses

참고로 함수 역할에는 네트워크 인터페이스를 생성할 때만 이러한 권한이 필요하며 함수를 호출할 때는 필요하지 않습니다. 함수의 실행 역할에서 이러한 권한을 제거하더라도 함수가 Amazon VPC에 연결된 경우 여전히 성공적으로 함수를 호출할 수 있습니다.

함수를 VPC에 연결하려면 Lambda는 또한 IAM 사용자 역할을 사용하여 네트워크 리소스를 확인해야 합니다. 사용자 역할에 IAM 권한이 있는지 확인해야 합니다.

- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs

> **Note**
>
> 함수의 실행 역할에 부여한 Amazon EC2 권한은 Lambda 서비스에서 함수를 VPC에 연결하는 데 사용됩니다. 그러나 이러한 권한을 함수 코드에도 암시적으로 부여하고 있습니다. 이는 함수 코드가 이러한 Amazon EC2 API 호출을 수행할 수 있다는 의미입니다. 보안 모범 사례 준수에 대한 자세한 내용은 the section called “보안 모범 사례” 섹션을 참조하세요.

Lambda 함수를 AWS 계정의 Amazon VPC에 연결하기

Lambda 콘솔인 AWS CLI 또는 AWS SAM을 사용하여 함수를 AWS 계정의 Amazon VPC에 연결합니다. AWS CLI 또는 AWS SAM을 사용하는 경우나 Lambda 콘솔을 사용하여 VPC에 기존 함수를 연결하는 경우에는 함수의 실행 역할에 이전 섹션에서 나열된 필수 권한이 있는지 확인합니다.

Lambda 함수는 전용 인스턴스 테넌시를 사용하여 VPC에 직접 연결할 수 없습니다. 전용 VPC의 리소스에 연결하려면, 기본 테넌시를 사용하여 두 번째 VPC에 피어로 연결합니다.

Lambda console

Amazon VPC를 생성할 때 APC Amazon에 함수를 연결하려면

1. Lambda 콘솔의 함수 페이지를 열고 Create function(함수 생성)을 선택합니다.
2. 기본 정보(Basic information)에서 함수 이름(Function name)에 함수 이름을 입력합니다.
3. 다음을 수행하여 함수에 대한 VPC 설정을 구성합니다.

   a. Advanced settings(고급 설정)를 확장합니다.
   b. VPC 활성화를 선택한 다음 함수를 연결할 VPC를 선택합니다.
   c. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
   d. 네트워크 인터페이스를 생성할 서브넷 및 보안 그룹을 선택합니다. 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

   > **Note**
   >
   > 프라이빗 리소스에 연결하려면 함수를 프라이빗 서브넷에 액세스합니다. 함수에 인터넷 액세스가 필요한 경우 the section called “VPC 함수에 대한 인터넷 액세스” 섹션을 참조하세요. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스 권한 또는 퍼블릭 IP 주소는 제공되지 않습니다.

4. 함수 생성을 선택합니다.

Amazon VPC에 기존 함수를 연결하려면

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 구성 탭을 선택한 다음 VPC를 선택합니다.

3. 편집을 선택합니다.
4. VPC에서 함수를 연결할 Amazon VPC를 선택합니다.
5. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
6. 네트워크 인터페이스를 생성할 서브넷 및 보안 그룹을 선택합니다. 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

> **Note**
>
> 프라이빗 리소스에 연결하려면 함수를 프라이빗 서브넷에 액세스합니다. 함수에 인터넷 액세스가 필요한 경우 the section called “VPC 함수에 대한 인터넷 액세스” 섹션을 참조하세요. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스 권한 또는 퍼블릭 IP 주소는 제공되지 않습니다.

7. Save(저장)를 선택합니다.

AWS CLI

Amazon VPC를 생성할 때 APC Amazon에 함수를 연결하려면

- Lambda 함수를 생성하고 VPC에 연결하려면 다음 CLI `create-function` 명령을 실행합니다.

```
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
```

자체 서브넷과 보안 그룹을 지정하고 사용 사례에 따라 `Ipv6AllowedForDualStack`을 `true` 또는 `false`에 설정합니다.

Amazon VPC에 기존 함수를 연결하려면

- VPC에 기존 함수를 연결하려면 다음 CLI `update-function-configuration` 명령을 실행합니다.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

VPC에서 함수를 연결 해제하려면

- VPC에서 함수를 연결 해제하려면 VPC 서브넷 및 보안 그룹의 빈 목록을 사용하여 다음 `update-function-configuration`CLI 명령을 실행합니다.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

함수를 VPC에 연결하려면

- Amazon VPC에 Lambda 함수를 연결하려면 다음 예제 템플릿과 같이 함수 정의에 `VpcConfig` 속성을 추가합니다. 이 속성에 대한 자세한 내용은 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function VpcConfig를 참조하세요(AWS SAM `VpcConfig` 속성은 AWS CloudFormation `AWS::Lambda::Function` 리소스의 `VpcConfig` 속성으로 직접 전달됨).

```
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 SAM에서 VPC 구성에 대한 자세한 내용은 AWS CloudFormation 사용 설명서의 AWS::EC2::VPC를 참조하세요.

VPC에 연결 시 인터넷 액세스

기본적으로 Lambda 함수는 퍼블릭 인터넷에 액세스할 수 있습니다. 함수를 VPC에 연결하는 경우 해당 VPC 내에서 사용 가능한 리소스에만 액세스할 수 있습니다. 함수에 인터넷 액세스 권한을 부여하려면 인터넷에 액세스 할 수 있도록 VPC도 구성해야 합니다. 자세한 내용은 the section called “VPC 함수에 대한 인터넷 액세스”을 참조하십시오.

Amazon VPC로 Lambda를 사용하는 모범 사례

Lambda VPC 구성이 모범 사례 지침을 준수하려면 다음 섹션의 조언을 따릅니다.

보안 모범 사례

Lambda 함수를 VPC에 연결하려면 함수의 실행 역할에 여러 Amazon EC2 권한을 부여해야 합니다. 함수가 VPC의 리소스에 액세스하는 데 사용하는 네트워크 인터페이스를 생성하려면 이러한 권한이 필요합니다. 그러나 이러한 권한은 함수 코드에도 암시적으로 부여됩니다. 이는 함수 코드에 이러한 Amazon EC2 API 호출을 수행할 수 있는 권한이 있다는 의미입니다.

최소 권한 액세스 원칙을 따르려면 함수의 실행 역할에 다음 예시와 같은 거부 정책을 추가합니다. 이 정책은 Lambda 서비스가 함수를 VPC에 연결하는 데 사용하는 Amazon EC2 API를 함수가 호출하지 못하게 합니다.

```
{
    "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"
                    ]
                }
            }
        }
    ]
}
```

AWS는 VPC의 보안을 강화하기 위해 보안 그룹과 네트워크 액세스 제어 목록(ACL)을 제공합니다. 보안 그룹은 리소스용 인바운드 및 아웃바운드 트래픽을 제어하고, 네트워크 ACL은 서브넷용 인바운드 및 아웃바운드 트래픽을 제어합니다. 보안 그룹은 대부분의 서브넷에 대해 충분한 액세스 제어를 제공합니다. VPC에 대한 추가 보안 계층을 원하는 경우 네트워크 ACL을 사용할 수 있습니다. Amazon

VPC를 사용할 때의 보안 모범 사례에 대한 일반 지침은 Amazon Virtual Private Cloud 사용 설명서에서 VPC의 보안 모범 사례를 참조하세요.

성능 모범 사례

함수를 VPC에 연결하면 Lambda는 연결에 사용할 수 있는 이용 가능한 네트워크 리소스(Hyperplane ENI)가 있는지 확인합니다. Hyperplane ENI는 보안 그룹 및 VPC 서브넷의 특정 조합과 연결됩니다. 한 함수를 VPC에 이미 연결한 경우 다른 함수를 연결할 때 동일한 서브넷과 보안 그룹을 지정하면 Lambda가 네트워크 리소스를 공유할 수 있으므로 Hyperplane ENI를 새로 생성할 필요가 없어집니다. Hyperplane ENI와 수명 주기에 대한 자세한 내용은 the section called “Hyperplane 탄력적 네트워크 인터페이스(ENIs) 이해하기” 섹션을 참조하세요.

Hyperplane 탄력적 네트워크 인터페이스(ENIs) 이해하기

Hyperplane ENI는 Lambda 함수와 함수를 연결하려는 리소스 간의 네트워크 인터페이스 역할을 하는 관리형 리소스입니다. Lambda 서비스는 함수를 VPC에 연결할 때 이러한 ENI를 자동으로 생성하고 관리합니다.

Hyperplane ENI는 직접 볼 수 없으므로 구성하거나 관리할 필요가 없습니다. 하지만 작동 방식을 알면 함수를 VPC에 연결할 때의 함수 동작을 이해하는 데 도움이 될 수 있습니다.

특정 서브넷과 보안 그룹 조합을 사용하여 함수를 VPC에 처음 연결하는 경우 Lambda가 Hyperplane ENI를 생성합니다. 동일한 서브넷과 보안 그룹 조합을 사용하는 계정의 다른 함수도 이 ENI를 사용할 수 있습니다. Lambda는 가능한 곳에서 기존 ENI를 재사용하여 리소스 활용을 최적화하고 새 ENI 생성을 최소화합니다. 각 Hyperplane ENI는 최대 65,000개의 연결/포트를 지원합니다. 연결 수가 이 한도를 초과하는 경우 Lambda는 네트워크 트래픽 및 동시성 요구 사항에 따라 ENI 수를 자동으로 조정합니다.

새 함수의 경우, Lambda가 Hyperplane ENI를 생성하는 동안에는 함수가 보류 중 상태로 유지되므로 함수를 호출할 수 없습니다. 함수는 Hyperplane ENI가 준비된 경우에만 활성 상태로 전환되며 몇 분 정도 소요될 수 있습니다. 기존 함수의 경우 버전 생성 또는 함수의 코드 업데이트 같이 함수를 대상으로 하는 추가 작업을 수행할 수는 없지만 이전 버전의 함수를 계속 호출할 수는 있습니다.

> **Note**
>
> Lambda 함수가 연속으로 30일 동안 유휴 상태로 유지되면 Lambda는 사용되지 않는 모든 Hyperplane ENI를 회수하고 함수 상태를 유휴 상태로 설정합니다. 다음 호출은 실패하며 Lambda가 Hyperplane ENI의 생성 또는 할당을 완료할 때까지 함수는 보류 중 상태로 다시 전

환됩니다. Lambda 함수 상태에 대한 자세한 내용은 the section called “함수 상태” 섹션을 참조하세요.

Hyperplane ENI 수명 주기에 대한 자세한 내용은 the section called “Lambda Hyperplane ENI” 섹션을 참조하세요.

VPC 설정에 IAM 조건 키 사용

VPC 설정에 Lambda 특정 조건 키를 사용하여 Lambda 함수에 대한 추가 권한 제어를 제공할 수 있습니다. 예를 들어 조직의 모든 함수가 VPC에 연결되도록 요구할 수 있습니다. 또한 함수의 사용자가 사용할 수 있고 사용할 수 없는 서브넷 및 보안 그룹을 지정할 수도 있습니다.

Lambda는 IAM 정책에서 다음 조건 키를 지원합니다.

- lambda:VpcIds – 하나 이상의 VPC를 허용하거나 거부합니다.
- lambda:SubnetIds – 하나 이상의 서브넷을 허용하거나 거부합니다.
- lambda:SecurityGroupIds – 하나 이상의 보안 그룹을 허용하거나 거부합니다.

Lambda API 작업 CreateFunction 및 UpdateFunctionConfiguration은 이러한 조건 키를 지원합니다. IAM 정책의 조건 키 사용 방법에 대한 자세한 내용은 IAM 사용 설명서의 IAM JSON 정책 요소: 조건을 참조하세요.

> **Tip**
>
> 함수에 이전 API 요청의 VPC 구성이 이미 포함되어 있는 경우 VPC 구성 없이 `UpdateFunctionConfiguration` 요청을 보낼 수 있습니다.

VPC 설정에 대한 조건 키가 있는 정책의 예제

다음 예제에서는 VPC 설정에 조건 키를 사용하는 방법을 보여줍니다. 원하는 제한 사항이 있는 정책 구문을 생성한 후 대상 사용자 또는 역할에 대한 정책 구문을 추가합니다.

사용자가 VPC 연결 함수만 배포하도록 보장

모든 사용자가 VPC 연결 함수만 배포하도록 보장하려면 유효한 VPC ID가 포함되지 않은 함수 생성 및 업데이트 작업을 거부할 수 있습니다.

VPC ID는 CreateFunction 또는 UpdateFunctionConfiguration 요청에 대한 입력 파라미터가 아닙니다. Lambda는 서브넷 및 보안 그룹 파라미터를 기반으로 VPC ID 값을 검색합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

특정 VPC, 서브넷 또는 보안 그룹에 대한 사용자 액세스 거부

특정 VPC에 대한 사용자의 액세스를 거부하려면 StringEquals를 사용하여 lambda:VpcIds 조건 값을 확인합니다. 다음 예제에서는 vpc-1 및 vpc-2에 대한 사용자 액세스를 거부합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceOutOfVPC",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
```

```
        }
      }
    }
```

특정 서브넷에 대한 사용자의 액세스를 거부하려면 StringEquals를 사용하여 lambda:SubnetIds 조건 값을 확인합니다. 다음 예제에서는 subnet-1 및 subnet-2에 대한 사용자 액세스를 거부합니다.

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

특정 보안 그룹에 대한 사용자의 액세스를 거부하려면 StringEquals를 사용하여 lambda:SecurityGroupIds 조건 값을 확인합니다. 다음 예제에서는 sg-1 및 sg-2에 대한 사용자 액세스를 거부합니다.

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
```

```
    }
  ]
}
```

사용자가 특정 VPC 설정을 사용하여 함수를 생성 및 업데이트하도록 허용

특정 VPC에 대한 사용자의 액세스를 허용하려면 `StringEquals`를 사용하여 `lambda:VpcIds` 조건 값을 확인합니다. 다음 예제에서는 사용자가 `vpc-1` 및 `vpc-2`에 액세스하도록 허용합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceStayInSpecificVpc",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
            "lambda:VpcIds": ["vpc-1", "vpc-2"]
        }
      }
    }
```

특정 서브넷에 대한 사용자의 액세스를 허용하려면 `StringEquals`를 사용하여 `lambda:SubnetIds` 조건 값을 확인합니다. 다음 예제에서는 사용자가 `subnet-1` 및 `subnet-2`에 액세스하도록 허용합니다.

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
```

```
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

특정 보안 그룹에 대한 사용자의 액세스를 허용하려면 `StringEquals`를 사용하여 `lambda:SecurityGroupIds` 조건 값을 확인합니다. 다음 예제에서는 사용자가 `sg-1` 및 `sg-2`에 액세스하도록 허용합니다.

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC 자습서

다음 자습서에서는 Lambda 함수를 VPC의 리소스에 연결합니다.

- 자습서: Amazon VPC에서 Amazon RDS에 액세스하도록 Lambda 함수 사용
- 자습서: Amazon VPC에서 Amazon ElastiCache에 액세스하도록 Lambda 함수 구성

VPC 연결 Lambda 함수에 대한 인터넷 액세스 활성화

기본적으로 Lambda 함수는 인터넷에 액세스할 수 있는 Lambda 관리형 VPC에서 실행됩니다. 계정의 VPC에 있는 리소스에 액세스하려면 함수에 VPC 구성을 추가하면 됩니다. 이렇게 하면 VPC가 인터넷에 액세스할 수 없는 경우에는 해당 VPC 내의 리소스로 함수가 제한됩니다. 이 페이지에서는 VPC 연결 Lambda 함수에 인터넷 액세스를 제공하는 방법을 설명합니다.

아직 VPC가 없습니다

VPC 생성

VPC 생성 워크플로는 서브넷, NAT 게이트웨이, 인터넷 게이트웨이, 라우팅 테이블 항목을 포함하여 Lambda 함수가 프라이빗 서브넷에서 퍼블릭 인터넷에 액세스하는 데 필요한 모든 VPC 리소스를 생성합니다.

VPC를 생성하려면

1. https://console.aws.amazon.com/vpc/에서 Amazon VPC 콘솔을 엽니다.
2. 대시보드에서 VPC 생성을 선택합니다.
3. 생성할 리소스에서 VPC 등을 선택합니다.
4. VPC 구성
   a. Name tag auto-generation(이름 태그 자동 생성)에 VPC의 이름을 입력합니다.
   b. IPv4 CIDR 블록에 애플리케이션 또는 네트워크에 필요한 CIDR 블록을 입력하거나 기본 사항을 유지할 수 있습니다.
   c. 애플리케이션이 IPv6 주소를 사용하여 통신하는 경우 IPv6 CIDR 블록, 즉 Amazon에서 제공한 IPv6 CIDR 블록을 선택합니다.
5. 서브넷 구성
   a. 가용 영역 수에서 2를 선택합니다. 고가용성을 위해 2개 이상의 AZ를 권장합니다.
   b. 퍼블릭 서브넷 수는 2를 선택합니다.
   c. 프라이빗 서브넷 수는 2를 선택합니다.
   d. 퍼블릭 서브넷에 대한 기본 CIDR 블록을 유지하거나 서브넷 CIDR 블록 사용자 지정을 확장하고 CIDR 블록을 입력할 수 있습니다. 자세한 내용은 서브넷 CIDR 블록을 참조하세요.
6. NAT 게이트웨이에서 복원력 향상을 위해 AZ당 1개를 선택합니다.
7. 송신 전용 인터넷 게이트웨이의 경우 IPv6 CIDR 블록을 포함하도록 선택한 경우 예를 선택합니다.

8. VPC 엔드포인트의 경우 기본값(S3 게이트웨이)을 유지합니다. 이 옵션은 비용이 들지 않습니다. 자세한 내용은 Amazon S3용 VPC 엔드포인트 유형을 참조하세요.
9. DNS 옵션의 경우 기본 설정을 유지합니다.
10. VPC 생성을 선택합니다.

Lambda 함수 구성

함수를 생성할 때 VPC를 구성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 생성을 선택합니다.
3. 기본 정보(Basic information)에서 함수 이름(Function name)에 함수 이름을 입력합니다.
4. Advanced settings(고급 설정)를 확장합니다.
5. VPC 활성화를 선택한 다음 VPC를 선택합니다.
6. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
7. 서브넷의 경우 모든 프라이빗 서브넷을 선택합니다. 프라이빗 서브넷은 NAT 게이트웨이를 통해 인터넷에 액세스할 수 있습니다. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스가 제공되지는 않습니다.

   > **Note**
   >
   > 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

8. 보안 그룹에서 아웃바운드 트래픽을 허용하는 보안 그룹을 선택합니다.
9. 함수 생성을 선택합니다.

Lambda는 AWSLambdaVPCAccessExecutionRole AWS 관리형 정책을 사용하여 실행 역할을 자동으로 생성합니다. 이 정책의 권한은 VPC 구성을 위한 탄력적 네트워크 인터페이스를 생성하는 데만 필요하며, 함수를 호출하는 데는 필요하지 않습니다. 최소 권한의 권한을 적용하려면 함수 및 VPC 구성을 생성한 후 실행 역할에서 AWSLambdaVPCAccessExecutionRole 정책을 제거하면 됩니다. 자세한 내용은 필수 IAM 권한 단원을 참조하십시오.

기존 함수에 대한 VPC를 구성하려면

기존 함수에 VPC 구성을 추가하려면 함수의 실행 역할에 탄력적 네트워크 인터페이스를 생성하고 관리할 수 있는 권한이 있어야 합니다. AWSLambdaVPCAccessExecutionRole AWS 관리형 정책에는 필요한 권한이 포함되어 있습니다. 최소 권한의 권한을 적용하려면 VPC 구성을 생성한 후 실행 역할에서 AWSLambdaVPCAccessExecutionRole 정책을 제거하면 됩니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택한 다음 VPC를 선택합니다.
4. VPC에서 편집을 선택합니다.
5. VPC를 선택합니다.
6. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
7. 서브넷의 경우 모든 프라이빗 서브넷을 선택합니다. 프라이빗 서브넷은 NAT 게이트웨이를 통해 인터넷에 액세스할 수 있습니다. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스가 제공되지는 않습니다.

   > **Note**
   >
   > 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

8. 보안 그룹에서 아웃바운드 트래픽을 허용하는 보안 그룹을 선택합니다.
9. Save(저장)를 선택합니다.

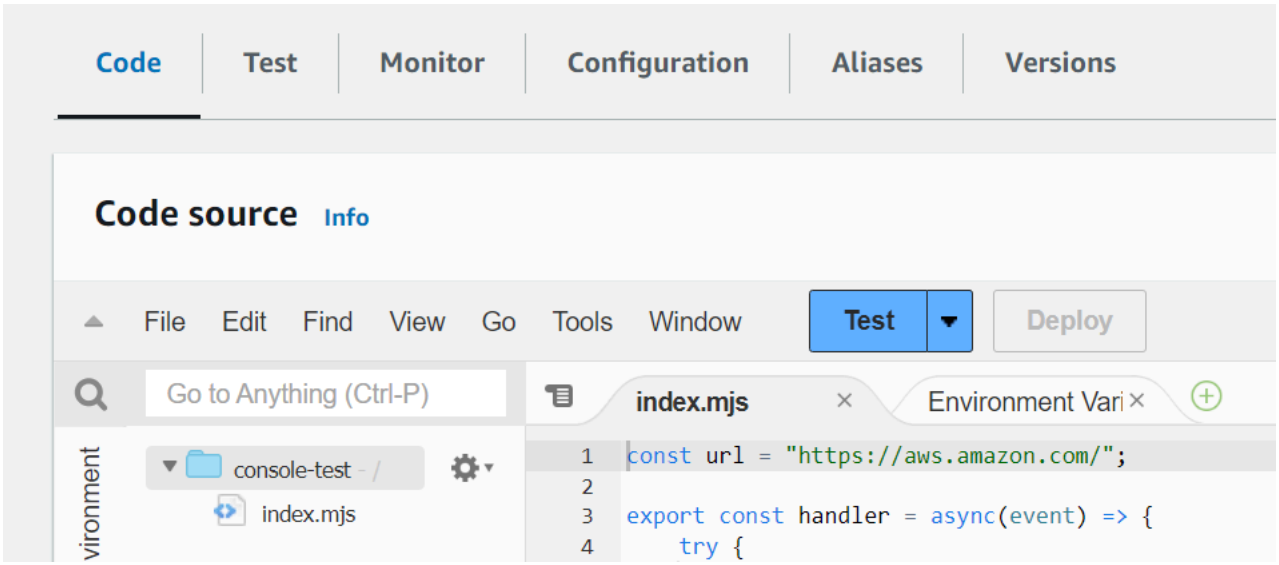
함수 테스트

다음 샘플 코드를 사용하여 VPC 연결 함수가 퍼블릭 인터넷에 연결할 수 있는지 확인하세요. 성공하면 코드가 `200` 상태 코드를 반환합니다. 실패하면 함수가 제한 시간을 초과합니다.

Node.js

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 `fetch`를 사용합니다.

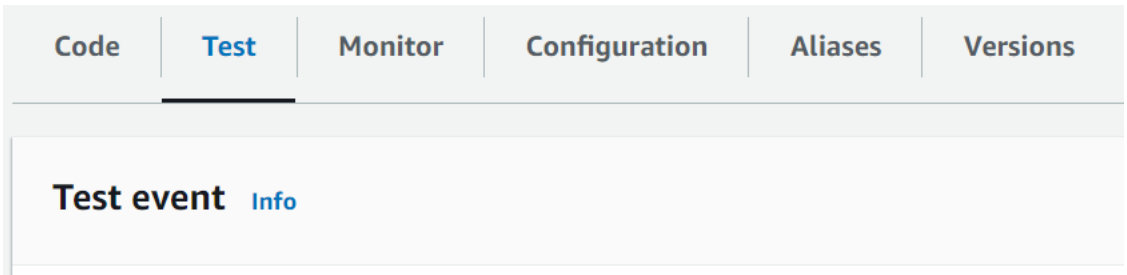
1. Lambda 콘솔의 코드 소스 창에서 index.mjs 파일에 다음 코드를 붙여넣습니다. 이 함수는 퍼블릭 엔드포인트에 HTTP GET 요청을 하고 HTTP 응답 코드를 반환하여 함수가 퍼블릭 인터넷에 액세스할 수 있는지 테스트합니다.



Example - async/await를 사용한 HTTP 요청

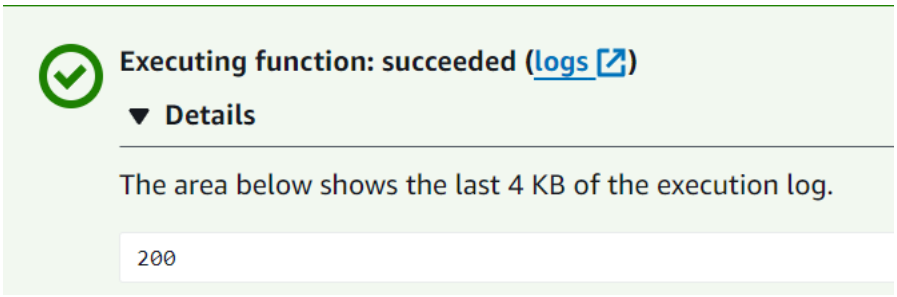
```
const url = "https://aws.amazon.com/";

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. [배포]를 선택합니다.
3. 테스트 탭을 선택합니다.



4. 테스트를 선택합니다.

5. 함수가 200 상태 코드를 반환합니다. 이는 해당 함수에 아웃바운드 인터넷 액세스 권한이 있음을 의미합니다.

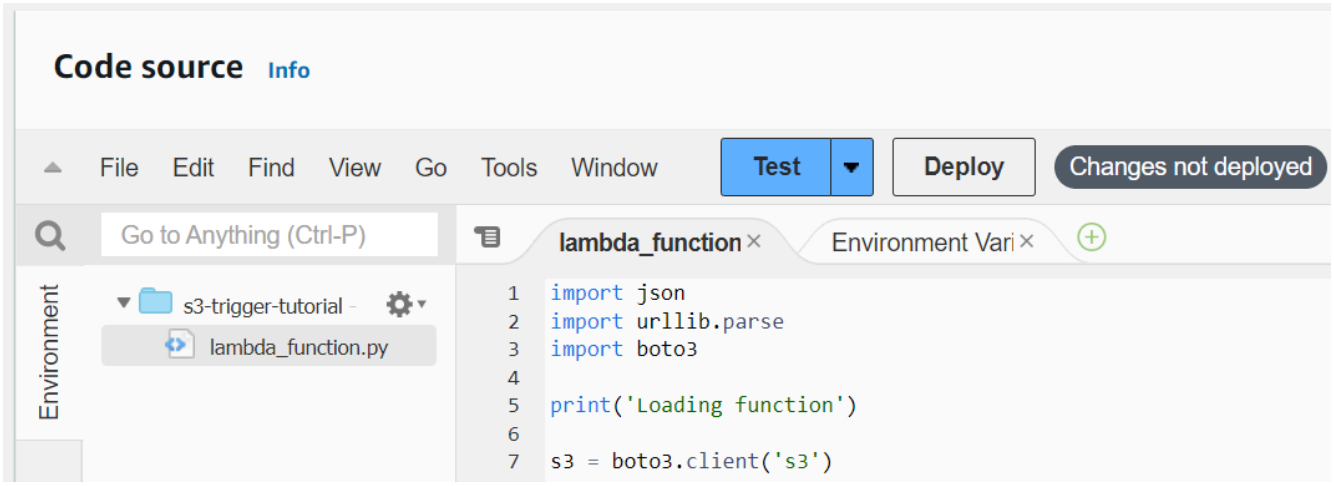


함수가 퍼블릭 인터넷에 연결할 수 없으면 다음과 같은 오류 메시지가 표시됩니다.

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

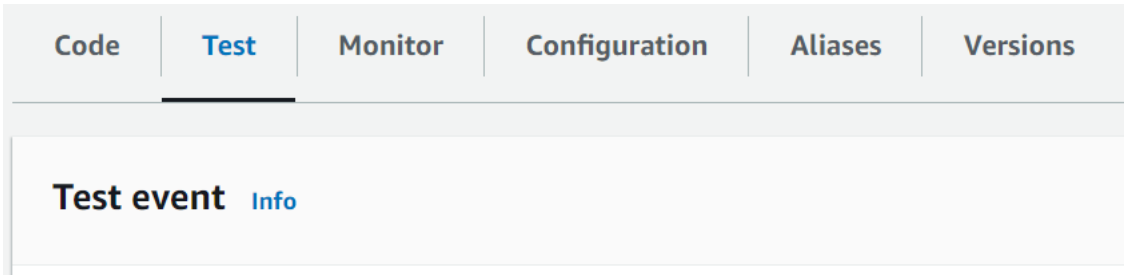
1. Lambda 콘솔의 코드 소스 창에서 lambda_function.py 파일에 다음 코드를 붙여넣습니다. 이 함수는 퍼블릭 엔드포인트에 HTTP GET 요청을 하고 HTTP 응답 코드를 반환하여 함수가 퍼블릭 인터넷에 액세스할 수 있는지 테스트합니다.



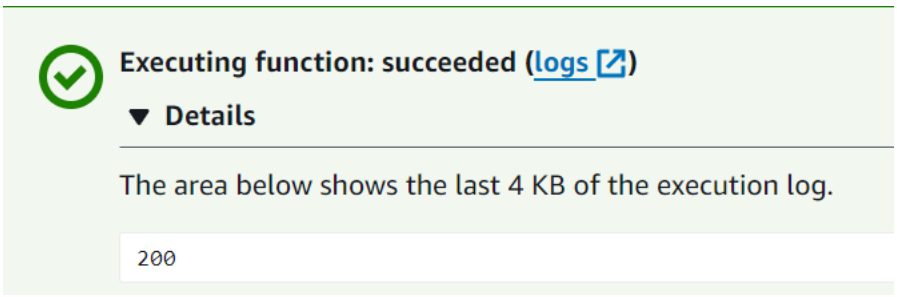

```
import urllib.request

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. [배포]를 선택합니다.
3. 테스트 탭을 선택합니다.



4. 테스트를 선택합니다.
5. 함수가 200 상태 코드를 반환합니다. 이는 해당 함수에 아웃바운드 인터넷 액세스 권한이 있음을 의미합니다.



함수가 퍼블릭 인터넷에 연결할 수 없으면 다음과 같은 오류 메시지가 표시됩니다.

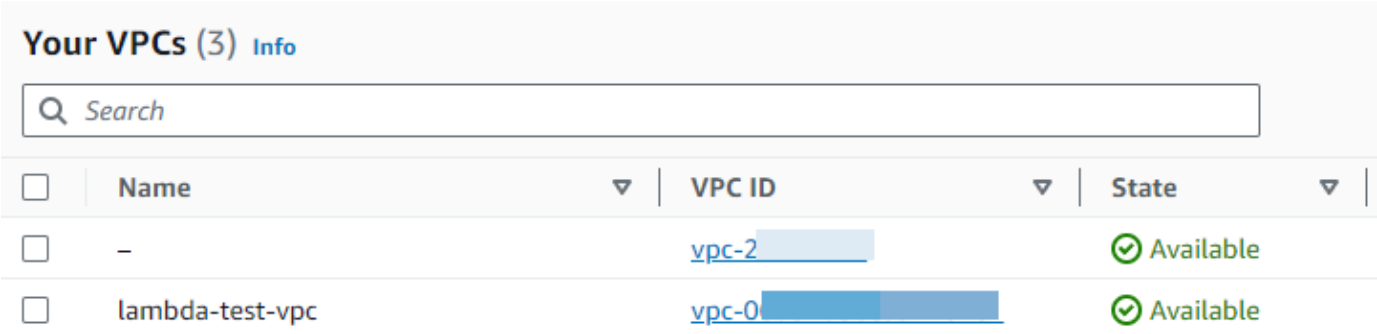
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

이미 VPC가 있습니다

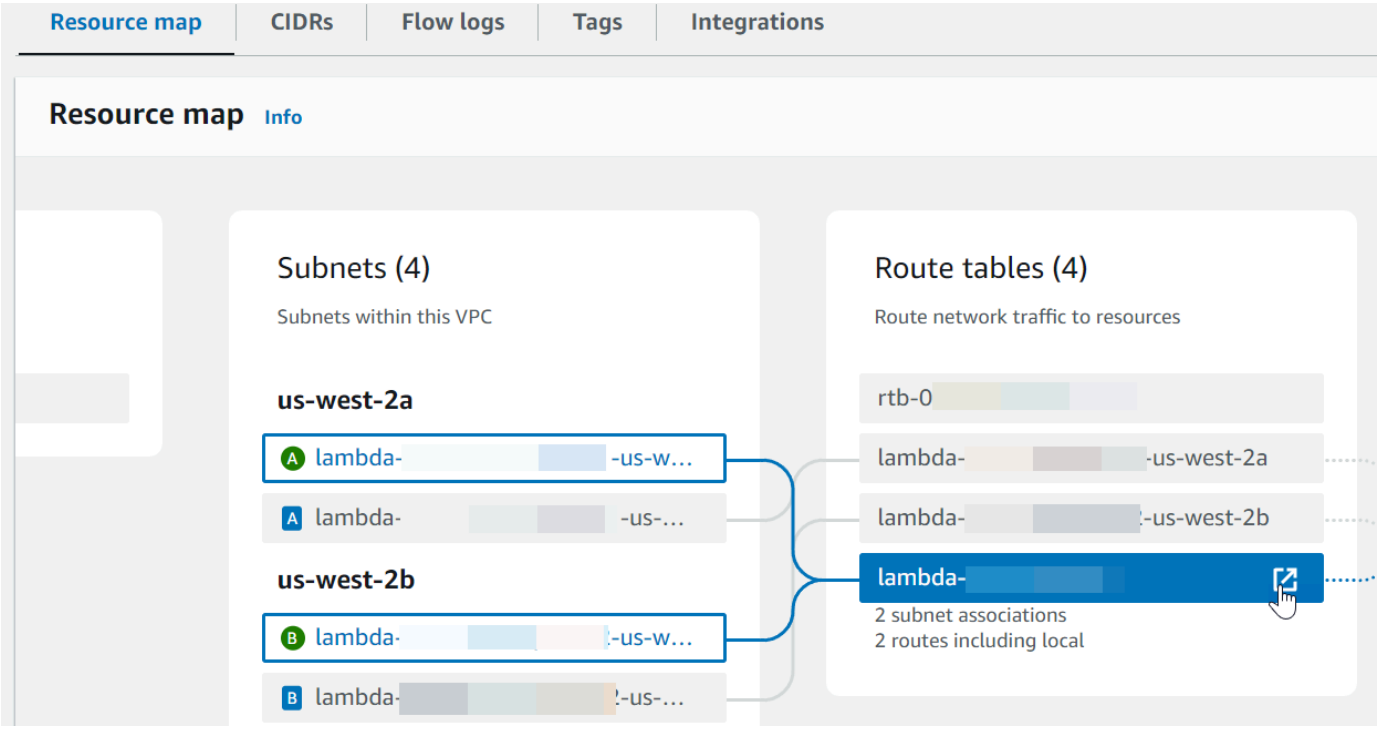
이미 VPC가 있지만 Lambda 함수에 대한 퍼블릭 인터넷 액세스를 구성해야 하는 경우 다음 단계를 따르세요. 이 절차에서는 VPC에 서브넷이 2개 이상 있다고 가정합니다. 서브넷 2개가 없는 경우 Amazon VPC 사용 설명서의 서브넷 생성을 참조하세요.

라우팅 테이블 구성 확인

1. https://console.aws.amazon.com/vpc/에서 Amazon VPC 콘솔을 엽니다.
2. VPC ID를 선택합니다.



3. 아래로 스크롤하여 리소스 맵 섹션으로 이동합니다. 라우팅 테이블 매핑에 유의하세요. 서브넷에 매핑된 각각의 라우팅 테이블을 엽니다.

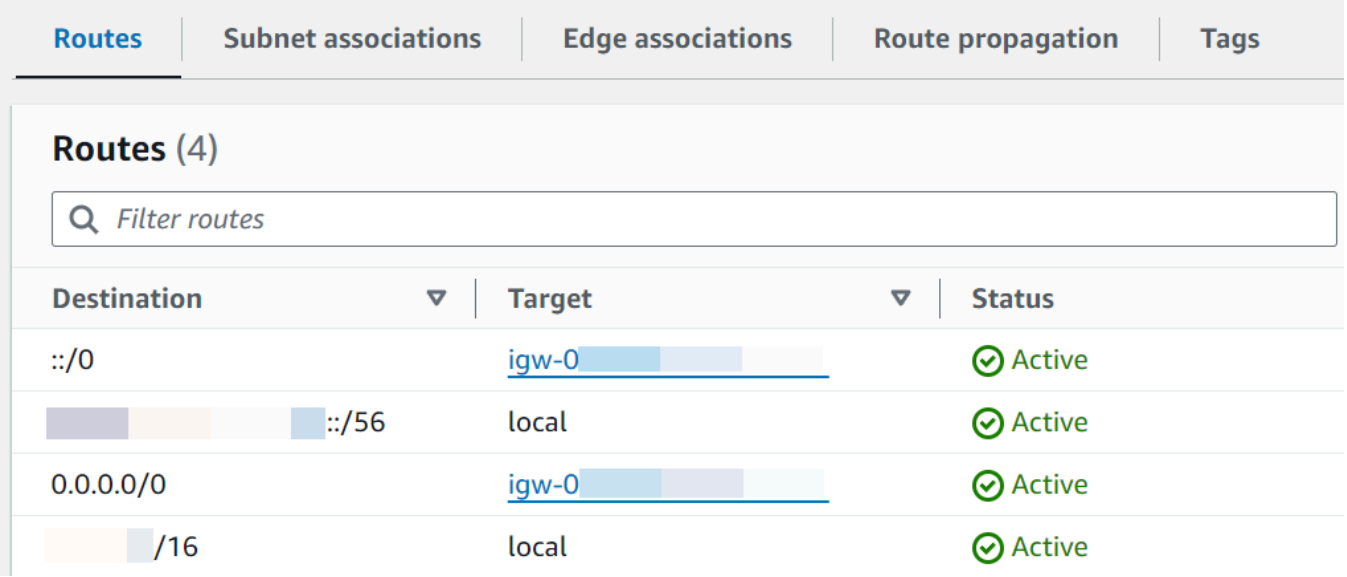


4. 아래로 스크롤하여 라우팅 탭으로 이동합니다. 라우팅을 검토하여 다음 중 하나에 해당하는지 확인합니다. 이러한 각 요구 사항은 별도의 라우팅 테이블을 통해 충족되어야 합니다.

   - 인터넷 바운드 트래픽(IPv4의 경우 `0.0.0.0/0`, IPv6의 경우 `::/0`)은 인터넷 게이트웨이(`igw-xxxxxxxxxx`)로 라우팅됩니다. 이는 라우팅 테이블과 연결된 서브넷이 퍼블릭 서브넷임을 의미합니다.

> **Note**
>
> 서브넷에 IPv6 CIDR 블록이 없는 경우 IPv4 라우팅(`0.0.0.0/0`)만 표시됩니다.

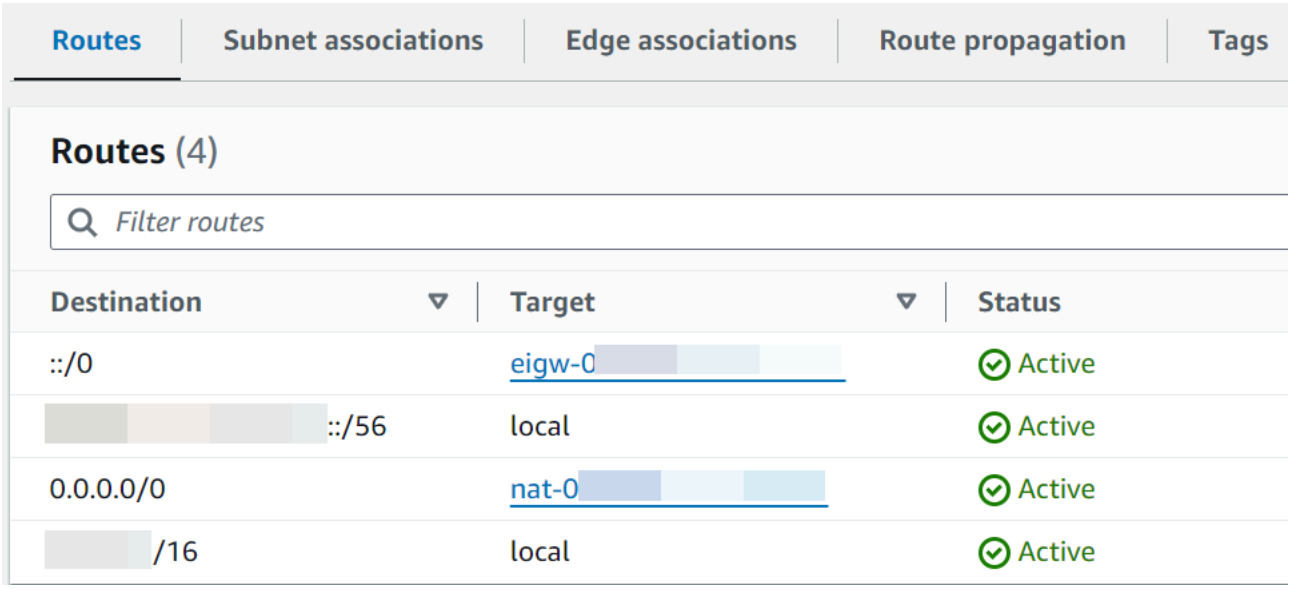
Example 퍼블릭 서브넷 라우팅 테이블



- IPv4(`0.0.0.0/0`)에 대한 인터넷에 바인딩된 트래픽은 퍼블릭 서브넷과 연결된 NAT 게이트웨이(`nat-xxxxxxxxxx`)로 라우팅됩니다. 이는 해당 서브넷이 NAT 게이트웨이를 통해 인터넷에 액세스할 수 있는 프라이빗 서브넷임을 의미합니다.

> **Note**
>
> 서브넷에 IPv6 CIDR 블록이 있는 경우 라우팅 테이블은 인터넷에 바인딩된 IPv6 트래픽(`::/0`)도 송신 전용 인터넷 게이트웨이(`eigw-xxxxxxxxxx`)로 라우팅해야 합니다. 서브넷에 IPv6 CIDR 블록이 없는 경우 IPv4 라우팅(`0.0.0.0/0`)만 표시됩니다.

Example 프라이빗 서브넷 라우팅 테이블



5. VPC의 서브넷과 연결된 각 라우팅 테이블을 검토하고 인터넷 게이트웨이가 있는 라우팅 테이블과 NAT 게이트웨이가 있는 라우팅 테이블이 있음을 확인할 때까지 이전 단계를 반복합니다.

   라우팅 테이블 2개(인터넷 게이트웨이에 대한 라우팅이 있는 라우팅 테이블과 NAT 게이트웨이에 대한 라우팅이 있는 라우팅 테이블)가 없는 경우 다음 단계에 따라 누락된 리소스와 라우팅 테이블 항목을 생성합니다.

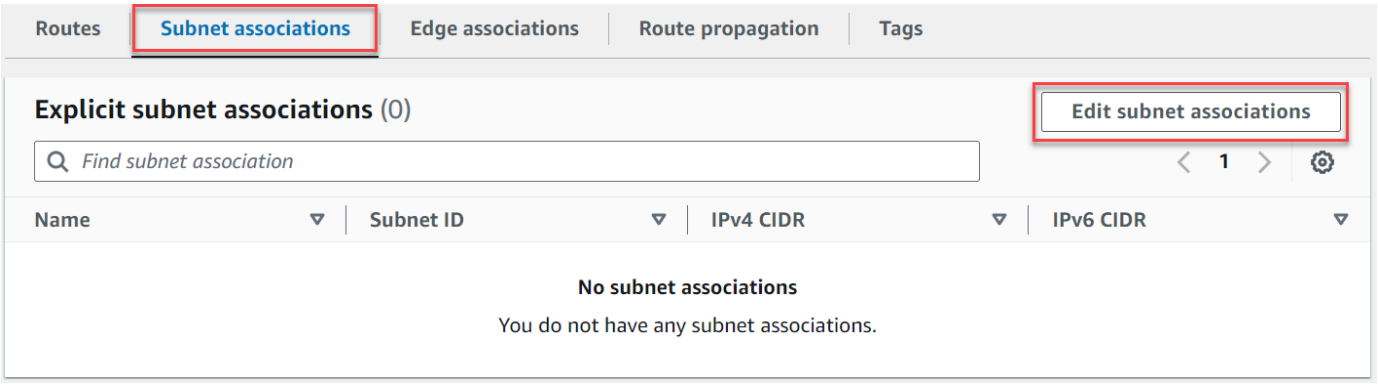
라우팅 테이블 생성

라우팅 테이블을 생성하고 이를 서브넷과 연결하려면 다음 단계를 따르세요.

Amazon VPC 콘솔을 사용하여 사용자 지정 라우팅 테이블을 생성하려면 다음을 수행합니다.

1. https://console.aws.amazon.com/vpc/에서 Amazon VPC 콘솔을 엽니다.
2. 탐색 창에서 라우팅 테이블을 선택합니다.
3. 라우팅 테이블 생성을 선택합니다.
4. (선택 사항) 이름(Name)에 라우팅 테이블의 이름을 입력합니다.
5. VPC에서 VPC를 선택합니다.
6. (선택 사항) 태그를 추가하려면 새 태그 추가(Add new tag)를 선택하고 태그 키와 태그 값을 입력합니다.
7. 라우팅 테이블 생성을 선택합니다.

8. [서브넷 연결(Subnet associations)] 탭에서 [서브넷 연결 편집(Edit subnet associations)]을 선택합니다.



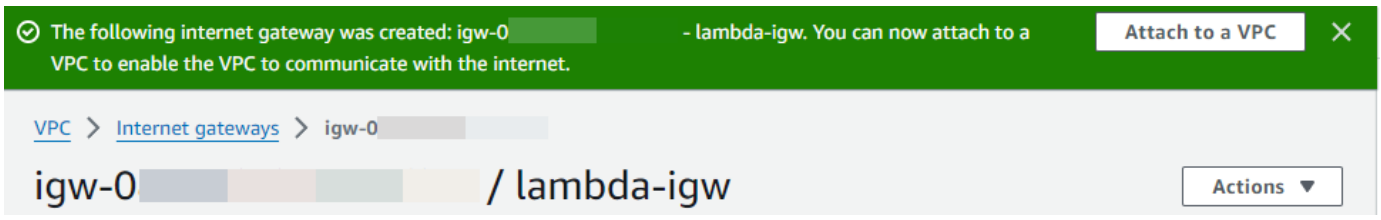
9. 라우팅 테이블과 연결할 서브넷에 대한 확인란을 선택합니다.
10. [연결 저장(Save associations)]을 선택합니다.

인터넷 게이트웨이 생성

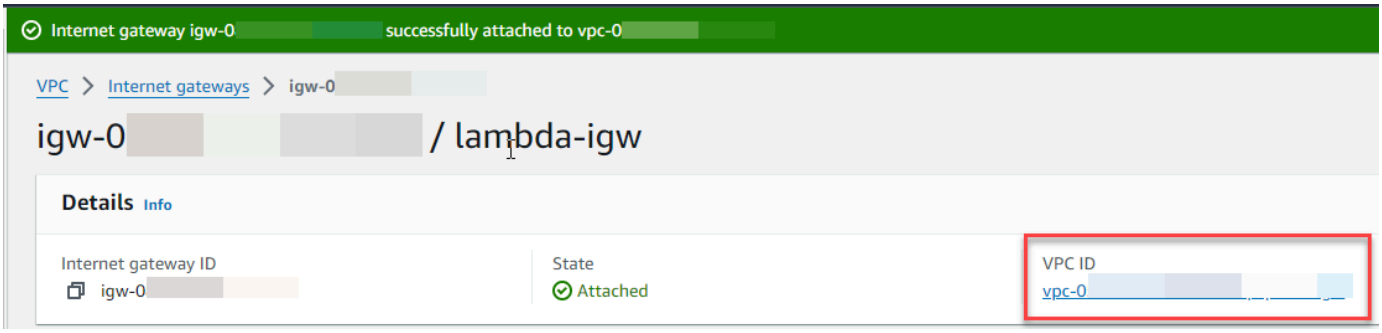
다음 단계에 따라 인터넷 게이트웨이를 생성하고, 이를 VPC에 연결하고, 퍼블릭 서브넷의 라우팅 테이블에 추가합니다.

인터넷 게이트웨이를 생성하려면

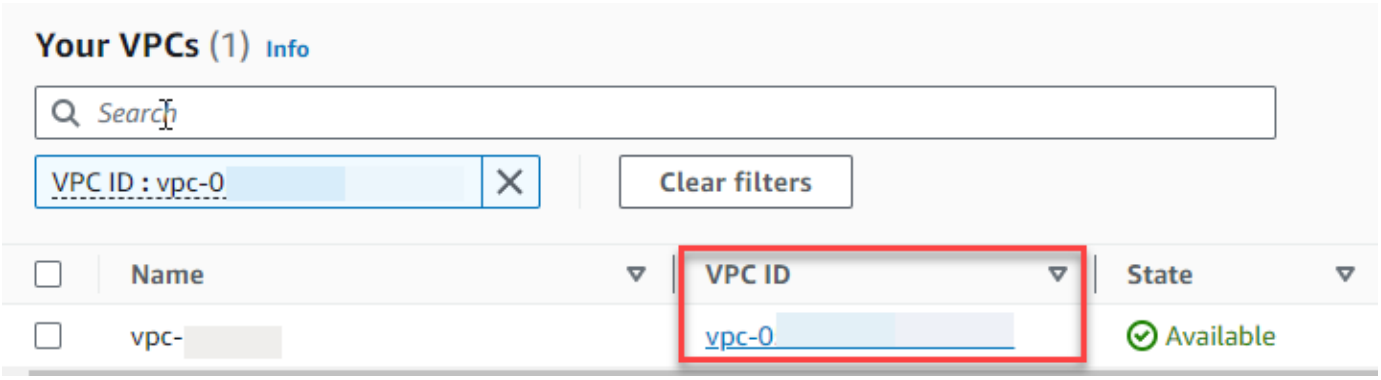
1. https://console.aws.amazon.com/vpc/에서 Amazon VPC 콘솔을 엽니다.
2. 탐색 창에서 인터넷 게이트웨이(Internet gateways)를 선택합니다.
3. 인터넷 게이트웨이 생성을 선택합니다.
4. (선택 사항) 인터넷 게이트웨이에 이름을 입력합니다.
5. (선택 사항) 태그를 추가하려면 Add new tag(새 태그 추가)를 선택하고 태그 키와 태그 값을 입력합니다.
6. 인터넷 게이트웨이 생성을 선택합니다.
7. 화면 상단의 배너에서 VPC에 연결을 선택하고 사용 가능한 VPC를 선택한 다음 인터넷 게이트웨이 연결을 선택합니다.



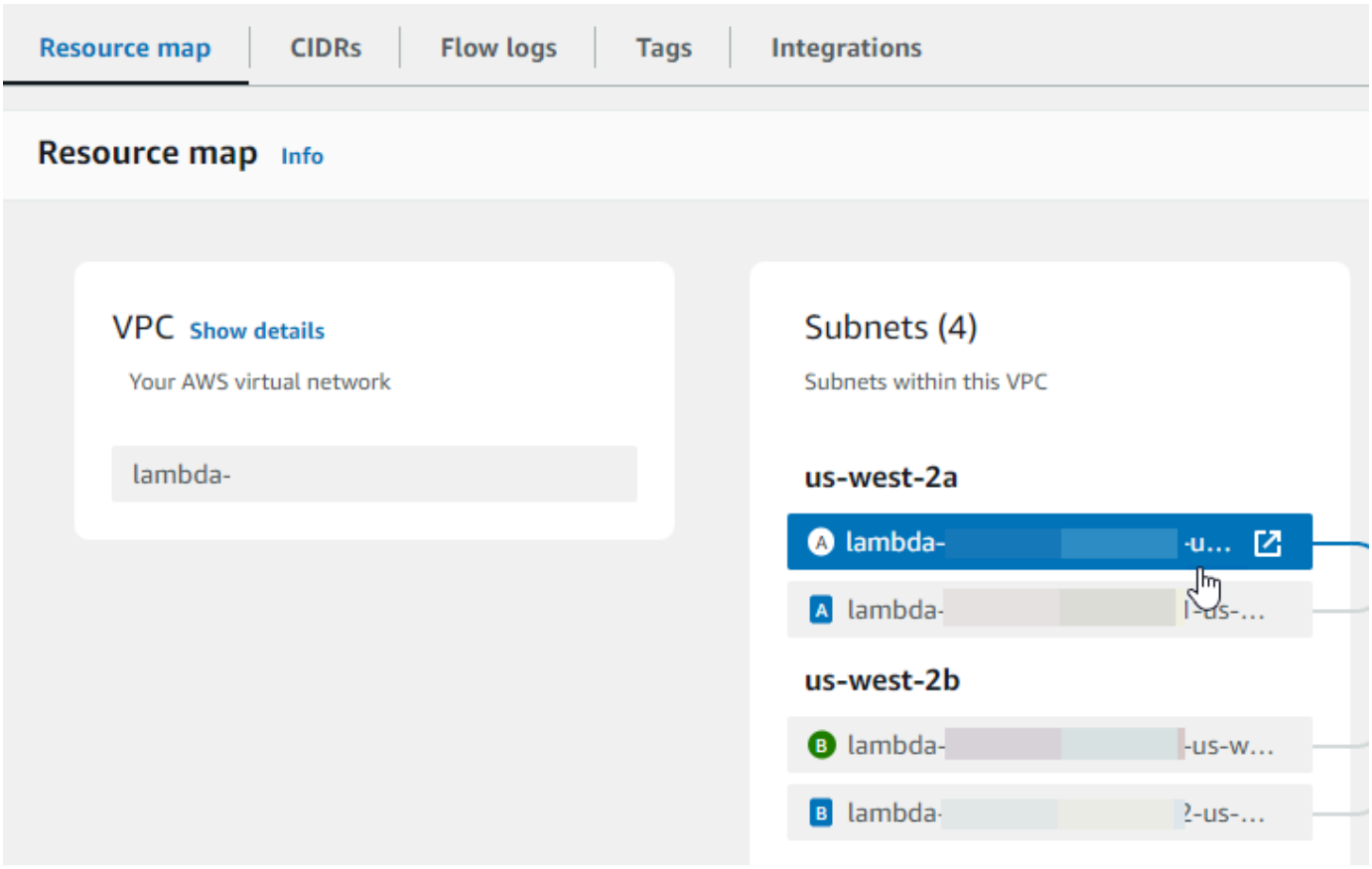
8. VPC ID를 선택합니다.



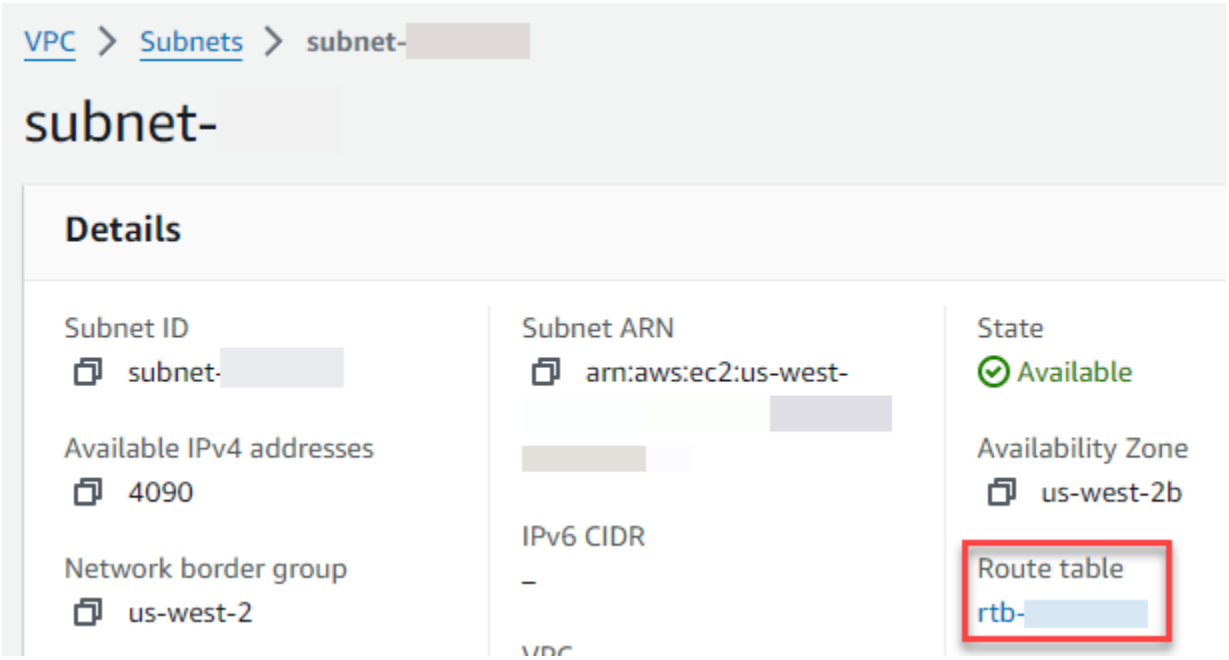
9. VPC ID를 다시 선택하여 VPC 세부 정보 페이지를 엽니다.



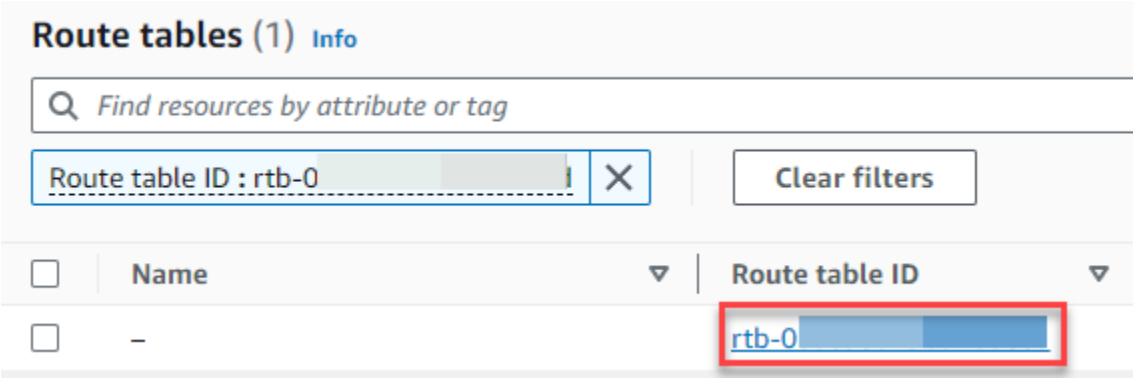
10. 아래로 스크롤하여 리소스 맵으로 이동한 다음 서브넷을 선택합니다. 서브넷 세부 정보가 새 탭에 표시됩니다.



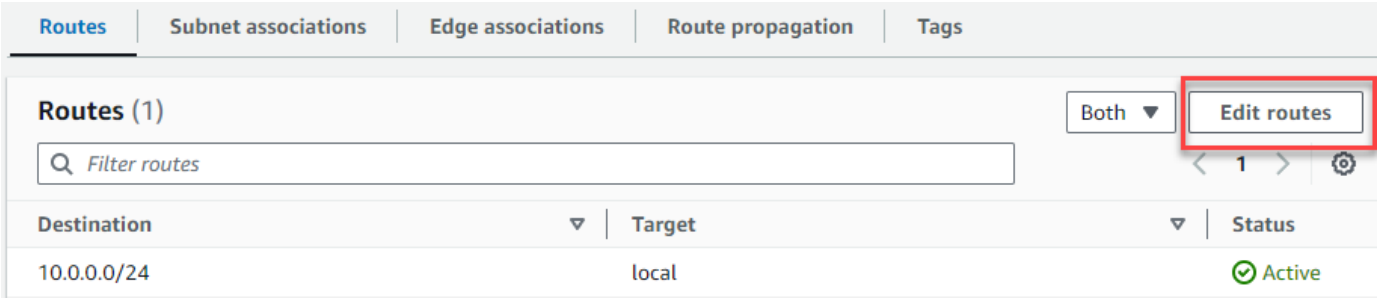
11. 라우팅 테이블에서 링크를 선택합니다.



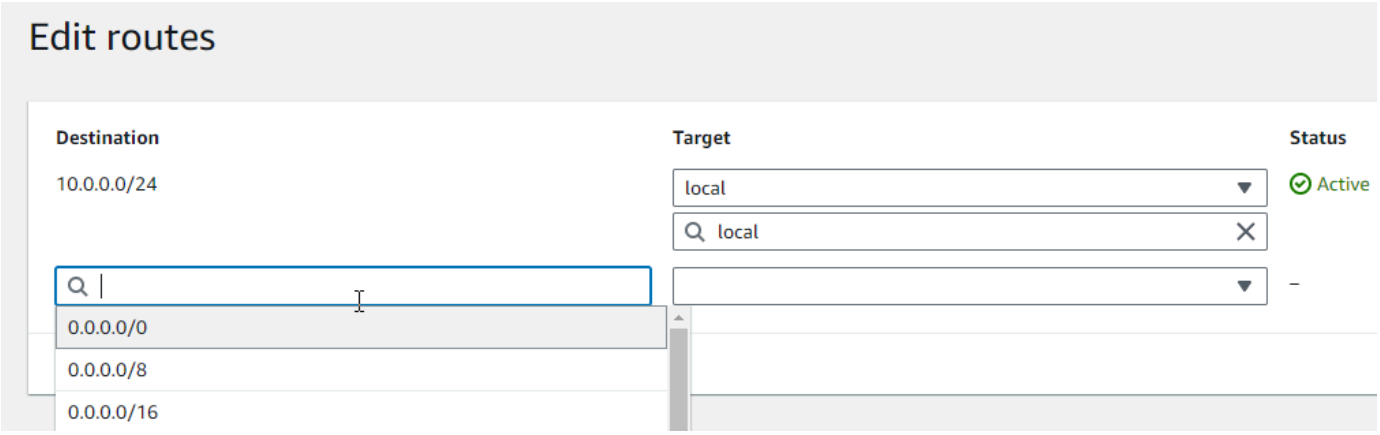
12. 라우팅 테이블 ID를 선택하여 라우팅 테이블 세부 정보 페이지를 엽니다.



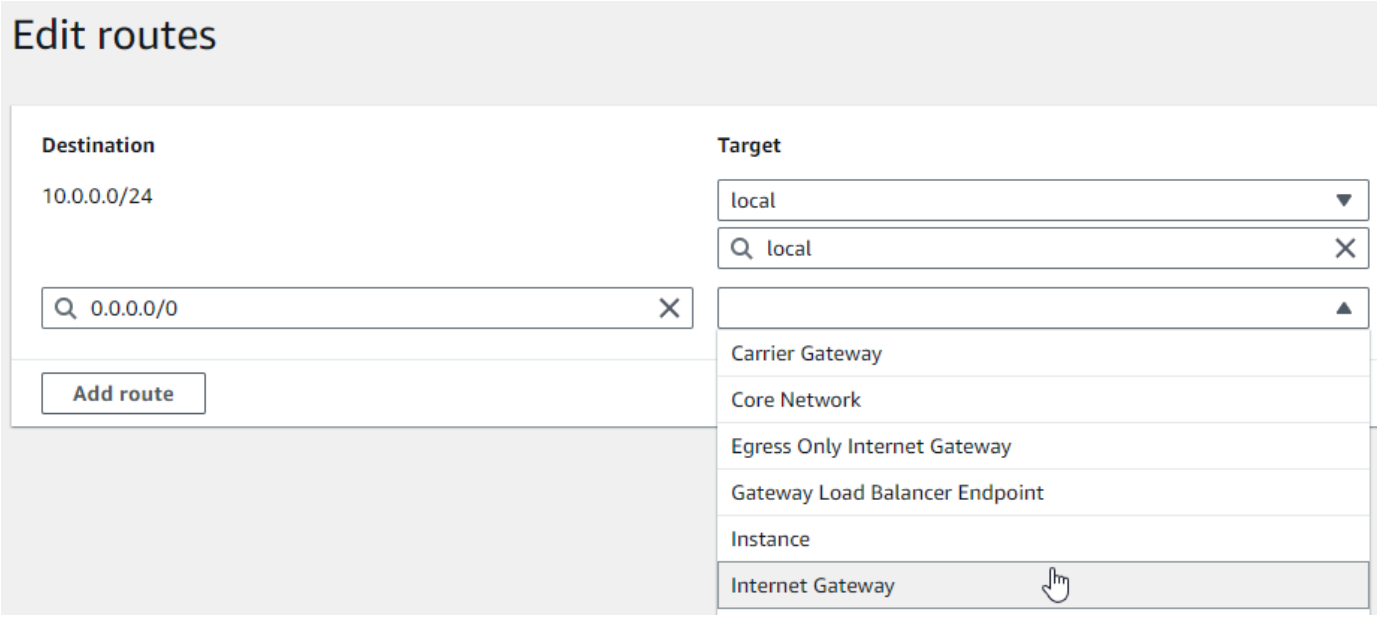
13. 라우팅에서 라우팅 편집을 선택합니다.



14. 라우팅 추가를 선택한 다음 대상 상자에 `0.0.0.0/0`을 입력합니다.



15. 대상에서 인터넷 게이트웨이를 선택한 다음 이전에 생성한 인터넷 게이트웨이를 선택합니다. 서브넷에 IPv6 CIDR 블록이 있는 경우 동일한 인터넷 게이트웨이에 `::/0`에 대한 라우팅도 추가해야 합니다.



16. Save changes(변경 사항 저장)를 선택합니다.

NAT 게이트웨이 만들기

다음 단계에 따라 NAT 게이트웨이를 생성하고 이를 퍼블릭 서브넷과 연결한 다음 프라이빗 서브넷의 라우팅 테이블에 추가합니다.

NAT 게이트웨이를 생성하고 이를 퍼블릭 서브넷과 연결하려면 다음을 수행합니다.

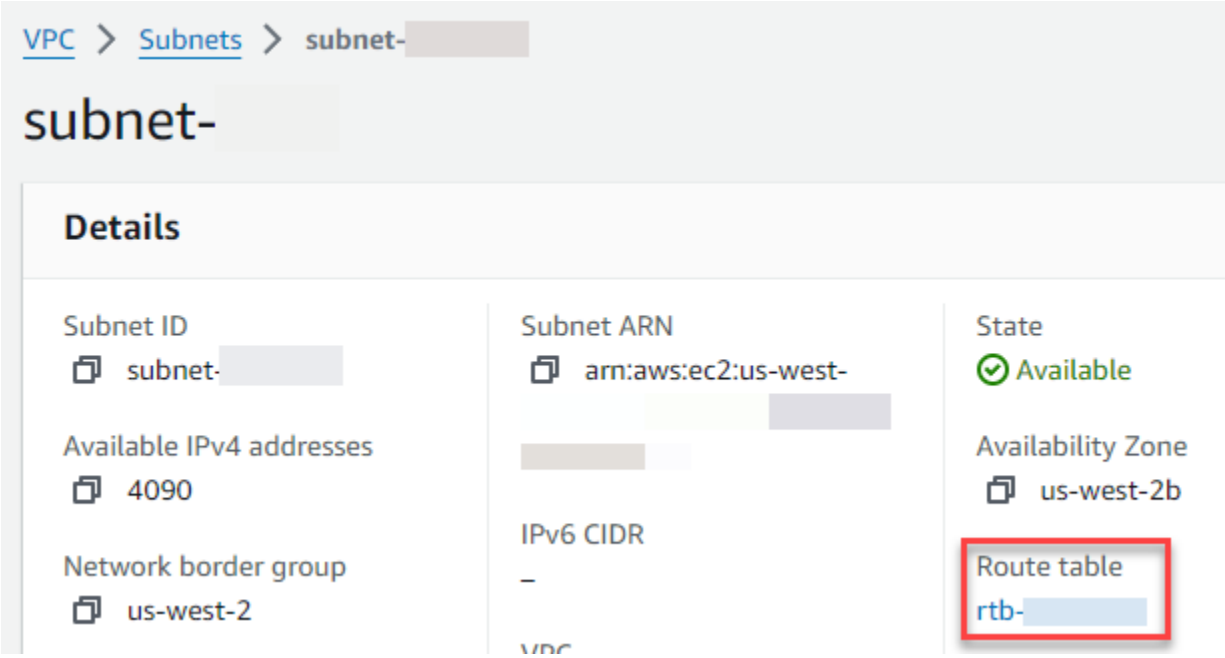
1. 탐색 창에서 NAT 게이트웨이를 선택합니다.
2. NAT 게이트웨이 생성을 선택합니다.
3. (선택 사항) NAT 게이트웨이에 이름을 입력합니다.
4. 서브넷에서 VPC의 퍼블릭 서브넷을 선택합니다. (퍼블릭 서브넷은 라우팅 테이블에 인터넷 게이트웨이에 대한 직접 라우팅이 있는 서브넷입니다.)

> **Note**
>
> NAT 게이트웨이는 퍼블릭 서브넷과 연결되어 있지만 라우팅 테이블 항목은 프라이빗 서브넷에 있습니다.

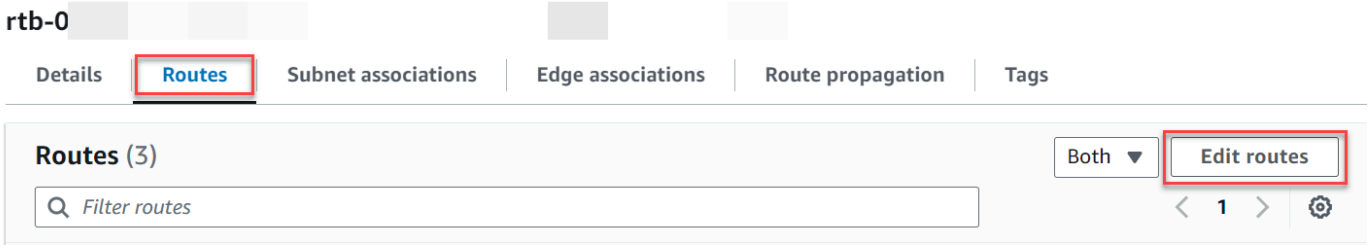
5. 탄력적 IP 할당 ID에서 탄력적 IP 주소를 선택하거나 탄력적 IP 할당을 선택합니다.
6. NAT 게이트웨이 생성을 선택합니다.

프라이빗 서브넷의 라우팅 테이블에 있는 NAT 게이트웨이에 라우팅을 추가하려면 다음을 수행합니다.

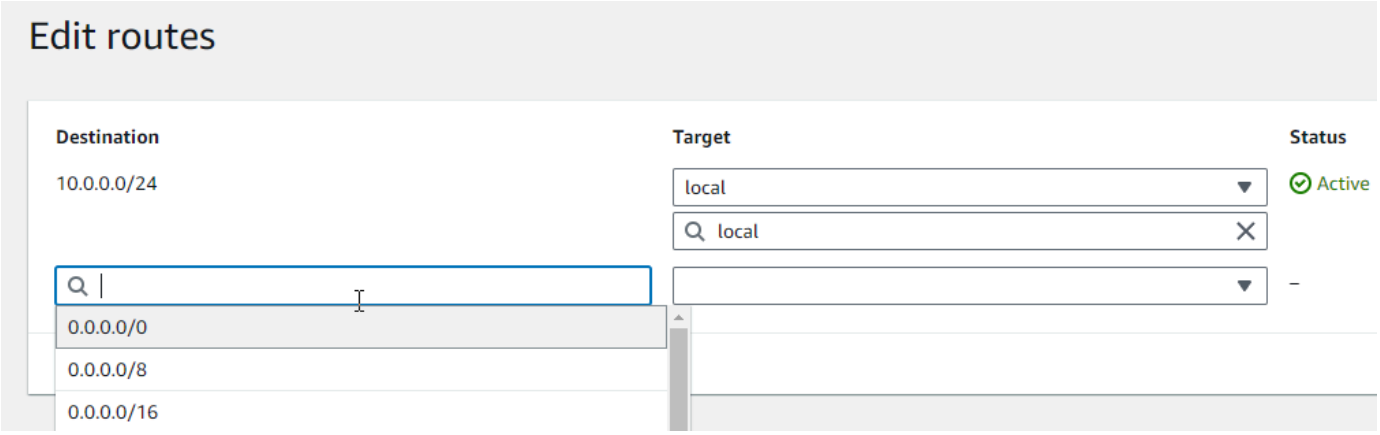
1. 탐색 창에서 서브넷을 선택합니다.
2. VPC에서 프라이빗 서브넷을 선택합니다. (프라이빗 서브넷은 라우팅 테이블에 인터넷 게이트웨이에 대한 라우팅이 없는 서브넷입니다.)
3. 라우팅 테이블에서 링크를 선택합니다.



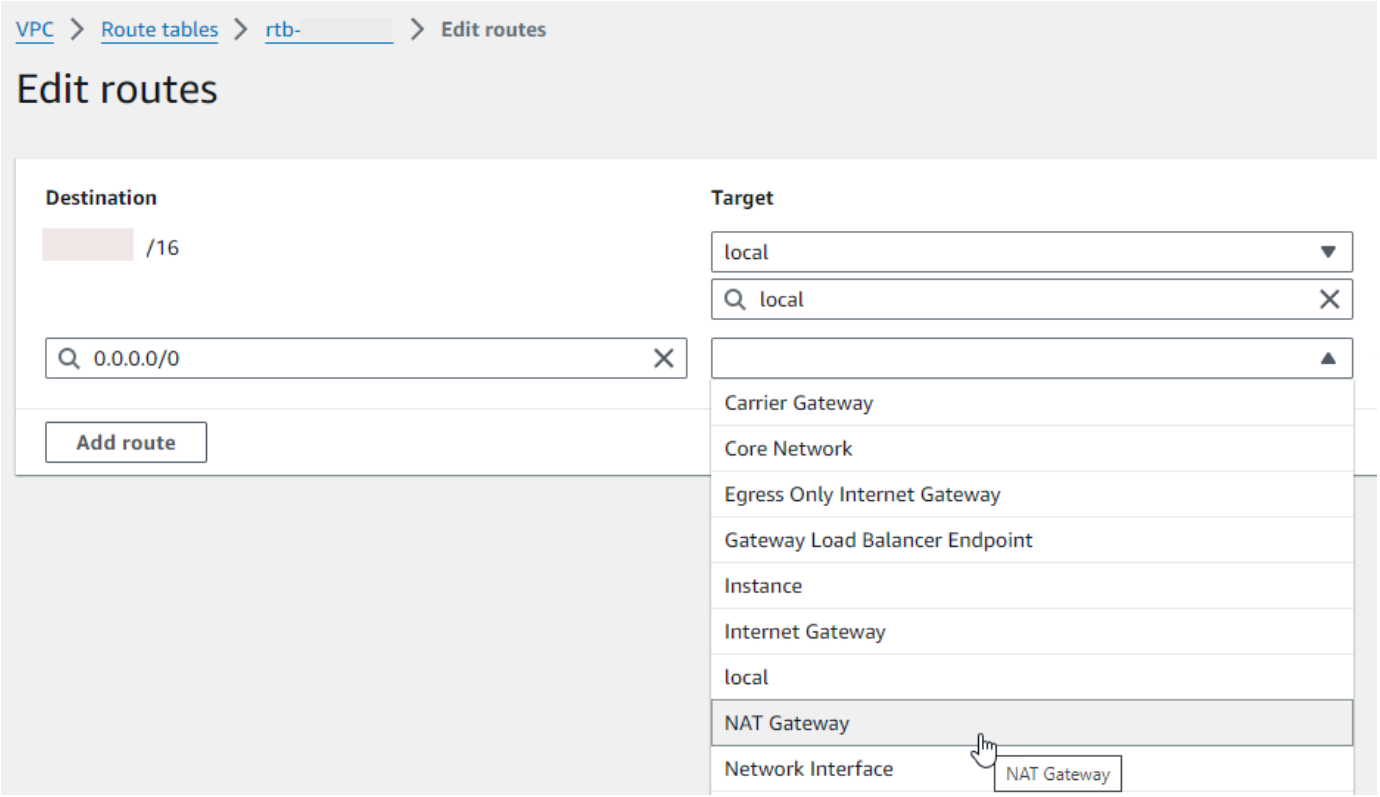
4. 아래로 스크롤하여 라우팅 탭을 선택한 다음 라우팅 편집을 선택합니다.



5. 라우팅 추가를 선택한 다음 대상 상자에 `0.0.0.0/0`을 입력합니다.



6. 대상에서 NAT 게이트웨이를 선택한 다음 이전에 생성한 NAT 게이트웨이를 선택합니다.



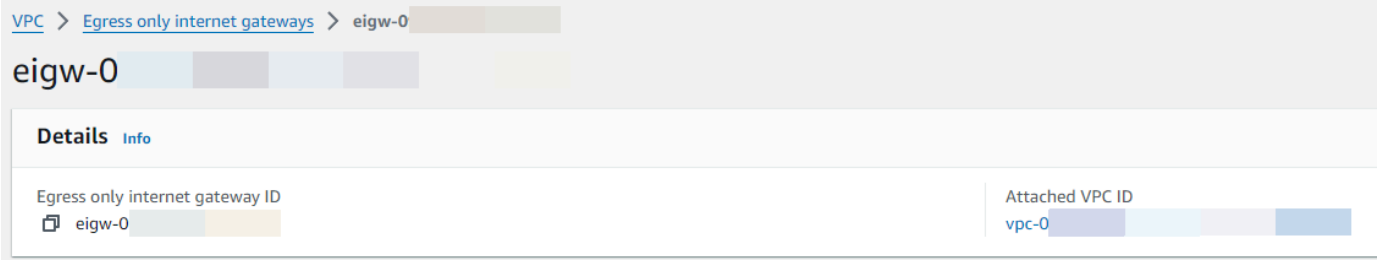
7. Save changes(변경 사항 저장)를 선택합니다.

송신 전용 인터넷 게이트웨이 생성(IPv6 전용)

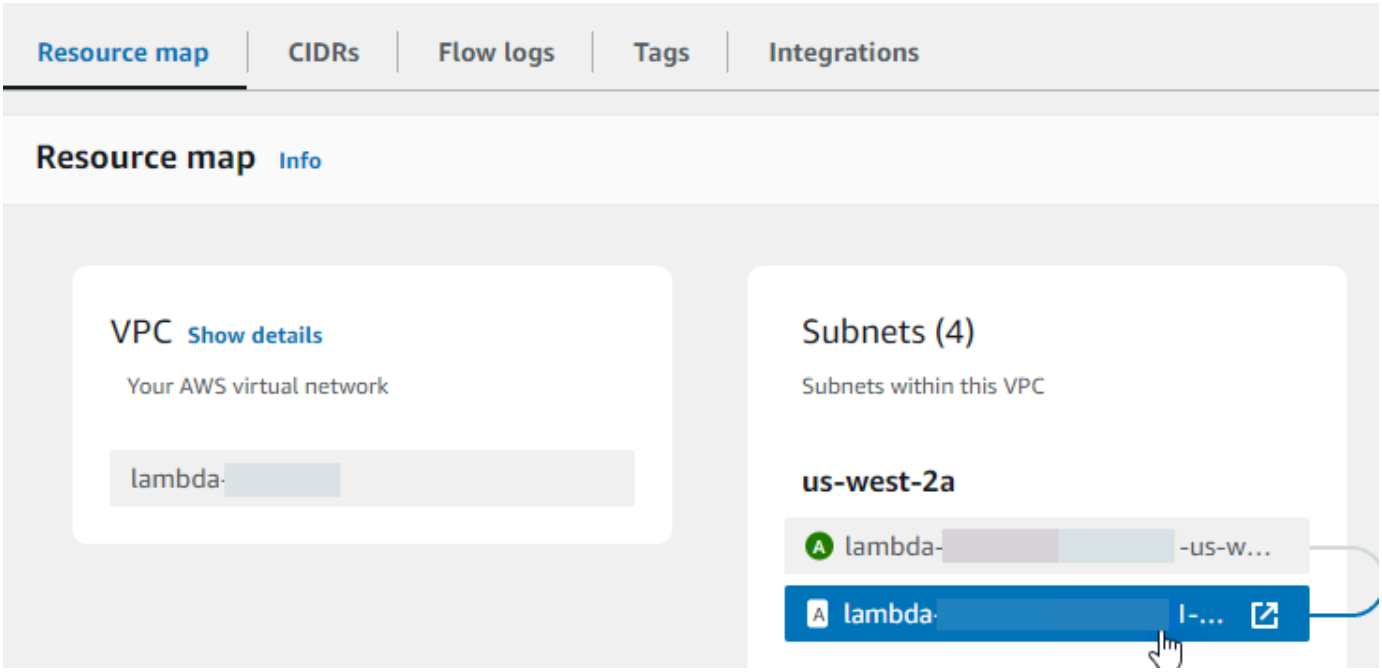
송신 전용 인터넷 게이트웨이를 생성하고 프라이빗 서브넷의 라우팅 테이블에 추가하려면 다음 단계를 따르세요.

외부 전용 인터넷 게이트웨이를 생성하려면

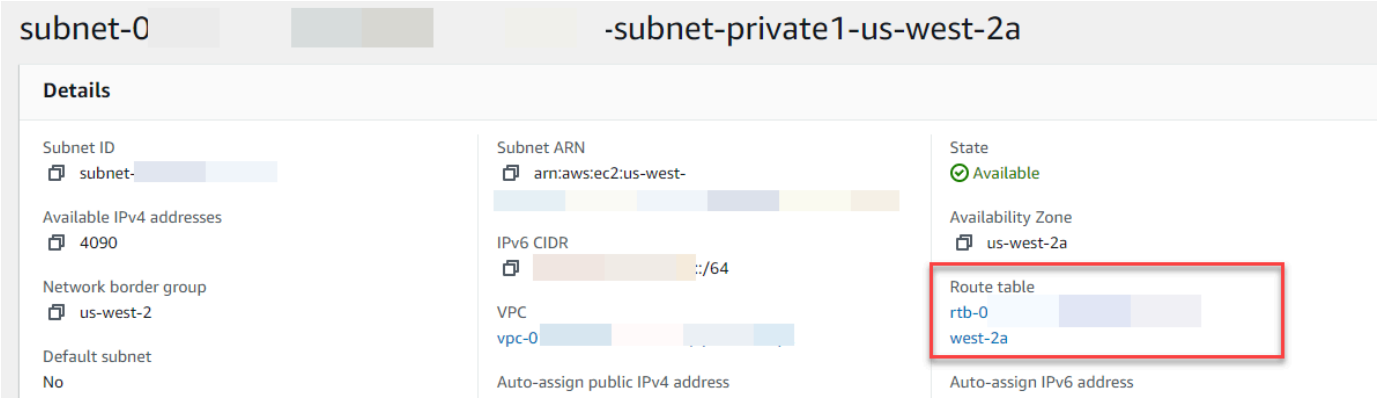
1. 탐색 창에서 송신 전용 인터넷 게이트웨이를 선택합니다.
2. 송신 전용 인터넷 게이트웨이 생성을 선택합니다.
3. (선택 사항) 이름을 입력합니다.
4. 외부 전용 인터넷 게이트웨이를 생성할 VPC를 선택합니다.
5. 송신 전용 인터넷 게이트웨이 생성을 선택합니다.
6. 연결된 VPC ID에서 링크를 선택합니다.



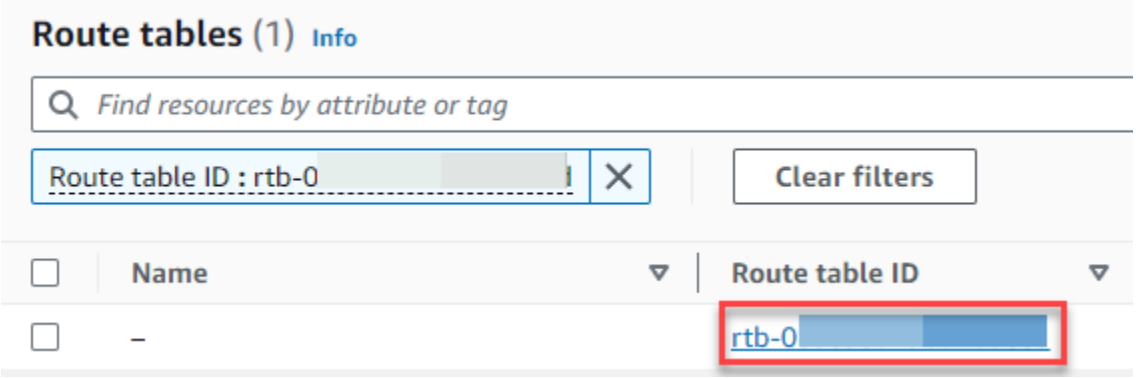
7. VPC ID 아래의 링크를 선택하여 VPC 세부 정보 페이지를 엽니다.
8. 아래로 스크롤하여 리소스 맵으로 이동한 다음 프라이빗 서브넷을 선택합니다. 서브넷 세부 정보가 새 탭에 표시됩니다.



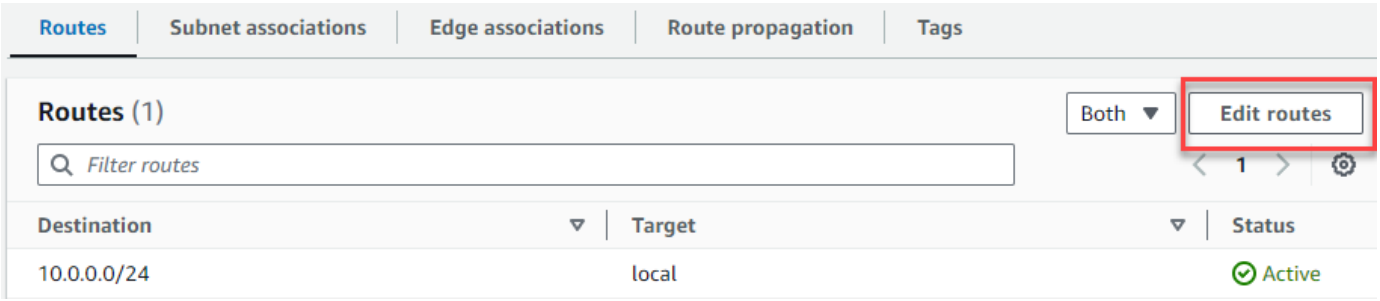
9. 라우팅 테이블에서 링크를 선택합니다.



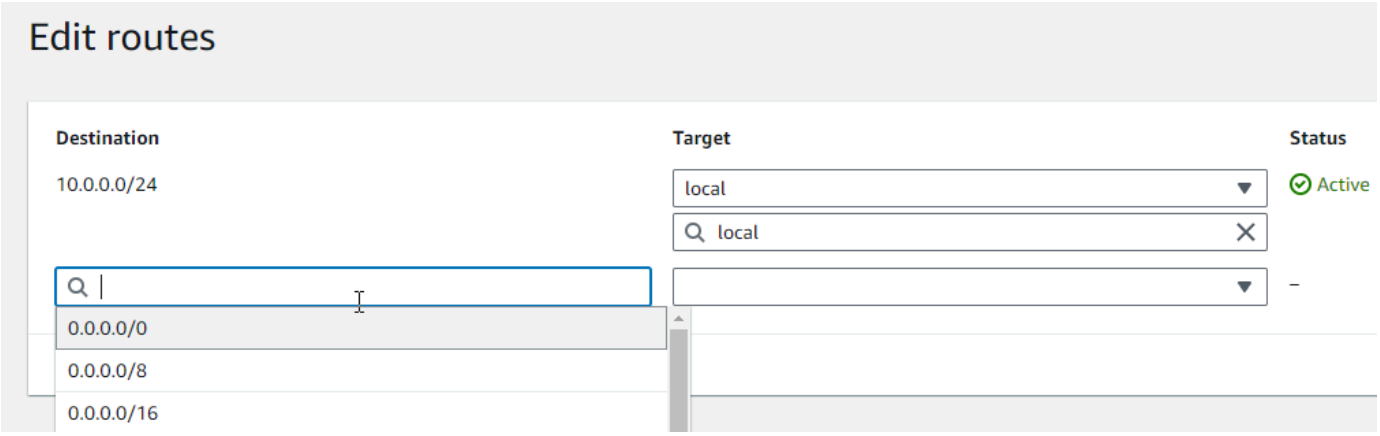
10. 라우팅 테이블 ID를 선택하여 라우팅 테이블 세부 정보 페이지를 엽니다.



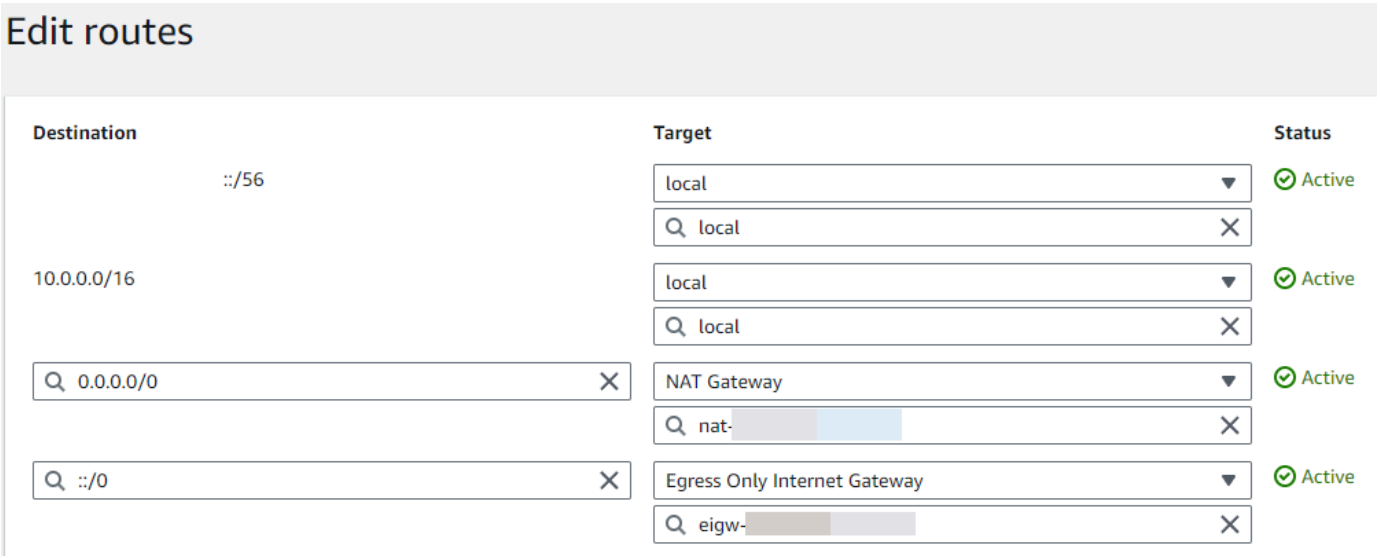
11. 라우팅에서 라우팅 편집을 선택합니다.



12. 라우팅 추가를 선택한 다음 대상 상자에 `::/0`을 입력합니다.



13. 대상에서 송신 전용 인터넷 게이트웨이를 선택한 다음 이전에 생성한 게이트웨이를 선택합니다.



14. Save changes(변경 사항 저장)를 선택합니다.

Lambda 함수 구성

함수를 생성할 때 VPC를 구성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 생성을 선택합니다.
3. 기본 정보(Basic information)에서 함수 이름(Function name)에 함수 이름을 입력합니다.
4. Advanced settings(고급 설정)를 확장합니다.
5. VPC 활성화를 선택한 다음 VPC를 선택합니다.
6. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
7. 서브넷의 경우 모든 프라이빗 서브넷을 선택합니다. 프라이빗 서브넷은 NAT 게이트웨이를 통해 인터넷에 액세스할 수 있습니다. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스가 제공되지는 않습니다.

> **Note**
>
> 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

8. 보안 그룹에서 아웃바운드 트래픽을 허용하는 보안 그룹을 선택합니다.

9. 함수 생성을 선택합니다.

Lambda는 AWSLambdaVPCAccessExecutionRole AWS 관리형 정책을 사용하여 실행 역할을 자동으로 생성합니다. 이 정책의 권한은 VPC 구성을 위한 탄력적 네트워크 인터페이스를 생성하는 데만 필요하며, 함수를 호출하는 데는 필요하지 않습니다. 최소 권한의 권한을 적용하려면 함수 및 VPC 구성을 생성한 후 실행 역할에서 AWSLambdaVPCAccessExecutionRole 정책을 제거하면 됩니다. 자세한 내용은 필수 IAM 권한 단원을 참조하십시오.

기존 함수에 대한 VPC를 구성하려면

기존 함수에 VPC 구성을 추가하려면 함수의 실행 역할에 탄력적 네트워크 인터페이스를 생성하고 관리할 수 있는 권한이 있어야 합니다. AWSLambdaVPCAccessExecutionRole AWS 관리형 정책에는 필요한 권한이 포함되어 있습니다. 최소 권한의 권한을 적용하려면 VPC 구성을 생성한 후 실행 역할에서 AWSLambdaVPCAccessExecutionRole 정책을 제거하면 됩니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택한 다음 VPC를 선택합니다.
4. VPC에서 편집을 선택합니다.
5. VPC를 선택합니다.
6. (선택 사항) 아웃바운드 IPv6 트래픽을 허용하려면 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택합니다.
7. 서브넷의 경우 모든 프라이빗 서브넷을 선택합니다. 프라이빗 서브넷은 NAT 게이트웨이를 통해 인터넷에 액세스할 수 있습니다. 함수를 퍼블릭 서브넷에 연결해도 인터넷 액세스가 제공되지는 않습니다.

   > **Note**
   >
   > 듀얼 스택 서브넷에 IPv6 트래픽 허용을 선택한 경우 선택한 모든 서브넷에 IPv4 CIDR 블록 및 IPv6 CIDR 블록이 있어야 합니다.

8. 보안 그룹에서 아웃바운드 트래픽을 허용하는 보안 그룹을 선택합니다.
9. Save(저장)를 선택합니다.

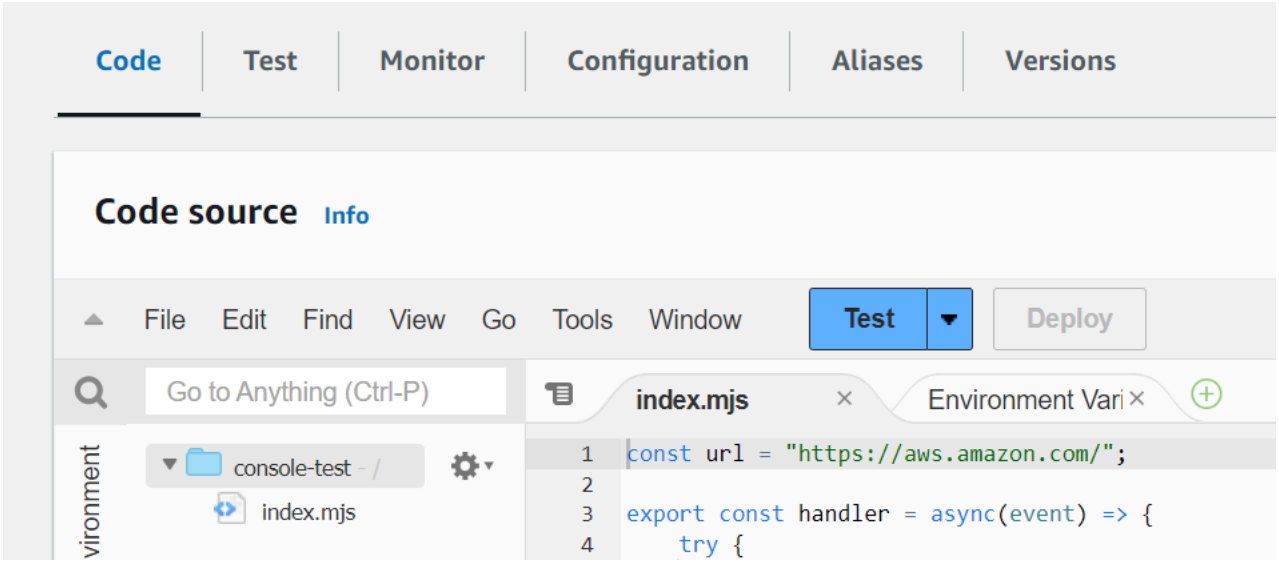
함수 테스트

다음 샘플 코드를 사용하여 VPC 연결 함수가 퍼블릭 인터넷에 연결할 수 있는지 확인하세요. 성공하면 코드가 200 상태 코드를 반환합니다. 실패하면 함수가 제한 시간을 초과합니다.

Node.js

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 `fetch`를 사용합니다.

1. Lambda 콘솔의 코드 소스 창에서 index.mjs 파일에 다음 코드를 붙여넣습니다. 이 함수는 퍼블릭 엔드포인트에 HTTP GET 요청을 하고 HTTP 응답 코드를 반환하여 함수가 퍼블릭 인터넷에 액세스할 수 있는지 테스트합니다.

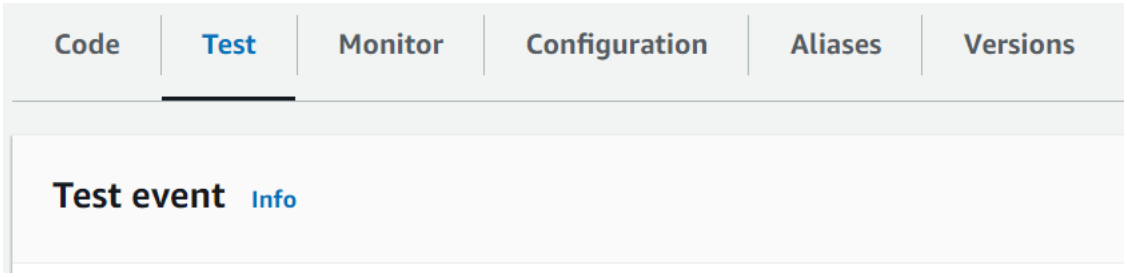


Example - async/await를 사용한 HTTP 요청

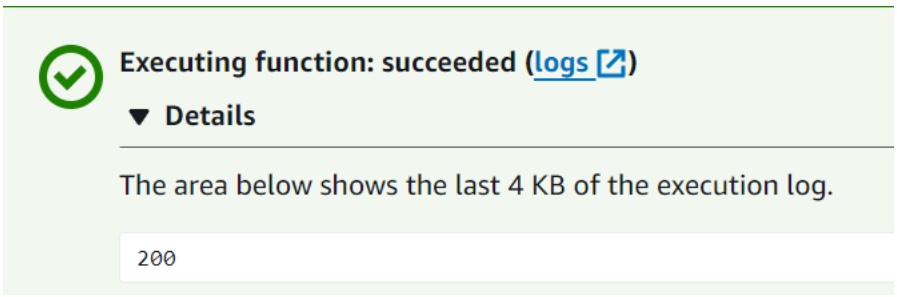
```
const url = "https://aws.amazon.com/";

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. [배포]를 선택합니다.
3. 테스트 탭을 선택합니다.



4. 테스트를 선택합니다.
5. 함수가 200 상태 코드를 반환합니다. 이는 해당 함수에 아웃바운드 인터넷 액세스 권한이 있음을 의미합니다.

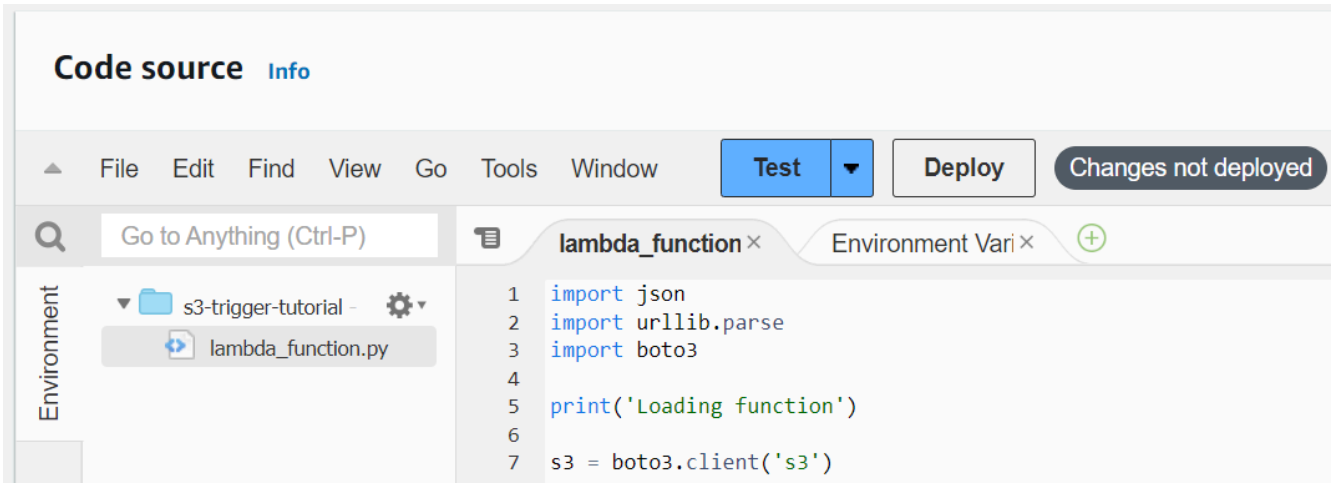


함수가 퍼블릭 인터넷에 연결할 수 없으면 다음과 같은 오류 메시지가 표시됩니다.

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

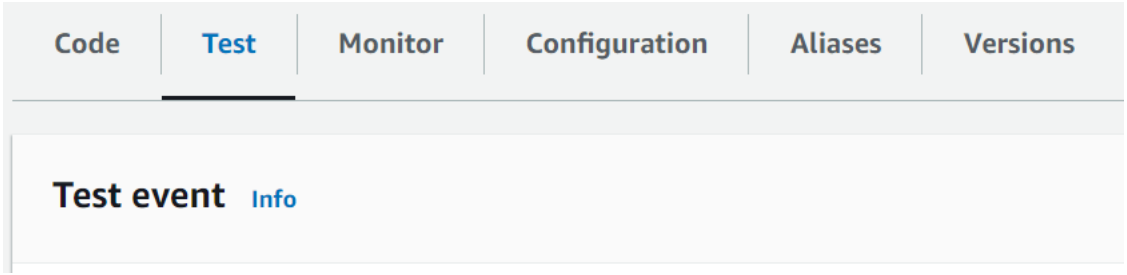
Python

1. Lambda 콘솔의 코드 소스 창에서 lambda_function.py 파일에 다음 코드를 붙여넣습니다. 이 함수는 퍼블릭 엔드포인트에 HTTP GET 요청을 하고 HTTP 응답 코드를 반환하여 함수가 퍼블릭 인터넷에 액세스할 수 있는지 테스트합니다.

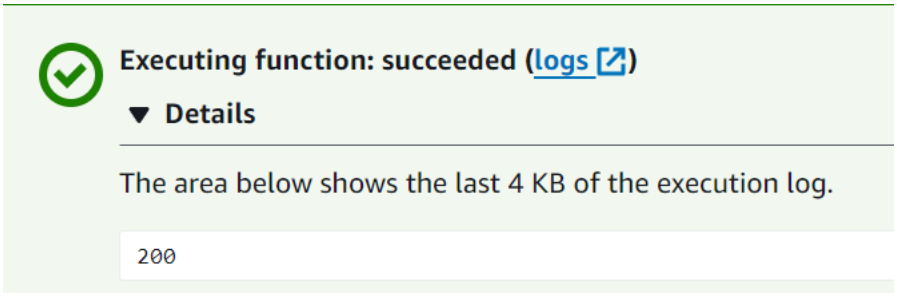



```
import urllib.request

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. [배포]를 선택합니다.
3. 테스트 탭을 선택합니다.



4. 테스트를 선택합니다.
5. 함수가 200 상태 코드를 반환합니다. 이는 해당 함수에 아웃바운드 인터넷 액세스 권한이 있음을 의미합니다.



함수가 퍼블릭 인터넷에 연결할 수 없으면 다음과 같은 오류 메시지가 표시됩니다.

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Lambda용 인바운드 인터페이스 VPC 엔드포인트 연결

Amazon Virtual Private Cloud(Amazon VPC)를 사용하여 AWS 리소스를 호스팅하는 경우, VPC와 Lambda 간에 연결을 설정할 수 있습니다. 이 연결을 사용하여 공용 인터넷을 통과하지 않고 Lambda 함수를 호출할 수 있습니다.

VPC와 Lambda 간에 프라이빗 연결을 설정하려면 인터페이스 VPC 엔드포인트를 생성합니다. 인터페이스 엔드포인트는 인터넷 게이트웨이, NAT 디바이스, VPN 연결 또는 AWS Direct Connect 연결 없이 비공개로 Lambda API에 액세스할 수 있도록 지원하는 AWS PrivateLink에 의해 구동됩니다. VPC의 인스턴스는 Lambda API와 통신하는 데 퍼블릭 IP 주소가 필요하지 않습니다. VPC와 Lambda 간의 트래픽은 AWS 네트워크를 벗어나지 않습니다.

각 인터페이스 엔드포인트는 서브넷에서 하나 이상의 탄력적 네트워크 인터페이스로 표현됩니다. 네트워크 인터페이스는 트래픽에 대한 진입점 역할을 하는 프라이빗 IP 주소를 Lambda에 제공합니다.

단원

- Lambda 인터페이스 엔드포인트의 고려 사항
- Lambda에 대한 인터페이스 엔드포인트 생성
- Lambda에 대한 인터페이스 엔드포인트 정책 생성

Lambda 인터페이스 엔드포인트의 고려 사항

Lambda에 대한 인터페이스 엔드포인트를 설정하기 전에 Amazon VPC 사용 설명서에서 인터페이스 엔드포인트 속성 및 제한 사항을 검토해야 합니다.

VPC에서 모든 Lambda API 작업을 호출할 수 있습니다. 예를 들어 VPC 내에서 `Invoke` API를 호출하여 Lambda 함수를 호출할 수 있습니다. Lambda API의 전체 목록은 Lambda API 참조의 작업을 참조하세요.

`use1-az3`은 Lambda VPC 함수를 위한 제한된 용량 리전입니다. 이 가용 영역의 서브넷을 Lambda 함수와 함께 사용해서는 안 됩니다. 이렇게 하면 운영 중단 시 영역 중복성이 감소할 수 있기 때문입니다.

지속적인 연결을 위한 연결 유지

Lambda는 시간이 지남에 따라 유휴 연결을 제거하므로 지속적인 연결을 유지하려면 연결 유지 지시문을 사용해야 합니다. 함수를 호출할 때 유휴 연결을 재사용하려고 하면 연결 오류가 발생합니다. 지속적인 연결을 유지하려면 런타임과 관련된 연결 유지 지시문을 사용하세요. 예를 들어 AWS SDK for JavaScript 개발자 안내서의 Node.js에서 연결 유지로 연결 재사용을 참조하세요.

청구 고려 사항

인터페이스 엔드포인트를 통해 Lambda 함수에 액세스하는 데는 추가 비용이 발생하지 않습니다. Lambda 요금에 대한 자세한 내용은 AWS Lambda 요금을 참조하세요.

AWS PrivateLink의 표준 요금이 Lambda의 인터페이스 엔드포인트에 적용됩니다. 각 가용 영역에서 인터페이스 엔드포인트가 프로비저닝된 각 시간과 인터페이스 엔드포인트를 통해 처리된 데이터에 대해 AWS 계정에 요금이 청구됩니다. 인터페이스 엔드포인트 요금에 대한 자세한 내용은 AWS PrivateLink 요금을 참조하세요.

VPC 피어링 고려 사항

VPC 피어링을 사용하여 인터페이스 엔드포인트가 있는 VPC에 다른 VPC를 연결할 수 있습니다. VPC 피어링은 두 VPC 간의 네트워킹 연결입니다. 사용자의 자체 두 VPC 간에 또는 다른 AWS 계정의 VPC와 VPC 피어링 연결을 설정할 수 있습니다. VPC는 두 개의 서로 다른 AWS 리전에 있을 수도 있습니다.

피어링된 VPC 간의 트래픽은 AWS 네트워크에 유지되며 공용 인터넷을 통과하지 않습니다. VPC가 피어링되면 두 VPC의 Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스, Amazon Relational Database Service(Amazon RDS) 인스턴스 또는 VPC 지원 Lambda 함수와 같은 리소스는 VPC 중 하나에서 생성된 인터페이스 엔드포인트를 통해 Lambda API에 액세스할 수 있습니다.

Lambda에 대한 인터페이스 엔드포인트 생성

Amazon VPC 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 Lambda에 대한 인터페이스 엔드포인트를 생성할 수 있습니다. 자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 엔드포인트 생성을 참조하세요.

Lambda에 대한 인터페이스 엔드포인트를 생성하려면(콘솔)

1. Amazon VPC 콘솔의 엔드포인트 페이지를 엽니다.
2. 엔드포인트 생성을 선택합니다.
3. [서비스 카테고리(Service category)]에서 AWS 서비스가 선택되어 있는지 확인합니다.
4. 서비스 이름에 com.amazonaws.***region***.lambda를 선택합니다. 유형이 인터페이스인지 확인합니다.
5. VPC와 서브넷을 선택합니다.
6. 인터페이스 엔드포인트에 대한 프라이빗 DNS를 활성화하려면 DNS 이름 활성화 확인란을 선택합니다. AWS 서비스용 VPC 엔드포인트에 프라이빗 DNS 이름을 사용하는 것이 좋습니다. 이렇게

하면 AWS SDK를 통해 이루어진 요청과 같이 퍼블릭 서비스 엔드포인트를 사용하는 요청이 VPC 엔드포인트로 확인됩니다.

7. 보안 그룹에서 하나 이상의 보안 그룹을 선택합니다.
8. Create endpoint(엔드포인트 생성)을 선택합니다.

프라이빗 DNS 옵션을 사용하려면 VPC의 `enableDnsHostnames` 및 `enableDnsSupportattributes`를 설정해야 합니다. 자세한 내용은 Amazon VPC 사용 설명서의 VPC에 대한 DNS 지원 보기 및 업데이트를 참조하세요. 인터페이스 엔드포인트에 프라이빗 DNS를 사용하도록 설정하는 경우, 리전에 대한 기본 DNS 이름(예: `lambda.us-east-1.amazonaws.com`)을 사용하여 Lambda에 API 요청을 할 수 있습니다. 서비스 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 서비스 엔드포인트 및 할당량을 참조하세요.

자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 엔드포인트를 통해 서비스 액세스를 참조하세요.

AWS CloudFormation을 사용하여 엔드포인트를 생성하고 구성하는 방법에 대한 자세한 내용은 AWS CloudFormation 사용 설명서의 AWS::EC2::VPCEndpoint 리소스를 참조하세요.

Lambda에 대한 인터페이스 엔드포인트를 생성하려면(AWS CLI)

`create-vpc-endpoint` 명령을 사용하여 엔드포인트 네트워크 인터페이스와 연결할 VPC ID, VPC 엔드포인트 유형(인터페이스), 서비스 이름, 엔드포인트를 사용할 서브넷 및 보안 그룹을 지정합니다. 예:

```
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
```

Lambda에 대한 인터페이스 엔드포인트 정책 생성

인터페이스 엔드포인트를 사용할 수 있는 사용자와 사용자가 액세스할 수 있는 Lambda 함수를 제어하기 위해 엔드포인트에 엔드포인트 정책을 연결할 수 있습니다. 이 정책은 다음 정보를 지정합니다.

- 작업을 수행할 수 있는 보안 주체.
- 보안 주체가 수행할 수 있는 작업입니다.

- 보안 주체가 작업을 수행할 수 있는 리소스입니다.

자세한 내용은 Amazon VPC 사용 설명서의 VPC 엔드포인트를 통해 서비스에 대한 액세스 제어를 참조하세요.

예: Lambda 작업에 대한 인터페이스 엔드포인트 정책

다음은 Lambda에 대한 엔드포인트 정책의 예입니다. 엔드포인트에 연결된 경우 이 정책을 통해 사용자 `MyUser`가 `my-function` 함수를 호출할 수 있습니다.

> **Note**
>
> 리소스에 정규화된 함수 ARN과 정규화되지 않은 함수 ARN을 모두 포함해야 합니다.

```
{
   "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 함수에 대한 파일 시스템 액세스 구성

Amazon Elastic File System(Amazon EFS) 파일 시스템을 로컬 디렉터리에 마운트하도록 함수를 구성할 수 있습니다. Amazon EFS를 사용하면 함수 코드가 안전하고 높은 동시성으로 공유 리소스에 액세스하여 수정할 수 있습니다.

Sections

실행 역할 및 사용자 권한

파일 시스템에 사용자 구성 AWS Identity and Access Management(IAM) 정책이 없는 경우, EFS는 파일 시스템 탑재 대상을 사용하여 파일 시스템에 연결할 수 있는 모든 클라이언트에 대한 모든 액세스 권한을 부여하는 기본 정책을 사용합니다. 파일 시스템에 사용자 구성 IAM 정책이 있는 경우 함수의 실행 역할에 올바른 `elasticfilesystem` 권한이 있어야 합니다.

실행 역할 권한

- elasticfilesystem:ClientMount
- elasticfilesystem:ClientWrite(읽기 전용 연결에는 필요하지 않음)

이러한 권한은 AmazonElasticFileSystemClientReadWriteAccess 관리형 정책에 포함되어 있습니다. 또한 실행 역할에는 파일 시스템의 VPC에 연결하는 데 필요한 권한도 있어야 합니다.

파일 시스템을 구성할 때 Lambda는 권한을 사용하여 마운트 대상을 확인합니다. 파일 시스템에 연결되도록 함수를 구성하려면 사용자에게 다음 권한이 필요합니다.

사용자 권한

- elasticfilesystem:DescribeMountTargets

파일 시스템 및 액세스 포인트 구성

함수가 연결되는 모든 가용 영역에 마운트 대상이 있는 Amazon EFS에서 파일 시스템을 생성합니다. 성능 및 복원력을 위해 최소 두 개의 가용 영역을 사용합니다. 예를 들어, 간단한 구성에서 별도의 가용 영역에 두 개의 프라이빗 서브넷이 있는 VPC를 가질 수 있습니다. 이 함수는 두 서브넷에 모두 연결되며 각 서브넷에서 마운트 대상을 사용할 수 있습니다. 함수와 마운트 대상에서 사용되는 보안 그룹에서 NFS 트래픽(포트 2049)을 허용하는지 확인합니다.

> **Note**
>
> 파일 시스템을 생성할 때 나중에 변경할 수 없는 성능 모드를 선택합니다. 범용 모드는 지연 시간이 짧으며 최대 I/O 모드는 더 높은 최대 처리량과 IOPS를 지원합니다. 선택에 관한 도움말은 Amazon Elastic File System 사용 설명서의 Amazon EFS 성능을 참조하세요.

액세스 포인트는 함수의 각 인스턴스를 연결된 가용 영역에 대한 올바른 마운트 대상에 연결합니다. 최상의 성능을 위해 루트가 아닌 경로를 사용하여 액세스 포인트를 생성하고 각 디렉터리에서 생성되는 파일 수를 제한합니다. 다음 예제에서는 파일 시스템에 `my-function`라는 디렉터리를 생성하고 소유자 ID를 표준 디렉터리 권한(755)으로 1001로 설정합니다.

Example 액세스 포인트 구성

- 이름 – `files`
- 사용자 ID – `1001`
- 그룹 ID – `1001`
- 경로 – `/my-function`
- 권한 – `755`
- 소유자 사용자 ID – `1001`
- 그룹 사용자 ID – `1001`

함수가 액세스 포인트를 사용할 때 사용자 ID 1001이 지정되고, 디렉터리에 대한 전체 액세스 권한을 가집니다.

자세한 내용은 Amazon Elastic File System 사용 설명서의 다음 항목을 참조하세요.

- Amazon EFS를 위한 리소스 생성

- 사용자, 그룹 및 권한 작업

파일 시스템에 연결(콘솔)

함수는 VPC의 로컬 네트워크를 통해 파일 시스템에 연결됩니다. 함수가 연결되는 서브넷은 파일 시스템의 마운트 지점을 포함하는 동일한 서브넷이거나 NFS 트래픽(포트 2049)을 파일 시스템으로 라우팅할 수 있는 동일한 가용 영역의 서브넷일 수 있습니다.

> **Note**
>
> 함수가 VPC에 아직 연결되어 있지 않은 경우 Lambda 함수에 Amazon VPC의 리소스에 대한 액세스 권한 부여 단원을 참조하세요.

파일 시스템 액세스를 구성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 파일 시스템(File systems)을 선택합니다.
4. 파일 시스템에서 파일 시스템 추가를 선택합니다.
5. 다음 속성을 구성합니다.
   - EFS 파일 시스템 – 동일한 VPC에 있는 파일 시스템에 대한 액세스 포인트입니다.
   - 로컬 마운트 경로 - 파일 시스템이 Lambda 함수에서 마운트되는 위치로, `/mnt/`로 시작합니다.

> **요금**
>
> Amazon EFS는 스토리지 및 처리량에 대해 요금을 청구하며, 요율은 스토리지 클래스에 따라 달라집니다. 자세한 내용은 Amazon EFS 요금을 참조하세요.
> Lambda는 VPC 간 데이터 전송에 대해 요금을 청구합니다. 이는 함수의 VPC가 파일 시스템이 있는 다른 VPC로 피어링된 경우에만 적용됩니다. 요금은 동일한 리전의 VPC 간 Amazon EC2 데이터 전송과 동일합니다. 자세한 내용은 Lambda 요금을 참조하세요.

Lambda의 Amazon EFS 통합에 대한 자세한 내용은 Lambda에서 Amazon EFS 사용 단원을 참조하세요.

Lambda 함수에 대해 다른 AWS 계정 계정의 Amazon EFS 파일 시스템 사용

다른 AWS 계정에 Amazon EFS 파일 시스템을 탑재하도록 함수를 구성할 수 있습니다. 파일 시스템을 탑재하기 전에 다음 사항을 확인해야 합니다.

- VPC 피어링을 구성하고 각 VPC의 라우팅 테이블에 적절한 경로를 추가해야 합니다.
- 탑재하려는 Amazon EFS 파일 시스템의 보안 그룹은 Lambda 함수와 연결된 보안 그룹의 인바운드 액세스를 허용하도록 구성되어야 합니다.
- 가용 영역 ID가 일치하는 각 VPC에 서브넷을 생성해야 합니다.
- 두 VPC 모두에서 DNS 호스트 이름을 활성화해야 합니다.

Lambda 함수가 다른 AWS 계정의 Amazon EFS 파일 시스템에 액세스하려면 함수에 권한을 부여하는 파일 시스템 정책도 해당 파일 시스템에 있어야 합니다. 파일 시스템 정책을 생성하는 방법을 알아보려면 Amazon Elastic File System User Guide의 Creating file system policies를 참조하세요.

다음은 지정된 계정의 Lambda 함수에 파일 시스템에서 모든 API 작업을 수행할 수 있는 권한을 부여하는 정책 예제를 보여줍니다.

```
{
    "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**
>
> 표시된 예제 정책은 와일드카드 문자('*')를 사용하여 지정된 AWS 계정의 Lambda 함수가 파일 시스템에서 API 작업을 수행할 수 있는 권한을 부여합니다. 여기에는 파일 시스템 삭제가 포함

됩니다. 다른 AWS 계정이 파일 시스템에서 수행할 수 있는 작업을 제한하려면 명시적으로 허용하려는 작업을 지정합니다. 가능한 API 작업 목록은 Amazon Elastic File System에 사용되는 작업, 리소스 및 조건 키를 참조하세요.

크로스 계정 파일 시스템 탑재를 구성하려면 AWS Command Line Interface(AWS CLI) `update-function-configuration` 작업을 사용합니다.

다른 AWS 계정에 파일 시스템을 탑재하려면 다음 명령을 실행합니다. 사용자의 함수 이름을 사용하고 Amazon 리소스 이름(ARN)을 탑재하려는 파일 시스템에 대한 Amazon EFS 액세스 포인트의 ARN으로 바꿉니다. `LocalMountPath`는 함수가 파일 시스템에 액세스할 수 있는 경로(`/mnt/`로 시작)입니다. Lambda 탑재 경로가 파일 시스템의 액세스 포인트 경로와 일치하는지 확인하세요. 예를 들어 액세스 포인트가 `/efs`인 경우 Lambda 탑재 경로는 `/mnt/efs`여야 합니다.

```
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 함수에 대한 별칭 생성

Lambda 함수의 별칭을 생성할 수 있습니다. Lambda 별칭은 업데이트할 수 있는 함수 버전에 대한 포인터입니다. 함수 사용자는 별칭 Amazon 리소스 이름(ARN)을 사용하여 함수 버전에 액세스할 수 있습니다. 새 버전을 배포할 때 새 버전을 사용하도록 별칭을 업데이트하거나 두 버전 간에 트래픽을 분할하도록 별칭을 업데이트할 수 있습니다.

Sections

함수 별칭 생성(콘솔)

Lambda 콘솔을 사용하여 함수 별칭을 생성할 수 있습니다.

별칭을 만들려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 별칭(Aliases)을 선택한 다음 별칭 생성(Create alias)을 선택합니다.
4. 별칭 생성(Create alias) 페이지에서 다음을 수행합니다.
    a. 별칭의 이름(Name)을 입력합니다.
    b. (선택 사항) 경보에 대한 설명(Description)을 입력합니다.
    c. 버전(Version)에서 별칭이 가리키도록 할 함수 버전을 선택합니다.
    d. (선택 사항) 별칭에 대한 라우팅을 구성하려면 가중치 기반 별칭(Weighted alias)을 확장합니다. 자세한 내용은 별칭 라우팅 구성 섹션을 참조하세요.
    e. 저장을 선택합니다.

Lambda API를 사용한 별칭 관리

AWS Command Line Interface(AWS CLI)를 사용하여 별칭을 생성하려면 `create-alias` 명령을 사용합니다.

```
aws lambda create-alias --function-name my-function --name alias-name --function-version version-number --description " "
```

새 버전의 함수를 가리키도록 별칭을 변경하려면 `update-alias` 명령을 사용합니다.

```
aws lambda update-alias --function-name my-function --name alias-name --function-version version-number
```

별칭을 삭제하려면 `delete-alias` 명령을 사용합니다.

```
aws lambda delete-alias --function-name my-function --name alias-name
```

이전 단계의 AWS CLI 명령은 다음 Lambda API 작업에 해당합니다.

- CreateAlias
- UpdateAlias
- DeleteAlias

AWS SAM 및 AWS CloudFormation을 사용한 별칭 관리

AWS Serverless Application Model(AWS SAM) 및 AWS CloudFormation을 사용하여 함수 별칭을 생성하고 관리할 수 있습니다.

AWS SAM 템플릿에서 함수 별칭을 선언하는 방법은 AWS SAM 개발자 가이드에서 AWS::Serverless::Function 페이지를 참조하세요. AWS CloudFormation을 사용하여 별칭을 생성하고 구성하는 방법에 대한 자세한 내용은 AWS CloudFormation 사용 설명서에서 AWS::Lambda::Alias를 참조하세요.

별칭 사용

각 별칭에는 고유한 ARN이 있습니다. 별칭은 다른 별칭이 아니라 함수 버전만 가리킬 수 있습니다. 함수의 새 버전을 가리키도록 별칭을 업데이트할 수 있습니다.

Amazon Simple Storage Service(Amazon S3)와 같은 이벤트 소스가 Lambda 함수를 호출합니다. 이러한 이벤트 소스에 이벤트가 발생할 때 호출할 함수를 식별하는 매핑이 유지됩니다. 매핑 구성에서 Lambda 함수 별칭을 지정하는 경우 함수 버전이 변경될 때 매핑을 업데이트할 필요가 없습니다. 자세한 내용은 Lambda가 스트림 및 대기열 기반 이벤트 소스의 레코드를 처리하는 방법 섹션을 참조하세요.

리소스 정책에서 이벤트 소스에 Lambda 함수를 사용할 수 있는 권한을 부여할 수 있습니다. 정책에서 별칭 ARN을 지정하면 함수 버전이 변경될 때 정책을 업데이트할 필요가 없습니다.

리소스 정책

리소스 기반 정책을 사용하여 서비스, 리소스 또는 계정에 함수에 대한 액세스 권한을 부여할 수 있습니다. 해당 권한의 범위는 별칭, 버전 또는 전체 함수에 적용하는지 여부에 따라 다릅니다. 예를 들어, 별칭 이름(예: `helloworld:PROD`)을 사용하는 경우, 권한이 있으면 별칭 ARN(`helloworld`)을 사용하여 `helloworld:PROD` 함수를 호출할 수 있습니다.

별칭이나 특정 버전 없이 함수를 호출하려고 하면 권한 오류가 발생합니다. 이 권한 오류는 별칭과 연결된 함수 버전을 직접 호출하려고 해도 계속 발생합니다.

예를 들어, 다음 AWS CLI 명령은 Amazon S3 permissions가 `DOC-EXAMPLE-BUCKET`을 대신하여 작동할 때 `helloworld` 함수의 PROD 별칭을 호출할 수 있는 Amazon S3 권한을 부여합니다.

```
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
```

정책에서 리소스 이름을 사용하는 방법에 대한 자세한 내용은 정책의 리소스 및 조건 섹션 미세 조정 단원을 참조하세요.

별칭 라우팅 구성

별칭에 대한 라우팅 구성을 사용하여 트래픽의 일부를 두 번째 함수 버전으로 보냅니다. 예를 들어 대부분의 트래픽은 기존 버전으로 전송되고 소수만 새 버전으로 전송되도록 별칭을 구성하여 새 버전을 배포하는 위험을 줄일 수 있습니다.

Lambda는 간단한 확률 모델을 사용하여 두 함수 버전 간에 트래픽을 분산시킵니다. 트래픽 수준이 낮으면 각 버전에서 구성된 트래픽과 실제 트래픽의 비율 간에 큰 차이가 나타날 수 있습니다. 함수가 프로비저닝된 동시성을 사용하는 경우 별칭 라우팅이 활성 상태인 동안 더 많은 수의 프로비전된 동시성 인스턴스를 구성하여 초과(spillover) 호출을 방지할 수 있습니다.

별칭이 최대 두 개의 Lambda 함수 버전을 가리키도록 설정할 수 있습니다. 버전은 다음 기준을 충족해야 합니다.

- 두 버전이 동일한 실행 역할을 가져야 합니다.
- 두 버전 모두 동일한 배달 못한 편지 대기열 구성을 가져야 하거나 배달 못한 편지 대기열 구성이 없어야 합니다.
- 두 버전 모두 게시해야 합니다. 별칭은 $LATEST을 가리킬 수 없습니다.

별칭에 대한 라우팅을 구성하려면

> **Note**
>
> 함수에 게시된 버전이 두 개 이상 있는지 확인합니다. 추가 버전을 생성하려면 Lambda 함수 버전의 지침을 따르세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 별칭(Aliases)을 선택한 다음 별칭 생성(Create alias)을 선택합니다.
4. 별칭 생성(Create alias) 페이지에서 다음을 수행합니다.
   a. 별칭의 이름(Name)을 입력합니다.
   b. (선택 사항) 경보에 대한 설명(Description)을 입력합니다.
   c. 버전(Version)에서 별칭이 가리키도록 할 첫 번째 함수 버전을 선택합니다.
   d. 가중치 기반 별칭(Weighted alias)을 확장합니다.
   e. 추가 버전(Additional version)에서 별칭이 가리킬 두 번째 함수 버전을 선택합니다.
   f. 가중치(Weight)(%)에서 함수의 가중치 값을 입력합니다. 가중치는 별칭이 호출될 때 해당 버전에 할당되는 트래픽의 비율입니다. 첫 번째 버전에는 남은 가중치가 할당됩니다. 예를 들어 Additional version(추가 버전)에 10%를 지정할 경우 첫 번째 버전에 자동으로 90%가 할당됩니다.
   g. 저장을 선택합니다.

CLI를 사용하여 별칭 라우팅 구성

`create-alias` 및 `update-alias` AWS CLI 명령을 사용하여 두 함수 버전 간의 트래픽 가중치를 구성합니다. 별칭을 만들거나 업데이트할 때 `routing-config` 파라미터에 트래픽 가중치를 지정합니다.

다음 예에서는 함수의 버전 1을 가리키는 routing-alias라는 Lambda 함수 별칭을 생성합니다. 함수의 버전 2는 트래픽의 3%를 수신합니다. 나머지 97%의 트래픽은 버전 1로 라우팅됩니다.

```
aws lambda create-alias --name routing-alias --function-name my-function --function-
version 1  \
--routing-config AdditionalVersionWeights={"2"=0.03}
```

이 `update-alias` 명령을 사용하여 버전 2로 들어오는 트래픽의 비율을 늘립니다. 다음 예에서는 트래픽을 5%로 늘립니다.

```
aws lambda update-alias --name routing-alias --function-name my-function  \
--routing-config AdditionalVersionWeights={"2"=0.05}
```

모든 트래픽을 버전 2로 라우팅하려면 `update-alias` 명령을 사용하여 별칭이 버전 2를 가리키도록 `function-version` 속성을 변경합니다. 이 명령은 라우팅 구성도 재설정합니다.

```
aws lambda update-alias --name routing-alias --function-name my-function  \
--function-version 2 --routing-config AdditionalVersionWeights={}
```

이전 단계의 AWS CLI 명령은 다음 Lambda API 작업에 해당합니다.

- CreateAlias
- UpdateAlias

호출된 버전 확인

두 함수 버전 간에 트래픽 가중치를 구성할 때 호출된 Lambda 함수 버전을 확인하는 방법은 두 가지가 있습니다.

- CloudWatch Logs – 매 함수 호출 시 Lambda가 호출된 버전 ID를 포함한 `START` 로그 항목을 자동으로 Amazon CloudWatch Logs로 내보냅니다. 다음은 그 한 예입니다.

  ```
  19:44:37 START RequestId: request id Version: $version
  ```

별칭 호출을 위해 Lambda는 `Executed Version` 차원을 사용하여 호출된 버전을 기준으로 지표 데이터를 필터링합니다. 자세한 내용은 Lambda 함수 지표 작업 섹션을 참조하세요.

- 응답 페이로드(동기식 호출) – 동기식 함수 호출에 대한 응답에 호출된 함수 버전을 나타내는 `x-amz-executed-version` 헤더가 포함되어 있습니다.

Lambda 함수 버전

버전을 사용하여 함수 배포를 관리할 수 있습니다. 예를 들어, 안정적인 프로덕션 버전 사용에 영향을 주지 않고 베타 테스트를 위한 새 버전의 함수를 게시할 수 있습니다. 함수를 게시할 때마다 Lambda는 함수의 새 버전을 생성합니다. 새 버전은 함수의 게시 되지 않은 버전입니다. 게시되지 않은 버전의 이름은 $LATEST입니다.

> **Note**
>
> 함수의 새 버전을 생성하려면 먼저 게시되지 않은 버전($LATEST)을 변경해야 합니다. 이러한 변경에는 코드 업데이트 또는 구성 설정 수정이 포함될 수 있습니다. $LATEST가 이전에 게시된 버전과 동일한 경우 $LATEST에 변경 내용을 배포할 때까지 새 버전을 생성할 수 없습니다.

함수 버전을 게시한 후에는 해당 코드, 런타임, 아키텍처, 메모리, 계층 및 대부분의 기타 구성 설정을 변경할 수 없습니다. 즉, $LATEST에서 새 버전을 게시하지 않으면 이러한 설정을 변경할 수 없습니다. 게시된 함수 버전에 대해 다음 항목을 구성할 수 있습니다.

- 트리거
- 대상
- 프로비저닝된 동시성
- 비동기식 호출
- 데이터베이스 연결 및 프록시

> **Note**
>
> 자동 모드에서 런타임 관리 제어를 사용하는 경우 함수 버전에서 사용하는 런타임 버전이 자동으로 업데이트됩니다. 함수 업데이트 또는 수동 모드를 사용하는 경우 런타임 버전이 업데이트되지 않습니다. 자세한 내용은 the section called “런타임 업데이트” 단원을 참조하십시오.

Sections

- 함수 버전 생성
- 버전 사용
- 권한 부여

함수 버전 생성

함수의 게시 되지 않은 버전에서만 함수 코드와 설정을 변경할 수 있습니다. 버전을 게시하면 Lambda는 해당 버전의 사용자에게 일관된 경험을 유지하기 위해 코드와 대부분의 설정을 잠급니다.

Lambda 콘솔을 사용하여 함수 버전을 생성할 수 있습니다.

새 함수 버전을 생성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택한 다음 버전(Versions)을 선택합니다.
3. 버전 구성 페이지에서 새 버전 발행(Publish new version)을 선택합니다.
4. (선택 사항) 버전 설명을 입력합니다.
5. 게시를 선택합니다.

또는 PublishVersion API 작업을 사용하여 함수의 버전을 게시할 수 있습니다.

다음 AWS CLI 명령은 함수의 새 버전을 게시합니다. 응답으로 버전 번호, 버전 접미사가 포함된 함수 ARN과 같은 새 버전에 대한 구성 정보가 반환됩니다.

```
aws lambda publish-version --function-name my-function
```

다음 결과가 표시됩니다.

```
{
  "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**
>
> Lambda는 버전 관리를 위해 단조 증가하는 시퀀스 번호를 할당합니다. Lambda는 함수를 삭제하고 다시 생성한 후에도 버전 번호를 재사용하지 않습니다.

버전 사용

정규화된 ARN 또는 정규화되지 않은 ARN을 사용하여 Lambda 함수를 참조할 수 있습니다.

- 정규화된 ARN – 버전 접미사가 포함된 함수 ARN입니다. 다음 예에서는 `helloworld` 함수의 버전 42를 참조합니다.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- 정규화되지 않은 ARN – 버전 접미사가 포함되지 않은 함수 ARN입니다.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

모든 관련 API 작업에서 정규화된 ARN 또는 정규화되지 않은 ARN을 사용할 수 있습니다. 그러나 정규화되지 않은 ARN을 사용하여 별칭을 생성할 수는 없습니다.

함수 버전을 게시하지 않기로 결정한 경우, 이벤트 소스 매핑에서 정규화된 ARN 또는 정규화되지 않은 ARN을 사용하여 함수를 호출할 수 있습니다. 정규화되지 않은 ARN을 사용하여 함수를 호출하면 Lambda가 $LATEST를 묵시적으로 호출합니다.

Lambda는 코드가 게시된 적이 없거나 마지막으로 게시된 버전과 비교하여 코드가 변경된 경우에만 새 함수 버전을 게시합니다. 변경 사항이 없으면 함수 버전은 마지막으로 게시된 버전으로 유지됩니다.

각 Lambda 함수 버전에 대한 정규화된 ARN은 고유합니다. 버전을 게시한 후에는 ARN 또는 함수 코드를 변경할 수 없습니다.

권한 부여

리소스 기반 정책 또는 자격 증명 기반 정책을 사용하여 함수에 대한 액세스 권한을 부여할 수 있습니다. 권한의 범위는 정책을 함수에 적용하는지 또는 함수의 한 버전에 적용하는지에 따라 다릅니다. 정책의 함수 리소스 이름에 대한 자세한 내용은 정책의 리소스 및 조건 섹션 미세 조정 단원을 참조하세요.

함수 별칭을 사용하여 이벤트 소스 및 AWS Identity and Access Management(IAM) 정책의 관리를 간소화할 수 있습니다. 자세한 내용은 Lambda 함수에 대한 별칭 생성 섹션을 참조하세요.

응답을 스트리밍하도록 Lambda 함수 구성

응답 페이로드를 클라이언트로 다시 스트리밍하도록 Lambda 함수 URL을 구성할 수 있습니다. 응답 스트리밍은 첫 번째 바이트까지 시간(TTFB) 성능을 개선하여 지연 시간에 민감한 애플리케이션에 도움이 될 수 있습니다. 이는 부분 응답을 사용할 수 있게 되면 클라이언트에 다시 전송할 수 있기 때문입니다. 또한 응답 스트리밍을 사용하여 더 큰 페이로드를 반환하는 함수를 빌드할 수 있습니다. 버퍼링된 응답의 소프트 한도는 6MB인데 반해 응답 스트림 페이로드의 소프트 한도는 20MB입니다. 응답을 스트리밍하므로 함수가 전체 응답을 메모리에 담을 필요가 없습니다. 응답이 매우 큰 경우 함수에 대해 구성해야 하는 메모리 양을 줄일 수 있습니다.

Lambda가 응답을 스트리밍하는 속도는 응답 크기에 따라 달라집니다. 함수 응답의 처음 6MB에 대한 스트리밍 속도에는 제한이 없습니다. 6MB보다 큰 응답의 경우 응답의 나머지 부분에 대역폭 한도가 적용됩니다. 스트리밍 대역폭에 대한 자세한 내용은 응답 스트리밍에 대한 대역폭 한도 섹션을 참조하세요.

스트리밍 응답에는 비용이 발생합니다. 자세한 내용은 AWS Lambda 요금을 참조하세요.

Lambda는 Node.js 관리형 런타임에서 응답 스트리밍을 지원합니다. 다른 언어의 경우, 사용자 지정 런타임을 사용자 지정 런타임 API 통합과 함께 사용하여 응답을 스트리밍하거나 Lambda Web Adapter를 사용할 수 있습니다. Lambda 함수 URL, AWS SDK 또는 Lambda API를 사용하여 응답을 스트리밍할 수 있습니다. InvokeWithResponseStream

> **Note**
>
> Lambda 콘솔을 통해 함수를 테스트할 때 항상 응답이 버퍼링된 것으로 표시됩니다.

응답 스트리밍 지원 함수 작성

응답 스트리밍 함수의 핸들러 작성은 일반적인 핸들러 패턴과 다릅니다. 스트리밍 함수를 작성할 때 다음을 수행해야 합니다.

- 네이티브 Node.js 런타임에서 제공하는 `awslambda.streamifyResponse()` 데코레이터로 함수를 래핑합니다.
- 모든 데이터 처리가 완료되도록 스트림을 정상적으로 종료합니다.

응답을 스트리밍하도록 핸들러 함수 구성

Lambda가 함수의 응답을 스트리밍해야 함을 런타임에 나타내려면 함수를 `streamifyResponse()` 데코레이터로 래핑해야 합니다. 그러면 런타임이 응답 스트리밍에 적합한 로직 경로를 사용하고 함수가 응답을 스트리밍할 수 있습니다.

`streamifyResponse()` 데코레이터는 다음 파라미터를 수락하는 함수를 수락합니다.

- `event` - HTTP 메서드, 쿼리 파라미터 및 요청 본문과 같은 함수 URL의 호출 이벤트에 대한 정보를 제공합니다.
- `responseStream` - 쓰기 가능한 스트림을 제공합니다.
- `context` – 호출, 함수 및 실행 환경에 대한 정보를 메서드 및 속성에 제공합니다.

`responseStream` 객체는 Node.js `writableStream`입니다. 이러한 스트림과 마찬가지로 `pipeline()` 메서드를 사용해야 합니다.

Example 응답 스트리밍 지원 핸들러

```
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`은 스트림에 쓰기를 위한 `write()` 메서드를 제공하지만 가능하면 `pipeline()`을 사용하는 것이 좋습니다. `pipeline()`을 사용하면 쓰기 가능한 스트림이 더 빠른 읽기 가능한 스트림에 의해 압도되지 않도록 합니다.

스트리밍 종료

핸들러가 반환되기 전에 스트림을 제대로 종료해야 합니다. `pipeline()` 메서드가 이를 자동으로 처리합니다.

다른 사용 사례의 경우 `responseStream.end()` 메서드를 호출하여 스트림을 올바르게 종료하세요. 이 메서드는 스트림에 더 이상 데이터를 쓰지 말아야 한다는 신호를 보냅니다. `pipeline()` 또는 `pipe()`를 사용하여 스트림에 쓰는 경우에는 이 메서드가 필요하지 않습니다.

Example pipeline()을 사용하여 스트림을 종료하는 예제

```
const pipeline = require("util").promisify(require("stream").pipeline);

exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  await pipeline(requestStream, responseStream);
});
```

Example pipeline()을 사용하지 않고 스트림을 종료하는 예제

```
exports.handler = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Lambda 함수 URL을 사용하여 응답 스트리밍 지원 함수 호출

> **Note**
>
> 응답을 스트리밍하려면 함수 URL을 사용하여 함수를 호출해야 합니다.

함수 URL의 호출 모드를 변경하여 응답 스트리밍 지원 함수를 호출할 수 있습니다. 호출 모드는 Lambda가 함수를 호출하는 데 사용하는 API 작업을 결정합니다. 사용 가능한 호출 모드는 다음과 같습니다.

- `BUFFERED` - 기본 옵션입니다. Lambda는 `Invoke` API 작업을 사용하여 함수를 호출합니다. 페이로드가 완료되면 호출 결과를 사용할 수 있습니다. 최대 페이로드 크기는 6MB입니다.
- `RESPONSE_STREAM` - 함수가 페이로드 결과를 사용할 수 있게 되면 스트리밍할 수 있도록 합니다. Lambda는 `InvokeWithResponseStream` API 작업을 사용하여 함수를 호출합니다. 최대 응답 페이로드 크기는 20MB입니다. 그러나 할당량 증가를 요청할 수 있습니다.

Invoke API 작업을 직접 호출하여 응답 스트리밍 없이 함수를 호출할 수 있습니다. 그러나 Lambda는 호출 모드를 BUFFERED로 변경할 때까지 함수 URL을 통해 들어오는 호출에 대한 모든 응답 페이로드를 스트리밍합니다.

함수 URL의 호출 모드 설정(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 호출 모드를 설정할 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 함수 URL(Function URL)을 선택합니다.
4. 편집을 선택하고 추가 설정을 선택합니다.
5. 호출 모드에서 원하는 호출 모드를 선택합니다.
6. 저장을 선택합니다.

함수 URL의 호출 모드 설정(AWS CLI)

```
aws lambda update-function-url-config --function-name my-function --invoke-mode
 RESPONSE_STREAM
```

함수 URL의 호출 모드 설정(AWS CloudFormation)

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

함수 구성에 대한 자세한 내용은 Lambda 함수 URL 섹션을 참조하세요.

응답 스트리밍에 대한 대역폭 한도

함수 응답 페이로드의 처음 6MB에는 대역폭 한도가 없습니다. 이 초기 버스트 이후 Lambda는 최대 2MBps의 속도로 응답을 스트리밍합니다. 함수 응답이 6MB를 초과하지 않는 경우 이 대역폭 한도는 적용되지 않습니다.

> **Note**
>
> 대역폭 한도는 함수의 응답 페이로드에만 적용되며 함수의 네트워크 액세스에는 적용되지 않습니다.

제한되지 않은 대역폭의 속도는 함수의 처리 속도를 비롯한 여러 요인에 따라 달라집니다. 일반적으로 함수 응답의 처음 6MB에 대해 2MBps보다 높은 속도를 예상할 수 있습니다. 함수가 AWS 외부의 대상으로 응답을 스트리밍하는 경우 스트리밍 속도는 외부 인터넷 연결 속도에 따라서도 달라집니다.

자습서: 함수 URL을 사용하여 응답 스트리밍 Lambda 함수 생성

이 자습서에서는 응답 스트림을 반환하는 함수 URL 엔드포인트를 사용하여 .zip 파일 아카이브로 정의된 Lambda 함수를 생성합니다. 함수 구성에 대한 자세한 내용은 함수 URL 생성 및 관리 섹션을 참조하세요.

필수 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:zip)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식

을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 만들기

Lambda 함수에 AWS 리소스에 액세스할 수 있는 권한을 제공하는 실행 역할을 만듭니다.

실행 역할을 만들려면

1. AWS Identity and Access Management(IAM) 콘솔의 역할 페이지를 엽니다.
2. 역할 생성(Create role)을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.
   - 신뢰할 수 있는 엔티티 유형 — AWS서비스
   - 사용 사례 — Lambda
   - 권한 – AWSLambdaBasicExecutionRole
   - 역할 이름 – **`response-streaming-role`**

AWSLambdaBasicExecutionRole 정책은 함수가 Amazon CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다. 역할을 생성한 후 Amazon 리소스 이름(ARN)을 기록해 둡니다. 다음 단계에서 이 정보를 사용할 것입니다.

응답 스트리밍 함수 생성(AWS CLI)

AWS Command Line Interface(AWS CLI)를 사용하여 함수 URL 엔드포인트로 응답 스트리밍 Lambda 함수를 생성합니다.

응답을 스트리밍할 수 있는 함수 생성

1. 다음 코드 예제를 `index.mjs`라는 파일에 복사합니다.

```
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. 배포 패키지를 만듭니다.

```
zip function.zip index.mjs
```

3. `create-function` 명령을 사용해 Lambda 함수를 만듭니다. `--role` 값을 이전 단계의 역할 ARN으로 변경합니다.

```
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
```

함수 URL 생성

1. 함수 URL에 대한 액세스를 허용하도록 함수에 리소스 기반 정책을 추가합니다. `--principal` 값을 AWS 계정 ID로 바꿉니다.

```
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` 명령을 사용하여 함수에 대한 URL 엔드포인트를 만듭니다.

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

함수 URL 엔드포인트 테스트

함수를 호출하여 통합을 테스트합니다. 브라우저에서 함수의 URL을 열거나 curl을 사용할 수 있습니다.

```
curl --request GET "<function_url>" --user "<key:token>" --aws-sigv4 "aws:amz:us-
east-1:lambda" --no-buffer
```

함수 URL은 `IAM_AUTH` 인증 유형을 사용합니다. 즉, AWS 액세스 키와 비밀 키를 모두 사용하여 요청에 서명해야 합니다. 이전 명령에서 `<key:token>`을 AWS 액세스 키 ID로 바꿉니다. 메시지가 나타나면 AWS 비밀 키를 입력합니다. AWS 비밀 키가 없으면 임시 AWS 보안 인증을 대신 사용할 수 있습니다.

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수 배포

zip 파일 아카이브를 업로드하거나 컨테이너 이미지를 생성 및 업로드하여 Lambda 함수에 코드를 배포할 수 있습니다.

주제

.zip 파일 아카이브

.zip 파일 아카이브에는 애플리케이션 코드와 해당 종속 항목이 포함됩니다. Lambda 콘솔이나 도구 키트를 사용하여 함수를 작성하면 Lambda가 코드의.zip 파일 아카이브를 자동으로 생성합니다.

Lambda API, 명령줄 도구 또는 AWS SDK를 사용하여 함수를 생성하는 경우 배포 패키지를 생성해야 합니다. 함수가 컴파일된 언어를 사용하거나 함수에 종속 항목을 추가하는 경우에도 배포 패키지를 생성해야 합니다. 함수 코드를 배포하려면 Amazon Simple Storage Service(Amazon S3) 또는 로컬 컴퓨터에서 배포 패키지를 업로드합니다.

Lambda 콘솔 또는 AWS Command Line Interface(AWS CLI)를 사용하여 Amazon Simple Storage Service(Amazon S3) 버킷에 .zip 파일을 배포 패키지로 업로드할 수 있습니다.

배포 패키지 파일 권한

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 chmod 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

컨테이너 이미지

Docker 명령줄 인터페이스(CLI)와 같은 도구를 사용하여 코드 및 종속 항목을 컨테이너 이미지로 패키지화할 수 있습니다. 그런 다음 Amazon Elastic Container Registry(Amazon ECR)에서 호스팅되는 컨테이너 레지스트리에 이미지를 업로드할 수 있습니다.

함수를 호출하면 Lambda는 컨테이너 이미지를 실행 환경에 배포합니다. Lambda는 모든 익스텐션를 초기화한 다음 함수의 초기화 코드(기본 핸들러 외부의 코드)를 실행합니다. 함수 초기화 기간은 청구된 실행 시간에 포함됩니다.

이후 Lambda는 함수 구성(ENTRYPOINT 및 CMD 컨테이너 이미지 설정)에 지정된 코드 진입점을 직접적으로 호출하여 함수를 실행합니다.

AWS는 함수 코드의 컨테이너 이미지를 작성하는 데 사용할 수 있는 일련의 오픈 소스 기본 이미지를 제공합니다. 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수도 있습니다. AWS는 대체 기본 이미지에 추가하여 Lambda 서비스와 호환되도록 만들 수 있는 오픈 소스 런타임 클라이언트를 제공합니다.

또한 AWS는 Docker CLI와 같은 도구를 사용하여 로컬에서 함수를 테스트할 수 있는 런타임 인터페이스 에뮬레이터를 제공합니다.

> **Note**
>
> Lambda가 지원하는 명령 세트 아키텍처 중 하나와 호환되도록 각 컨테이너 이미지를 만듭니다. Lambda는 각 명령 세트 아키텍처에 대한 기본 이미지를 제공하며 Lambda는 두 아키텍처를 모두 지원하는 기본 이미지도 제공합니다.
> 함수에 대해 빌드하는 이미지는 아키텍처 중 하나만 대상으로 해야 합니다.

함수를 컨테이너 이미지로 패키징하고 배포하는 데에는 추가 비용이 들지 않습니다. 컨테이너 이미지로 배포된 함수가 호출되면 호출 요청 및 실행 기간에 대한 비용을 지불합니다. Amazon ECR에 컨테이너 이미지를 저장하는 것과 관련된 요금이 발생합니다 . 자세한 정보는 Amazon ECR 요금을 참조하세요.

이미지 보안

Lambda가 원래 소스(Amazon ECR)에서 컨테이너 이미지를 처음 다운로드할 때 컨테이너 이미지가 인증된 컨버전트 암호화 방법을 사용하여 최적화되고 암호화되며 저장됩니다. 고객 데이터를 복호화

하는 데 필요한 모든 키는 AWS KMS 고객 관리형 키를 사용하여 보호됩니다. Lambda의 고객 관리형 키 사용량을 추적하고 감사하려면 AWS CloudTrail 로그를 보면 됩니다.

.zip 파일 아카이브를 사용하여 Lambda 함수 배포

Lambda 함수를 만들 때 배포 패키지에 함수 코드를 패키징합니다. Lambda는 컨테이너 이미지와 .zip 파일 아카이브라는 두 가지 배포 패키지를 지원합니다. 함수를 만드는 워크플로는 배포 패키지 유형에 따라 다릅니다. 컨테이너 이미지로 정의된 함수를 구성하려면 the section called “컨테이너 이미지” 섹션을 참조하세요.

Lambda 콘솔과 Lambda API를 사용하여 .zip 파일 아카이브로 정의된 함수를 만들 수 있습니다. 업데이트된 .zip 파일을 업로드하여 함수 코드를 변경할 수도 있습니다.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

주제

함수 생성

.zip 파일 아카이브로 정의된 함수를 만들 때 코드 템플릿, 언어 버전, 함수의 실행 역할을 선택합니다. Lambda가 함수를 생성하면 함수 코드를 추가합니다.

함수를 만들려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 생성을 선택합니다.
3. 함수 생성을 위해 새로 작성 또는 블루프린트 사용을 선택합니다.

4. 기본 정보에서 다음과 같이 합니다.

   a. 함수 이름(Function name)에 함수 이름을 입력합니다. 함수 이름은 64자로 제한됩니다.

   b. 런타임에서 함수에 사용할 언어 버전을 선택합니다.

   c. (선택 사항) [아키텍처(Architecture)]에서 함수에 사용할 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수의 배포 패키지를 빌드할 때 해당 이미지가 이 명령 세트 아키텍처와 호환되는지 확인합니다.

5. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 역할을 사용할 수 있습니다.

6. (선택 사항) Advanced settings(고급 설정)를 확장합니다. 함수에 대한 코드 서명 구성을 선택할 수 있습니다. 액세스할 함수에 대해 (Amazon VPC)를 구성할 수도 있습니다.

7. 함수 생성을 선택합니다.

Lambda가 새 함수를 생성합니다. 이제 콘솔을 사용하여 함수 코드를 추가하고 다른 함수 파라미터와 기능을 구성할 수 있습니다. 코드 배포 지침은 함수에서 사용하는 런타임의 핸들러 페이지를 참조하세요.

Node.js

.zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포

Python

Python Lambda 함수에 대한 .zip 파일 아카이브 작업

Ruby

Ruby Lambda 함수에 대한 .zip 파일 아카이브 작업

Java

.zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포

Go

.zip 파일 아카이브를 사용하여 Go Lambda 함수 배포

C#

.zip 파일 아카이브를 사용하여 C# Lambda 함수를 빌드 및 배포

PowerShell

.zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포

콘솔 코드 편집기 사용

콘솔은 단일 소스 파일로 Lambda 함수를 생성합니다. 스크립트 언어에 맞게 기본 제공 코드 편집기에서 이 파일을 편집하고 더 많은 파일을 추가할 수 있습니다. 변경 사항을 저장하려면 [Save]를 선택합니다. 그런 다음 코드를 실행하려면 테스트를 선택합니다.

> **Note**
>
> Lambda 콘솔은 AWS Cloud9를 사용하여 브라우저에서 통합 개발 환경(IDE)을 제공합니다. AWS Cloud9을 사용하면 자신의 환경에서 Lambda 함수를 개발할 수도 있습니다. 자세한 내용은 AWS Cloud9 사용 설명서의 Working with AWS Lambda functions using the AWS Toolkit을 참조하세요.

함수 코드를 저장하면 Lambda 콘솔에서 .zip 파일 아카이브 배포 패키지를 만듭니다. 콘솔 외부에서(IDE를 사용해) 함수 코드를 개발하는 경우 Lambda 함수에 코드를 업로드하려면 배포 패키지를 생성해야 합니다.

함수 코드 업데이트

스크립트 언어(Node.js, Python, Ruby)에 맞게 내장된 코드 편집기에서 함수 코드를 편집할 수 있습니다. 코드가 3MB보다 크거나 라이브러리를 추가해야 하는 경우 또는 편집기가 지원하지 않는 언어(Java, Go, C#)의 경우, 함수 코드를 .zip 아카이브로 업로드해야 합니다. .zip 파일 아카이브가 50MB보다 작은 경우, 로컬 시스템에서 .zip 파일 아카이브를 업로드할 수 있습니다. 파일이 50MB보다 큰 경우 Amazon S3 버킷에서 함수로 파일을 업로드합니다.

함수 코드를 .zip 아카이브로 업로드하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 업데이트할 함수를 선택하고 코드 탭을 선택합니다.
3. 코드 소스(Code source)에서 업로드(Upload from)를 선택합니다.
4. .zip 파일을 선택한 후 [업로드(Upload)]를 선택합니다.
   - 파일 선택기에서 새 이미지 버전을 선택하고 [열기(Open)]와 [저장(Save)]을 차례로 선택합니다.
5. (4단계의 대안) Amazon S3 위치를 선택합니다.
   - 텍스트 상자에서 .zip 파일 아카이브의 S3 링크 URL을 입력한 후 저장을 선택합니다.

런타임 변경

새 런타임을 사용하도록 함수 구성을 업데이트하는 경우, 새 런타임과 호환되도록 함수 코드를 업데이트해야 할 수 있습니다. 다른 런타임을 사용하도록 함수 구성을 업데이트하는 경우, 런타임 및 아키텍처와 호환되는 새로운 함수 코드를 제공해야 합니다. 함수 코드의 배포 패키지를 만드는 방법에 대한 지침은 함수가 사용하는 런타임의 핸들러 페이지를 참조하세요.

런타임을 변경하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 업데이트할 함수를 선택하고 코드 탭을 선택합니다.
3. 코드 편집기 아래에 있는 런타임 설정 섹션까지 아래로 스크롤합니다.
4. 편집을 선택합니다.
   a. Runtime(런타임)에서 런타임 식별자를 선택합니다.
   b. 핸들러에서 함수의 파일 이름과 핸들러를 지정합니다.
   c. [아키텍처(Architecture)]에서 함수에 사용할 명령 세트 아키텍처를 선택합니다.
5. Save(저장)를 선택합니다.

아키텍처 변경

명령 세트 아키텍처를 변경하려면 먼저 함수의 코드가 대상 아키텍처와 호환되는지 확인해야 합니다.

Node.js, Python 또는 Ruby를 사용하고 내장된 편집기에서 함수 코드를 편집할 경우, 기존 코드는 수정 없이 실행될 수 있습니다.

그러나 .zip 파일 아카이브 배포 패키지를 사용하여 함수 코드를 제공하는 경우에는 대상 런타임 및 명령 세트 아키텍처에 맞게 올바르게 컴파일되고 빌드되는 새로운 .zip 파일 아카이브를 준비해야 합니다. 자세한 내용은 함수 런타임의 핸들러 페이지를 참조하세요.

명령 세트 아키텍처를 변경하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 업데이트할 함수를 선택하고 코드 탭을 선택합니다.
3. 런타임 설정에서 편집을 선택합니다.
4. [아키텍처(Architecture)]에서 함수에 사용할 명령 세트 아키텍처를 선택합니다.
5. Save(저장)를 선택합니다.

Lambda API 사용

.zip 파일 아카이브를 사용하는 함수를 만들고 구성하려면 다음 API 작업을 사용합니다.

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

AWS CloudFormation

AWS CloudFormation을 사용해 .zip 파일 아카이브를 사용하는 Lambda 함수를 생성할 수 있습니다. AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스는 Lambda 함수를 지정합니다. `AWS::Lambda::Function` 리소스의 속성에 대한 설명은 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function을 참조하세요.

`AWS::Lambda::Function` 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 정의된 함수를 생성합니다.

- AWS::Lambda::Function
  - PackageType - `Zip`(으)로 설정합니다.
  - 코드 - `S3Bucket` 및 `S3Key` 필드에 Amazon S3 버킷 이름과 .zip 파일 이름을 입력합니다. Node.js 또는 Python의 경우 Lambda 함수의 인라인 소스 코드를 제공할 수 있습니다.
  - 런타임 - 런타임 값을 설정합니다.
  - 아키텍처 - AWS Graviton2 프로세서를 사용하도록 아키텍처 값을 `arm64`로 설정합니다. 기본적으로, 아키텍처 값은 `x86_64`입니다.

컨테이너 이미지를 사용하여 Lambda 함수 생성

AWS Lambda 함수의 코드는 스크립트 또는 컴파일된 프로그램과 해당 종속 항목으로 구성됩니다. 함수 코드는 배포 패키지를 사용하여 Lambda에 배포합니다. Lambda는 컨테이너 이미지 및 .zip 파일 아카이브의 두 가지 배포 패키지를 지원합니다.

Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- Lambda용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.
- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 해당 언어용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.
- 비 AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 해당 언어용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

컨테이너 이미지에서 Lambda 함수를 생성하려면 이미지를 로컬에서 빌드하고 Amazon Elastic Container Registry(Amazon ECR) 리포지토리에 업로드합니다. 그런 다음 함수를 생성할 때 리포지토리 URI를 지정합니다. Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다. 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

이 페이지에서는 Lambda 호환 컨테이너 이미지 생성을 위한 기본 이미지 유형과 요구 사항을 설명합니다.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

주제

요구 사항

AWS Command Line Interface(AWS CLI) 버전 2와 Docker CLI를 설치합니다. 또한 다음 요구 사항에 유의하세요.

- 컨테이너 이미지는 Lambda 런타임 API를 구현해야 합니다. AWS 오픈 소스 런타임 인터페이스 클라이언트는 이 API를 구현합니다. 런타임 인터페이스 클라이언트를 선호하는 기본 이미지에 추가하여 Lambda와 호환되도록 만들 수 있습니다.
- 컨테이너 이미지는 읽기 전용 파일 시스템에서 실행할 수 있어야 합니다. 함수 코드는 512MB에서 10,240MB 사이의 스토리지가 있는 쓰기 가능한 `/tmp` 디렉터리에 1MB 단위로 액세스할 수 있습니다.
- 기본 Lambda 사용자는 함수 코드를 실행하는 데 필요한 모든 파일을 읽을 수 있어야 합니다. Lambda는 권한이 최소 권한인 기본 Linux 사용자를 정의하여 보안 모범 사례를 따릅니다. 애플리케이션 코드가 다른 Linux 사용자의 실행이 제한된 파일에 의존하지 않는지 확인합니다.
- Lambda는 Linux 기반 컨테이너 이미지만 지원합니다.

- Lambda는 다중 아키텍처 기본 이미지를 제공합니다. 하지만 함수에 대해 빌드하는 이미지는 아키텍처 중 하나만 대상으로 해야 합니다. Lambda는 다중 아키텍처 컨테이너 이미지를 사용하는 함수를 지원하지 않습니다.

Lambda용 AWS 기본 이미지 사용

Lambda용 AWS 기본 이미지 중 하나를 사용하여 함수 코드의 컨테이너 이미지를 빌드할 수 있습니다. 기본 이미지는 Lambda에서 컨테이너 이미지를 실행하는 데 필요한 언어 런타임 및 기타 구성 요소가 미리 로드되어 있습니다. 함수 코드와 종속 항목을 기본 이미지에 추가한 다음 컨테이너 이미지로 패키징합니다.

AWS는 Lambda용 AWS 기본 이미지를 주기적으로 업데이트합니다. Dockerfile의 FROM 속성에 이미지 이름이 포함되어 있으면 Docker 클라이언트는 Amazon ECR 리포지토리에서 최신 버전의 이미지를 가져옵니다. 업데이트된 기본 이미지를 사용하려면 컨테이너 이미지를 다시 빌드하고 함수 코드를 업데이트해야 합니다.

Node.js 20, Python 3.12, Java 21, AL2023 이상의 기본 이미지는 Amazon Linux 2023 최소 컨테이너 이미지를 기반으로 합니다. 이전 기본 이미지는 Amazon Linux 2를 사용합니다. AL2023은 작은 배포 공간과 `glibc`와 같이 업데이트된 라이브러리 버전을 포함하여 Amazon Linux 2에 비해 여러 가지 이점을 제공합니다.

AL2023 기반 이미지는 `microdnf`(`dnf` 심볼릭 링크)를 Amazon Linux 2에서 기본 패키지 관리자인 `yum` 대신 패키지 관리자로 사용합니다. `microdnf`는 `dnf`의 독립 실행형 구현입니다. AL2023 기반 이미지에 포함된 패키지 목록의 경우 Comparing packages installed on Amazon Linux 2023 Container Images의 Minimal Container 열을 참조하세요. AL2023과 Amazon Linux 2의 차이점에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Introducing the Amazon Linux 2023 runtime for AWS Lambda를 참조하세요.

> **Note**
>
> AWS Serverless Application Model(AWS SAM)을 포함하여 AL2023 기반 이미지를 로컬에서 실행하려면 Docker 버전 20.10.10 이상을 사용해야 합니다.

AWS 기본 이미지를 사용하여 컨테이너 이미지를 빌드하려면 선호하는 언어에 대한 지침을 선택합니다.

- Node.js

- TypeScript(Node.js 기본 이미지 사용)
- Python
- Java
- Go
- .NET
- Ruby

AWS OS 전용 기본 이미지 사용

AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 해당 언어용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
al2023	OS 전용 런타임	Amazon Linux 2023	GitHub의 OS 전용 런타임용 Dockerfile	
al2	OS 전용 런타임	Amazon Linux 2	GitHub의 OS 전용 런타임용 Dockerfile	

Amazon Elastic Container Registry 퍼블릭 갤러리: gallery.ecr.aws/lambda/provided

비 AWS 기본 이미지 사용

Lambda는 다음 이미지 매니페스트 형식 중 하나에 부합하는 모든 이미지를 지원합니다.

- Docker 이미지 매니페스트 V2, 스키마 2(Docker 버전 1.10 이상에서 사용됨)
- Open Container Initiative(OCI) 사양(v1.0.0 이상)

Lambda는 모든 레이어를 포함한 최대 10GB의 비압축 이미지 크기를 지원합니다.

> **Note**
>
> 이미지가 Lambda와 호환되도록 하려면 해당 언어용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

런타임 인터페이스 클라이언트

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API를 확장해야 합니다. AWS는 다음 언어에 대한 오픈 소스 런타임 인터페이스 클라이언트를 제공합니다.

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust - Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

AWS에서 제공하는 런타임 인터페이스 클라이언트가 없는 언어를 사용하는 경우 직접 생성해야 합니다.

Amazon ECR 권한

컨테이너 이미지에서 Lambda 함수를 생성하기 전에 이미지를 로컬로 빌드하고 Amazon ECR 리포지토리에 업로드해야 합니다. 함수를 생성할 때 Amazon ECR 리포지토리 URI를 지정합니다.

함수를 생성하는 사용자 또는 역할에 대한 권한에 `GetRepositoryPolicy` 및 `SetRepositoryPolicy`가 포함되어 있는지 확인합니다.

예를 들어 IAM 콘솔을 사용하여 다음 정책을 포함하는 역할을 생성합니다.

```
{
  "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 리포지토리 정책

Amazon ECR의 컨테이너 이미지와 동일한 계정의 함수에 대해 Amazon ECR 리포지토리 정책에 `ecr:BatchGetImage` 및 `ecr:GetDownloadUrlForLayer` 권한을 추가할 수 있습니다. 다음 예제는 최소한의 정책을 보여줍니다.

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

Amazon ECR 리포지토리 권한에 대한 자세한 내용은 Amazon Elastic Container Registry 사용 설명서의 프라이빗 리포지토리 정책을 참조하세요.

Amazon ECR 리포지토리에 이러한 권한이 없는 경우 Lambda는 컨테이너 이미지 리포지토리 권한에 `ecr:BatchGetImage` 및 `ecr:GetDownloadUrlForLayer`를 추가합니다. Lambda를 호출하는 보안 주체가 `ecr:getRepositoryPolicy` 및 `ecr:setRepositoryPolicy` 권한을 갖고 있는 경우에만 Lambda가 이러한 권한을 추가할 수 있습니다.

Amazon ECR 리포지토리 권한을 보거나 편집하려면 Amazon Elastic Container Registry 사용 설명서의 프라이빗 리포지토리 정책 설명 설정의 지침을 따르면 됩니다.

Amazon ECR 교차 계정 권한

동일한 리전의 다른 계정은 사용자 계정이 소유한 컨테이너 이미지를 사용하는 함수를 생성할 수 있습니다. 다음 예제에서 Amazon ECR 리포지토리 권한 정책은 계정 번호 123456789012에 대한 액세스 권한을 부여하는 다음 문이 필요합니다.

- CrossAccountPermission - 계정 123456789012가 이 ECR 리포지토리의 이미지를 사용하는 Lambda 함수를 생성하고 업데이트하도록 허용합니다.
- LambdaECRImageCrossAccountRetrievalPolicy - Lambda는 장기간 호출되지 않는 경우 결국 함수의 상태를 비활성으로 설정합니다. 이 문은 Lambda가 123456789012가 소유한 함수를 대신하여 최적화 및 캐싱을 위해 컨테이너 이미지를 검색할 수 있도록 하는 데 필요합니다.

Example - 리포지토리에 크로스 계정 권한 추가

```
{
  "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:*"
```

```
        }
      }
    }
  ]
}
```

여러 계정에 대한 액세스 권한을 부여하려면 `CrossAccountPermission` 정책의 보안 주체 목록과 `LambdaECRImageCrossAccountRetrievalPolicy`의 조건 평가 목록에 계정 ID를 추가합니다.

AWS 조직의 여러 계정으로 작업하는 경우 ECR 권한 정책에서 각 계정 ID를 열거하는 것이 좋습니다. 이 접근 방식은 IAM 정책에서 제한된 권한을 설정하는 AWS 보안 모범 사례와 일치합니다.

Lambda 권한 외에도 함수를 생성하는 사용자 또는 역할에는 `BatchGetImage` 및 `GetDownloadUrlForLayer` 권한도 있어야 합니다.

함수 수명 주기

새 컨테이너 이미지 또는 업데이트된 컨테이너 이미지를 업로드하면 함수가 호출을 처리하기 전에 Lambda가 이미지를 최적화합니다. 최적화 프로세스에는 몇 초가 걸릴 수 있습니다. 이 프로세스가 완료될 때까지 함수는 `Pending` 상태로 유지됩니다. 그런 다음 함수가 `Active` 상태로 전환됩니다. 상태가 `Pending`인 동안 함수를 호출할 수 있지만 함수에 대한 다른 작업은 실패합니다. 이미지 업데이트가 진행되는 동안 발생하는 호출은 이전 이미지의 코드를 실행합니다.

함수가 여러 주 동안 호출되지 않으면 Lambda는 최적화된 버전을 회수하고 함수는 `Inactive` 상태로 전환됩니다. 함수를 다시 활성화하려면 함수를 호출해야 합니다. Lambda는 첫 번째 호출을 거부하고, Lambda가 이미지를 다시 최적화할 때까지 함수는 `Pending` 상태가 됩니다. 그런 다음 함수는 `Active` 상태로 돌아갑니다.

Lambda는 Amazon ECR 리포지토리에서 연관된 컨테이너 이미지를 주기적으로 가져옵니다. 해당 컨테이너 이미지가 더 이상 Amazon ECR에 존재하지 않거나 권한이 취소되면, 함수가 `Failed` 상태에 들어가고 모든 함수 호출에 대해 Lambda가 실패를 반환합니다.

Lambda API를 사용하여 함수 상태에 대한 정보를 가져올 수 있습니다. 자세한 내용은 Lambda 함수 상태 단원을 참조하십시오.

Lambda 함수 간접 호출 방법 이해

Lambda 함수를 배포한 후에는 여러 가지 방법으로 함수를 간접적으로 호출할 수 있습니다.

- Lambda 콘솔 - Lambda 콘솔을 사용하여 함수를 간접적으로 호출하는 테스트 이벤트를 빠르게 생성합니다.
- AWS SDK - AWS SDK를 사용하여 프로그래밍 방식으로 함수를 간접적으로 호출합니다.
- 호출 API - Lambda 호출 API를 사용하여 함수를 직접 호출할 수 있습니다.
- AWS Command Line Interface(AWS CLI) – `aws lambda invoke` AWS CLI 명령을 사용하여 명령줄에서 직접 함수를 간접적으로 호출합니다.
- 함수 URL HTTP(S) 엔드포인트 - 함수 URL을 사용하여 함수를 간접적으로 호출하는 데 사용할 수 있는 전용 HTTP(S) 엔드포인트를 생성합니다.

이러한 모든 메서드는 함수를 직접 호출하는 방법입니다. Lambda에서 일반적인 사용 사례는 애플리케이션의 다른 곳에서 발생하는 이벤트를 기반으로 함수를 간접적으로 호출하는 것입니다. 일부 서비스는 새로운 이벤트가 발생할 때마다 Lambda 함수를 간접적으로 호출할 수 있습니다. 이를 트리거라고 합니다. 스트림 및 대기열 기반 서비스의 경우 Lambda는 레코드 배치와 함께 함수를 간접적으로 호출합니다. 이를 이벤트 소스 매핑이라고 합니다.

함수를 호출할 때 동기식으로 호출할 것인지 비동기식으로 호출할 것인지 선택할 수 있습니다. 동기식 호출의 경우 함수가 이벤트를 처리하여 응답을 반환하기를 기다립니다. 비동기식 호출의 경우, Lambda는 처리를 위해 이벤트를 대기열에 저장하고 즉시 응답을 반환합니다. 호출 API의 `InvocationType` 요청 파라미터에 따라 Lambda가 함수를 간접적으로 호출하는 방식이 결정됩니다. `RequestResponse` 값은 동기식 호출을 나타내고 `Event` 값은 비동기 호출을 나타냅니다.

함수 간접 호출에서 오류가 발생하는 경우 동기식 호출의 경우에는 응답에서 오류 메시지를 확인하고 간접 호출을 수동으로 다시 시도합니다. 비동기 호출의 경우 Lambda는 재시도를 자동으로 처리하고 간접 호출 레코드를 대상에 보낼 수 있습니다.

동기식 호출

함수를 동기식으로 호출하는 경우 Lambda는 함수를 실행하고 응답을 기다립니다. 함수가 완료되면 Lambda는 호출된 함수의 버전과 같은 추가 데이터가 포함된 응답을 함수의 코드에서 반환합니다. AWS CLI를 사용해 동기식으로 함수를 호출하려면 `invoke` 명령을 사용하세요.

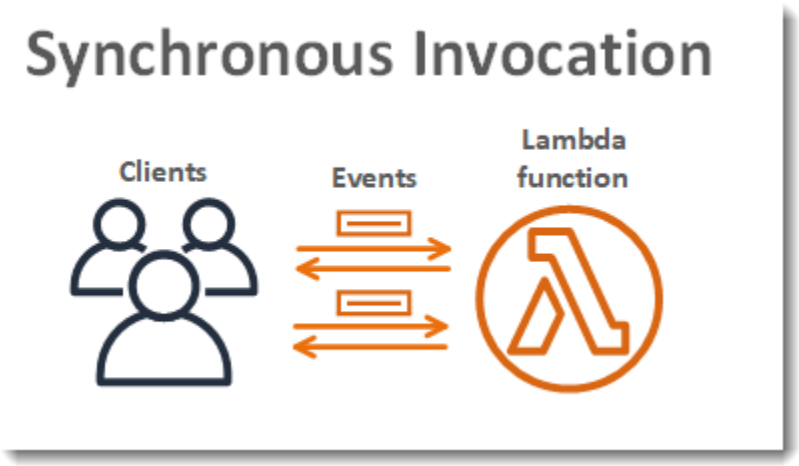
```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{ "key": "value" }' response.json
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

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

다음 다이어그램은 Lambda 함수를 동기적으로 호출하는 클라이언트를 보여줍니다. Lambda는 이벤트를 함수로 직접 보내고 호출자에게 함수의 응답을 다시 보냅니다.



`payload`은 JSON 형식의 이벤트를 포함하는 문자열입니다. AWS CLI가 함수에서 응답을 기록하는 파일의 이름은 `response.json`입니다 . 함수가 개체 또는 오류를 반환하는 경우 응답 본문은 JSON 형식의 개체 또는 오류입니다. 함수가 오류 없이 종료되는 경우 응답 본문은 입니다. `null`

> **Note**
>
> Lambda는 응답을 전송하기 전에 외부 확장이 완료되는 것을 기다리지 않습니다. 외부 확장은 실행 환경에서 독립 프로세스로 실행되며 함수 호출이 완료된 후에도 계속 실행됩니다. 자세한 설명은 Lambda 확장을 사용하여 Lambda 함수 보강 섹션을 참조하세요.

명령의 출력은 터미널에 표시되며 Lambda의 응답에 있는 헤더의 정보를 포함합니다. 여기에는 이벤트를 처리한 버전(별칭 사용 시 유용)과 Lambda가 반환한 상태 코드가 들어 있습니다. Lambda가 함수를 실행할 수 있었다면 함수가 오류를 반환한 경우에도 상태 코드는 200입니다.

> **Note**
>
> 제한 시간이 긴 함수의 경우 클라이언트는 응답을 기다릴 때 동기식 호출 중 연결이 해제될 수 있습니다. HTTP 클라이언트, SDK, 방화벽, 프록시 또는 운영 체제에서 제한 시간과의 장시간 연결 또는 연결 유지 설정을 감안하도록 구성합니다.

Lambda가 함수를 실행할 수 없는 경우 출력에 오류가 표시됩니다.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload value response.json
```

다음 결과가 표시됩니다:

```
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 CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - aws configure을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 LogResult이라는 함수의 my-function 필드에서 로그 ID를 검색하는 방법을 보여 줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 base64 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 my-function에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

파라미터, 헤더 및 오류의 전체 목록을 포함한 Invoke API에 대한 자세한 내용은 호출 단원을 참조하세요.

함수를 직접 호출하는 경우 오류 및 재시도에 대한 응답을 확인할 수 있습니다. AWS CLI 및 AWS SDK 역시 클라이언트 제한 시간, 조절 및 서비스 오류의 경우 자동으로 재시도합니다. 자세한 내용은 Lambda의 재시도 동작 이해 섹션을 참조하세요.

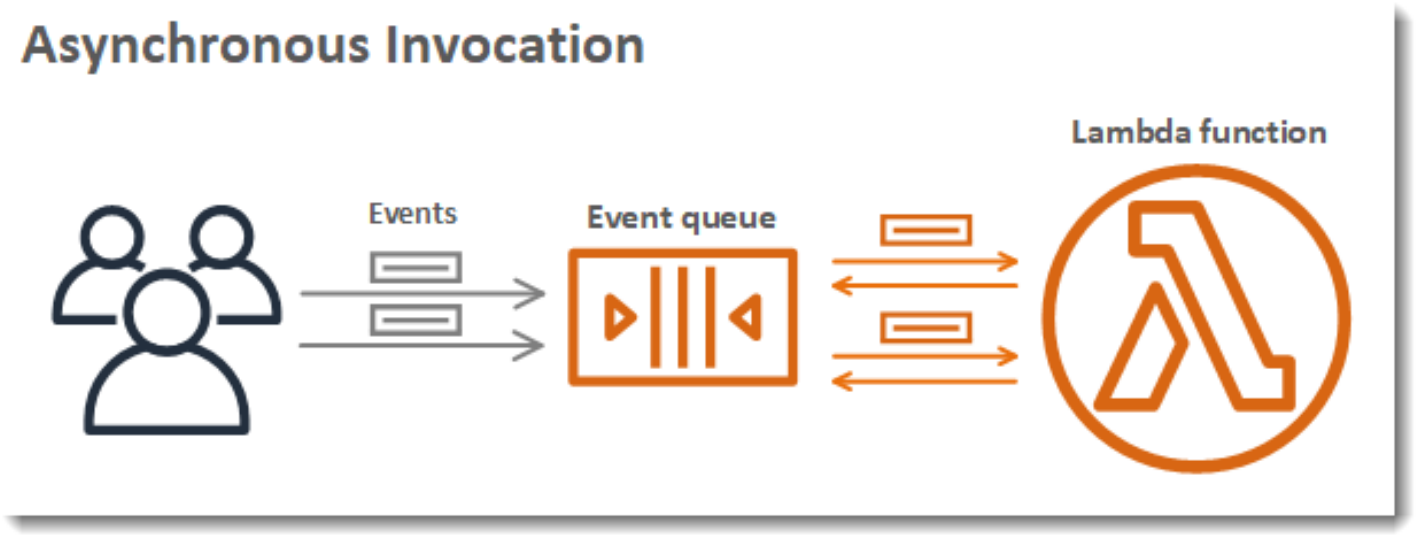
비동기식 호출

Amazon Simple Storage Service(S3), Amazon Simple Notification Service(SNS)와 같은 여러 AWS 서비스에서 함수를 비동기적으로 호출하여 이벤트를 처리합니다. 비동기적으로 함수를 호출하면 함수 코드에서 응답을 기다리지 않습니다. 이벤트를 Lambda에 전달하면 Lambda가 나머지를 처리합니다. Lambda가 오류를 처리하는 방법을 구성하고, 호출 레코드를 Amazon Simple Queue Service(Amazon SQS) 또는 Amazon EventBridge(EventBridge)와 같은 다운스트림 리소스로 전송하여 애플리케이션의 구성 요소를 하나로 연결할 수 있습니다.

Sections

Lambda가 비동기 호출을 처리하는 방법

다음 다이어그램은 Lambda 함수를 비동기적으로 호출하는 클라이언트를 보여줍니다. Lambda는 이벤트를 함수로 보내기 전에 대기열에 넣습니다.



비동기식 호출의 경우 Lambda는 이벤트를 대기열에 배치하고 추가 정보 없이 성공 응답을 반환합니다. 별도의 프로세스를 통해 대기열에서 이벤트를 읽은 후 함수로 보냅니다. 함수를 비동기식으로 호출하려면 호출 유형 파라미터를 `Event`로 설정하세요.

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
{
    "StatusCode": 202
}
```

출력 파일(`response.json`)에는 아무런 정보도 포함되어 있지 않지만 이 명령을 실행하면 여전히 생성됩니다. Lambda가 이벤트를 대기열에 추가할 수 없는 경우 명령 출력에 오류 메시지가 나타납니다.

Lambda는 함수의 비동기 이벤트 대기열을 관리하고 오류를 다시 시도합니다. 함수가 오류를 반환하면 Lambda는 함수를 두 번 더 실행하려 합니다. 이때 첫 두 시도 간에는 1분의 대기 시간이 있으며, 두 번째와 세 번째 시도 간에는 2분의 대기 시간이 있습니다. 함수 오류에는 함수의 코드가 반환하는 오류와 제한 시간과 같이 함수의 런타임에서 반환하는 오류가 포함되어 있습니다.

함수에 모든 이벤트를 처리하는 데 사용할 수 있을 정도로 동시성이 충분하지 않은 경우 추가 요청은 제한됩니다. 제한 오류(429) 및 시스템 오류(500 시리즈)의 경우 Lambda는 이벤트를 대기열로 반환하고 최대 6시간 동안 함수 재실행을 시도합니다. 재시도 간격은 첫 번째 시도 후 1초에서 최대 5분까지 기하급수적으로 증가합니다. 대기열에 많은 항목이 있는 경우 Lambda는 재시도 간격을 늘리고 대기열에서 이벤트를 읽는 속도를 줄입니다.

함수가 오류를 반환하지 않는다 하더라도 대기열 자체가 최종적으로 일관성을 갖기 때문에 Lambda에서 동일한 이벤트를 여러 번 수신할 수 있습니다. 함수가 수신 이벤트를 따라잡을 수 없는 경우 이벤트는 함수로 전송되지 않고 대기열에서 삭제될 수도 있습니다. 함수 코드가 중복 이벤트를 정상적으로 처리하는지, 모든 호출을 처리할 수 있을 만큼의 가용 동시성이 있는지 확인하세요.

대기열이 매우 길면 Lambda가 새 이벤트를 함수로 전송하기 전에 해당 이벤트가 만료될 수 있습니다. 이벤트가 만료되거나 모든 처리 시도에 실패하면 Lambda는 이벤트를 삭제합니다. 함수에 대한 오류 처리를 구성하여 Lambda가 수행하는 재시도 횟수를 줄이거나, 처리되지 않은 이벤트를 더 빠르게 폐기할 수 있습니다.

호출 레코드를 다른 서비스에 전송하도록 Lambda를 구성할 수도 있습니다. Lambda는 비동기식 호출을 위해 다음의 대상을 지원합니다. SQS FIFO 대기열과 SNS FIFO 주제는 지원되지 않습니다.

- Amazon SQS— 표준 SQS 대기열입니다.
- Amazon SNS – 표준 SNS 주제.
- AWS Lambda – Lambda 함수입니다.
- Amazon EventBridge - EventBridge 이벤트 버스입니다.

호출 레코드에는 JSON 형식의 요청 및 응답에 대한 세부 정보가 포함되어 있습니다. 처리에 성공한 이벤트와 모든 처리 시도에서 실패한 이벤트에 대해 별도의 대상을 구성할 수 있습니다. 또는 삭제된 이벤트에 대해 표준 Amazon SQS 대기열 또는 표준 Amazon SNS 주제를 배달 못한 편지 대기열로 구성할 수 있습니다. 배달 못한 편지 대기열의 경우 Lambda는 응답에 대한 세부 정보 없이 이벤트의 콘텐츠만 전송합니다.

Lambda에서 구성한 대상으로 레코드를 전송할 수 없는 경우 Amazon CloudWatch로 `DestinationDeliveryFailures` 지표를 전송합니다. 이는 구성에 Amazon SQS FIFO 대기열 또는 Amazon SNS FIFO 주제와 같이 지원되지 않는 대상 유형이 포함된 경우 발생할 수 있습니다. 전송 오류는 권한 오류 및 크기 제한으로 인해 발생할 수도 있습니다. Lambda 호출 지표에 대한 자세한 내용은 호출 지표의 내용을 참조하세요.

> **Note**
>
> 함수가 트리거되지 않도록 함수에 예약된 동시성을 0으로 설정할 수 있습니다. 비동기적으로 호출된 함수에 대해 예약된 동시성을 0으로 설정하면 Lambda는 모든 이벤트를 구성된 DLQ(Dead Letter Queue) 또는 실패 시 이벤트 대상으로 새 이벤트를 보내기 시작합니다. 예약된 동시성이 0으로 설정된 상태에서 전송된 이벤트를 처리하려면 DLQ(Dead Letter Queue) 대기열 또는 실패 시 이벤트 대상에서 이벤트를 반드시 사용해야 합니다.

비동기식 호출에 대한 오류 처리 구성

Lambda 콘솔을 사용하여 함수, 버전 또는 별칭에 대한 오류 처리 설정을 구성합니다.

오류 처리를 구성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.

3. 구성(Configuration)을 선택한 다음 비동기식 호출(Asynchronous invocation)을 선택합니다.
4. 비동기식 호출에서 편집을 선택합니다.
5. 다음 설정을 구성합니다.
   - 최대 이벤트 기간 - Lambda가 비동기 이벤트 대기열에서 이벤트를 유지하는 최대 시간입니다(최대 6시간).
   - 재시도 시도 - 함수가 오류를 반환할 때 Lambda 재시도 횟수입니다(0~2 사이).
6. 저장을 선택합니다.

호출 이벤트가 최대 기간을 초과하거나 모든 재시도 시도에 실패하면 Lambda는 해당 이벤트를 폐기합니다. 취소된 이벤트의 복사본을 보존하려면 실패한 이벤트 대상을 구성합니다.

비동기식 호출에 대한 대상 구성

비동기 호출 레코드를 유지하려면 함수에 대상을 추가합니다. 성공한 또는 실패한 간접 호출을 대상으로 전송하도록 선택할 수 있습니다. 각 함수에는 여러 대상이 있을 수 있으므로 성공 및 실패한 이벤트에 대해 별도의 대상을 구성할 수 있습니다. 대상으로 전송되는 각 레코드는 간접 호출에 대한 세부 정보가 포함된 JSON 문서입니다. 오류 처리 설정과 마찬가지로 함수, 함수의 버전 또는 별칭에 대상을 구성할 수 있습니다.

> **Note**
>
> Amazon Kinesis, Amazon DynamoDB, 자체 관리형 Apache Kafka 및 Amazon MSK와 같은 이벤트 소스 매핑 유형에 대해 실패한 간접 호출 기록을 유지할 수도 있습니다.

다음 표는 비동기 호출 레코드를 위해 지원되는 대상을 나열합니다. Lambda가 선택한 목적지로 레코드를 성공적으로 전송하려면 함수의 실행 역할에도 관련 권한이 포함되어야 합니다. 또한 표에는 각 대상 유형이 JSON 간접 호출 레코드를 수신하는 방법도 설명되어 있습니다.

대상 유형	필수 권한	대상별 JSON 형식
Amazon SQS 대기열	sqs:SendMessage	Lambda는 간접 호출 레코드를 `Message`로 목적지에 전달합니다.

대상 유형	필수 권한	대상별 JSON 형식
Amazon SNS 주제	sns:Publish	Lambda는 간접 호출 레코드를 `Message`로 목적지에 전달합니다.
Lambda 함수	InvokeFunction	Lambda는 간접 호출 레코드를 페이로드로 함수에 전달합니다.
EventBridge	events:PutEvents	• Lambda는 간접 호출 레코드를 PutEvents 호출의 `detail`로 전달합니다. • `source` 이벤트 필드의 값은 `lambda`입니다. • `detail-type` 이벤트 필드의 값은 “Lambda 함수 간접 호출 결과 - 성공” 또는 “Lambda 함수 간접 호출 결과 - 실패”입니다. • `resource` 이벤트 필드에는 함수와 대상 Amazon 리소스 이름(ARN)이 포함됩니다. • 다른 이벤트 필드의 경우 Amazon EventBridge 이벤트를 참조하세요.

다음 예제에서는 함수 오류로 인해 처리에 연속 3회 실패한 이벤트에 대한 호출 레코드를 보여 줍니다. 호출 레코드에는 이벤트, 응답 및 레코드가 전송된 이유에 대한 세부 정보가 포함되어 있습니다.

```
{
    "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"
    }
}
```

다음 단계에서는 Lambda 콘솔을 사용하여 함수의 대상을 구성하는 방법을 설명합니다.

비동기 호출 레코드의 대상 구성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 개요(Function overview)에서 대상 추가(Add destination)를 선택합니다.
4. 소스에서 비동기식 호출을 선택합니다.
5. Condition(조건)에서 다음 옵션 중에 선택합니다.
   - 실패 시 – 이벤트가 모든 처리 시도에 실패하거나 최대 기간을 초과할 때 레코드를 전송합니다.
   - 성공 시 – 함수가 비동기식 호출을 성공적으로 처리할 때 레코드를 전송합니다.
6. Destination type(대상 유형)에서 호출 레코드를 수신하는 리소스 유형을 선택합니다.
7. Destination(대상)에서 리소스를 선택합니다.
8. 저장을 선택합니다.

호출이 조건과 일치하면 Lambda는 호출에 대한 세부 정보가 포함된 JSON 문서를 대상으로 전송합니다.

대상별 JSON 형식

- Amazon SQS 및 Amazon SNS(`SnsDestination` 및 `SqsDestination`)의 경우 호출 레코드는 `Message`로 대상에 전달됩니다.
- Lambda(`LambdaDestination`)의 경우 호출 레코드는 페이로드로 함수에 전달됩니다.
- EventBridge(`EventBridgeDestination`)의 경우 호출 레코드는 PutEvents 호출의 `detail`로 전달됩니다. `source` 이벤트 필드의 값은 `lambda`입니다. `detail-type` 이벤트 필드의 값은 Lambda 함수 호출 결과 - 성공 또는 Lambda 함수 호출 결과 - 실패입니다. `resource` 이벤트 필드에는 함수와 대상 Amazon 리소스 이름(ARN)이 포함됩니다. 다른 이벤트 필드의 경우 Amazon EventBridge 이벤트를 참조하세요.

다음 예제에서는 함수 오류로 인해 처리에 연속 3회 실패한 이벤트에 대한 호출 레코드를 보여 줍니다.

Example 호출 레코드

```
{
    "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"
    }
```

```
}
```

호출 레코드에는 이벤트, 응답 및 레코드가 전송된 이유에 대한 세부 정보가 포함되어 있습니다.

목적지로 향하는 요청 추적

AWS X-Ray을 사용하면 대기열에 추가되고, Lambda 함수에서 처리되고, 대상 서비스로 전달되는 각 요청의 연결된 보기를 볼 수 있습니다. 함수 또는 함수를 호출하는 서비스에 대해 X-Ray 추적을 활성화하면 Lambda는 요청에 X-Ray 헤더를 추가하고 헤더를 대상 서비스에 전달합니다. 업스트림 서비스의 추적은 다운스트림 Lambda 함수 및 대상 서비스의 추적과 자동으로 연결되므로 전체 애플리케이션을 종합적으로 파악할 수 있습니다. 추적에 대한 자세한 내용은 AWS X-Ray로 Lambda 함수 간접 호출 시각화의 내용을 참조하세요.

비동기식 호출 구성 API

AWS CLI 또는 AWS SDK를 사용하여 비동기식 호출 설정을 관리하려면 다음과 같은 API 작업을 사용합니다.

- PutFunctionEventInvokeConfig
- GetFunctionEventInvokeConfig
- UpdateFunctionEventInvokeConfig
- ListFunctionEventInvokeConfigs
- DeleteFunctionEventInvokeConfig

AWS CLI를 사용하여 비동기식 호출을 구성하려면 `put-function-event-invoke-config` 명령을 사용합니다. 다음 예제에서는 최대 이벤트 기간이 1시간이고 재시도가 없는 함수를 구성합니다.

```
aws lambda put-function-event-invoke-config --function-name error \
--maximum-event-age-in-seconds 3600 --maximum-retry-attempts 0
```

다음 결과가 표시됩니다:

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:error:$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
```

```
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

이 put-function-event-invoke-config 명령은 함수, 버전 또는 별칭의 기존 구성을 덮어씁니다. 다른 옵션을 재설정하지 않고 옵션을 구성하려면 update-function-event-invoke-config를 사용합니다. 다음 예제에서는 이벤트를 처리할 수 없을 때 destination이라는 표준 SQS 대기열로 레코드를 전송하도록 Lambda를 구성합니다.

```
aws lambda update-function-event-invoke-config --function-name error \
--destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-2:123456789012:destination"}}'
```

다음 결과가 표시됩니다.

```
{
    "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"
        }
    }
}
```

배달 못한 편지 대기열

on-failure 대상의 대안으로, 이후 처리를 위해 폐기된 이벤트를 저장하기 위해 배달 못한 편지 대기열을 사용하여 함수를 구성할 수 있습니다. 배달 못한 편지 대기열은 이벤트가 모든 처리 시도에 실패하거나 처리되지 않고 만료될 때 사용된다는 점에서 on-failure 대상과 동일하게 작동합니다. 그러나 배달 못한 편지 대기열은 함수의 버전별 구성의 일부이므로 버전을 게시할 때 잠깁니다. On-failure 대상은 추가 대상도 지원하며, 호출 레코드에 함수의 응답에 관한 세부 정보를 포함합니다.

배달 못한 편지 대기열의 이벤트를 다시 처리하려면 Lambda 함수의 이벤트 소스로 설정하면 됩니다. 또는 수동으로 이벤트를 검색할 수도 있습니다.

배달 못한 편지 대기열에 대한 Amazon SQS 표준 대기열 또는 Amazon SNS 표준 주제를 선택할 수 있습니다. FIFO 대기열과 Amazon SNS FIFO 주제는 지원되지 않습니다. 대기열 또는 주제가 없는 경우 생성합니다. 사용 사례에 부합하는 대상 유형을 선택합니다.

- Amazon SQS 대기열 – 대기열은 실패한 이벤트를 검색될 때까지 보관합니다. Lambda 함수 또는 CloudWatch 경보와 같은 단일 엔터티가 실패한 이벤트를 처리해야 하는 경우 Amazon SQS 표준 대기열을 선택합니다. 자세한 내용은 Amazon SQS에서 Lambda 사용 단원을 참조하십시오.

  Amazon SQS 콘솔을 사용하여 대기열을 생성합니다.
- Amazon SNS 주제 – 주제는 실패한 이벤트를 한 개 이상의 대상으로 전달합니다. 실패한 이벤트에 대해 여러 엔터티가 작동할 것으로 예상되는 경우 Amazon SNS 표준 주제를 선택합니다. 예를 들어 이벤트를 이메일 주소, Lambda 함수 및/또는 HTTP 엔드포인트로 전송하도록 주제를 구성할 수 있습니다. 자세한 내용은 Amazon SNS 알림을 사용하여 Lambda 함수 간접 호출 단원을 참조하십시오.

  Amazon SNS 콘솔에서 주제를 생성합니다.

대기열 또는 주제로 이벤트를 전송하려면 함수에 추가 권한이 필요합니다. 필요한 권한이 있는 정책을 함수의 실행 역할에 추가합니다.

- Amazon SQS – sqs:SendMessage
- Amazon SNS – sns:Publish

대상 대기열 또는 주제가 고객 관리형 키로 암호화되는 경우, 실행 역할은 키의 리소스 기반 정책의 사용자여야 합니다.

대상을 생성하고 함수의 실행 역할을 업데이트한 후 배달 못한 편지 대기열을 함수에 추가합니다. 여러 함수에서 동일한 대상으로 이벤트를 전송하도록 구성할 수 있습니다.

배달 못한 편지 대기열을 구성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 비동기식 호출(Asynchronous invocation)을 선택합니다.
4. 비동기식 호출에서 편집을 선택합니다.
5. DLQ 리소스를 Amazon SQS 또는 Amazon SNS로 설정합니다.
6. 대상 대기열 또는 주제를 선택합니다.

7. 저장을 선택합니다.

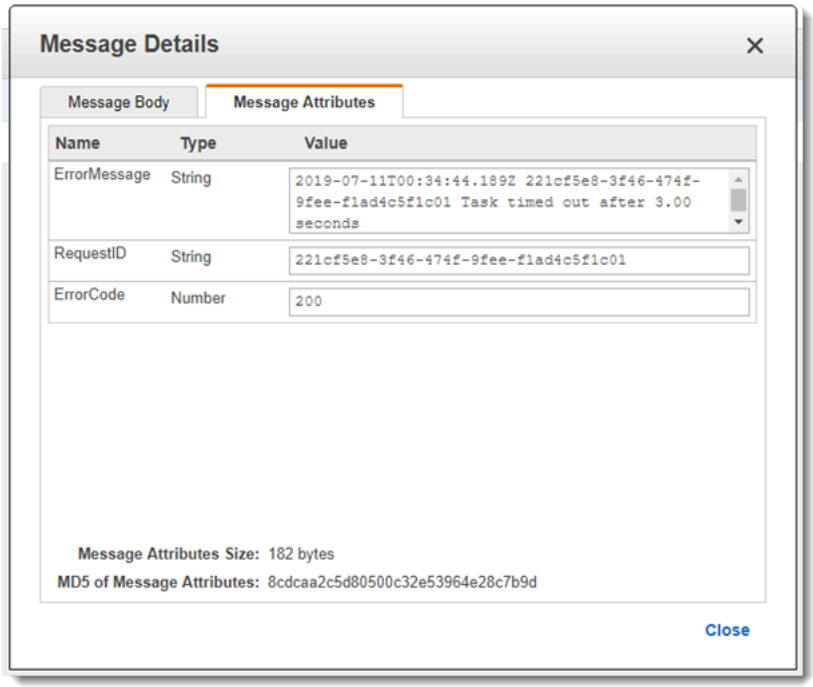
AWS CLI를 이용해 배달 못한 편지 대기열을 구성하려면 `update-function-configuration` 명령을 사용하세요.

```
aws lambda update-function-configuration --function-name my-function \
--dead-letter-config TargetArn=arn:aws:sns:us-east-2:123456789012:my-topic
```

Lambda는 이벤트를 속성 관련 추가 정보와 함께 배달 못한 편지 대기열에 있는 그대로 보냅니다. 이 정보를 사용해 함수가 반환한 오류를 식별하거나 이벤트와 로그 또는 AWS X-Ray 트레이스의 상관 관계를 알 수 있습니다.

배달 못한 편지 대기열 메시지 속성

- RequestID(문자열) – 호출 요청의 ID입니다. 요청 ID는 함수 로그에 표시됩니다. X-Ray SDK를 사용하여 트레이스의 속성에 대한 요청 ID를 기록할 수도 있습니다. 그런 다음 X-Ray 콘솔에서 요청 ID로 트레이스를 검색할 수 있습니다.
- ErrorCode(숫자) – HTTP 상태 코드입니다.
- ErrorMessage(문자열) – 오류 메시지의 첫 1KB입니다.



Lambda는 배달 못한 편지 대기열로 메시지를 전송할 수 없는 경우 이벤트를 삭제하고 DeadLetterErrors 지표를 내보냅니다. 이것은 권한이 없는 경우나 메시지 총 크기가 대상 대기열이나 주제의 한도를 초과하는 경우에 발생할 수 있습니다. 예를 들어 본문의 크기가 256KB에 가까운 Amazon SNS 알림이 함수를 트리거하여 오류가 발생한다고 가정합니다. 이 경우 Amazon SNS가 추가하는 이벤트 데이터가 Lambda가 추가하는 속성과 결합되어 메시지가 DLQ(Dead Letter Queue)에서 허용하는 최대 크기를 초과하게 될 수 있습니다.

Amazon SQS를 이벤트 소스로 사용하는 경우, Lambda 함수가 아닌 Amazon SQS 대기열 자체에 배달 못한 편지 대기열을 구성합니다. 자세한 내용은 Amazon SQS에서 Lambda 사용 섹션을 참조하세요.

Lambda가 스트림 및 대기열 기반 이벤트 소스의 레코드를 처리하는 방법

이벤트 소스 매핑은 스트림 및 대기열 기반 서비스에서 항목을 읽고 레코드 배치로 함수를 간접적으로 호출하는 Lambda 리소스입니다. 다음 서비스는 이벤트 소스 매핑을 사용하여 Lambda 함수를 간접적으로 호출합니다.

- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka(Amazon MSK)
- 자체 관리형 Apache Kafka
- Amazon Simple Queue Service(Amazon SQS)
- Amazon DocumentDB(MongoDB 호환) (Amazon DocumentDB)

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

이벤트 소스 매핑과 직접 트리거의 차이점

일부 AWS 서비스는 트리거를 사용하여 직접 Lambda 함수를 간접적으로 호출할 수 있습니다. 이러한 서비스는 Lambda로 이벤트를 푸시하고, 지정된 이벤트가 발생하면 함수가 즉시 간접적으로 호출됩니다. 트리거는 개별 이벤트와 실시간 처리에 적합합니다. Lambda 콘솔을 사용하여 트리거를 생성하면 콘솔은 해당 AWS 서비스와 상호 작용하여 서비스에 대한 이벤트 알림을 구성합니다. 트리거는 실제로 Lambda가 아니라 이벤트를 생성하는 서비스에 의해 저장되고 관리됩니다. 다음은 트리거를 사용하여 Lambda 함수를 간접적으로 호출하는 서비스의 몇 가지 예입니다.

- Amazon Simple Storage Service(S3): 버킷에서 객체가 생성, 삭제 또는 수정될 때 함수를 간접적으로 호출합니다. 자세한 내용은 자습서: Amazon S3 트리거를 사용하여 Lambda 함수 호출 단원을 참조하십시오.

- Amazon Simple Notification Service(SNS): SNS 주제에 메시지가 게시될 때 함수를 간접적으로 호출합니다. 자세한 내용은 자습서: Amazon Simple Notification Service에서 AWS Lambda 사용 단원을 참조하십시오.
- Amazon API Gateway: 특정 엔드포인트에 대한 API 요청 시 함수를 간접적으로 호출합니다. 자세한 내용은 Amazon API Gateway 엔드포인트를 사용하여 간접적으로 Lambda 함수 호출 단원을 참조하십시오.

이벤트 소스 매핑은 Lambda 서비스 내에서 생성되고 관리되는 Lambda 리소스입니다. 이벤트 소스 매핑은 대기열의 대용량 스트리밍 데이터 또는 메시지를 처리하도록 설계되었습니다. 스트림이나 대기열의 레코드를 일괄적으로 처리하는 것이 레코드를 개별적으로 처리하는 것보다 더 효율적입니다.

일괄 처리 동작

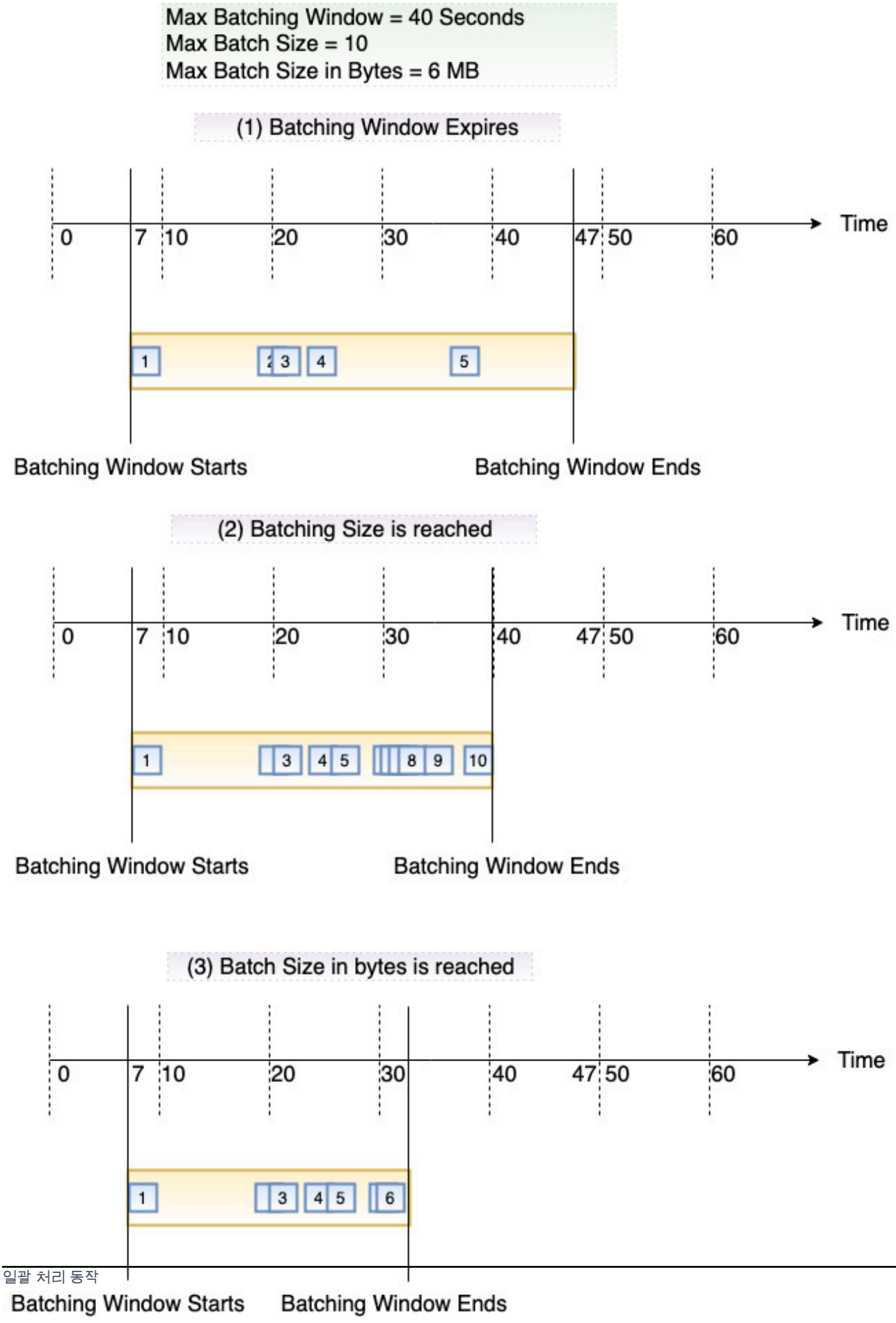
기본적으로 이벤트 소스 매핑은 레코드를 일괄 처리하고 Lambda는 이를 단일 페이로드로 함수에 전송합니다. 일괄 처리 동작을 미세 조정하려면 배치 기간(MaximumBatchingWindowInSeconds)과 배치 크기(BatchSize)를 구성합니다. 일괄 처리 기간은 레코드를 단일 페이로드로 수집할 최대 기간입니다. 배치 크기는 단일 배치의 최대 레코드 수입니다. Lambda는 다음 세 가지 기준 중 하나에 부합할 때 함수를 호출합니다.

- 일괄 처리 기간이 최댓값에 도달합니다. 기본 일괄 처리 기간 동작은 특정 이벤트 소스에 따라 다릅니다.
  - Kinesis, DynamoDB 및 Amazon SQS 이벤트 소스의 경우: 기본 일괄 처리 기간은 0초입니다. 즉, Lambda는 배치 크기가 충족되거나 페이로드 크기 제한에 도달한 경우에만 함수로 배치를 전송합니다. 배치 작업 기간을 설정하려면 `MaximumBatchingWindowInSeconds`를 구성합니다. 이 파라미터를 0~300초 범위에서 초 단위로 설정할 수 있습니다. 일괄 처리 기간을 구성하는 경우, 이전 함수 간접 호출이 완료되는 즉시 다음 기간이 시작됩니다.
  - Amazon MSK, 자체 관리형 Apache Kafka 및 Amazon MQ, Amazon DocumentDB 이벤트 소스의 경우 기본 일괄 처리 시간은 500ms입니다. `MaximumBatchingWindowInSeconds`는 0초에서 300초 사이의 값을 초 단위로 구성할 수 있습니다. 일괄 처리 기간은 첫 번째 레코드가 도착하는 즉시 시작됩니다.

> **Note**
>
> `MaximumBatchingWindowInSeconds`는 초 단위로만 변경할 수 있기 때문에 변경한 후에는 500ms 기본 배치 기간으로 되돌릴 수 없습니다. 기본 일괄 처리 기간을 복원하려면 새 이벤트 소스 매핑을 생성해야 합니다.

- 배치 크기가 충족됩니다. 최소 배치 크기는 1입니다. 기본 및 최대 배치 크기는 이벤트 소스에 따라 다릅니다. 이러한 값에 대한 자세한 내용은 `CreateEventSourceMapping` API 작업에 대한 BatchSize 사양을 참조하세요.
- 페이로드 크기가 6MB에 도달합니다. 이 한도는 수정할 수 없습니다.

다음 다이어그램은 이 세 가지 조건을 보여줍니다. 일괄 처리 기간이 `t = 7`초에서 시작한다고 가정합니다. 첫 번째 시나리오에서는 5개의 레코드를 누적한 후 일괄 처리 기간이 `t = 47`초로 2차 최댓값인 40에 도달합니다. 두 번째 시나리오에서는 일괄 처리 기간이 만료되기 전에 배치 크기가 10에 도달하므로 일괄 처리 기간이 일찍 종료됩니다. 세 번째 시나리오에서는 일괄 처리 기간이 만료되기 전에 최대 페이로드 크기에 도달하므로 일괄 처리 기간이 일찍 종료됩니다.


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
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

이벤트 소스 매핑 API

AWS Command Line Interface(AWS CLI) 또는 AWS SDK를 사용하여 이벤트 소스를 관리하려면 다음 API 작업을 사용할 수 있습니다.

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Amazon DynamoDB에서 AWS Lambda 사용

> **Note**
>
> Lambda 함수 이외의 대상으로 데이터를 전송하거나 데이터를 전송하기 전에 데이터를 보강하려는 경우 Amazon EventBridge 파이프를 참조하세요.

AWS Lambda 함수를 사용하여 Amazon DynamoDB 데이터 스트림의 레코드를 처리할 수 있습니다. DynamoDB Streams를 사용하여 DynamoDB 테이블이 업데이트될 때마다 추가 작업을 수행하는 Lambda 함수를 트리거할 수 있습니다.

Lambda는 스트림에서 레코드를 읽고 스트림 레코드를 포함한 이벤트와 동기적으로 함수를 호출합니다. Lambda는 배치의 레코드를 읽고 함수를 호출하여 배치의 레코드를 처리합니다.

Sections

- 예제 이벤트
- 폴링 및 배치 처리 스트림
- 폴링 및 스트리밍 시작 위치
- DynamoDB Streams 내 샤드의 동시 리더
- 실행 역할 권한
- 권한 추가 및 이벤트 소스 매핑 생성
- 오류 처리

- Amazon CloudWatch 측정치
- 시간 범위
- 배치 항목 실패 보고
- Amazon DynamoDB Streams 구성 파라미터
- 자습서: Amazon DynamoDB Streams와 함께 AWS Lambda 사용
- 샘플 함수 코드
- DynamoDB 애플리케이션을 위한 AWS SAM 템플릿

예제 이벤트

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"
    }
  ]}
```

폴링 및 배치 처리 스트림

Lambda는 초당 4회의 기본 속도로 레코드에 대해 DynamoDB 스트림의 샤드를 폴링합니다. 레코드를 사용할 수 있으면 Lambda가 함수를 호출하고 결과를 기다립니다. 처리가 성공하면 Lambda가 레코드를 더 받을 때까지 폴링을 재개합니다.

기본적으로, Lambda는 레코드가 사용 가능하게 되는 즉시 함수를 호출합니다. Lambda가 이벤트 소스에서 읽는 배치에 하나의 레코드만 있는 경우, Lambda는 함수에 하나의 레코드만 전송합니다. 소수의 레코드로 함수를 호출하는 것을 피하려면 일괄 처리 기간을 구성하여 이벤트 소스가 최대 5분 동안 레코드를 버퍼링하도록 지정할 수 있습니다. 함수를 호출하기 전에 Lambda는 전체 배치가 수집되거나, 일괄 처리 기간이 만료되거나, 배치가 페이로드 한도인 6MB에 도달할 때까지 이벤트 소스에서 레코드를 계속 읽습니다. 자세한 내용은 일괄 처리 동작 단원을 참조하십시오.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

DynamoDB 데이터 스트림의 한 샤드와 하나 이상의 Lambda 간접 호출을 동시에 처리하도록 ParallelizationFactor 설정을 구성합니다. Lambda가 병렬화 계수를 통해 샤드에서 폴링하는 동시 배치의 수는 1(기본값)부터 10까지 지정할 수 있습니다. 샤드당 동시 배치 수를 늘려도 Lambda는 항목(파티션 및 정렬 키) 수준에서 순차적인 처리를 계속 보장합니다.

폴링 및 스트리밍 시작 위치

이벤트 소스 매핑 생성 및 업데이트 중 스트림 폴링은 최종적으로 일관됩니다.

- 이벤트 소스 매핑 생성 중 스트림에서 이벤트 폴링을 시작하는 데 몇 분 정도 걸릴 수 있습니다.
- 이벤트 소스 매핑 업데이트 중 스트림에서 이벤트 폴링을 중지했다가 다시 시작하는 데 몇 분 정도 걸릴 수 있습니다.

이 동작은 스트림의 시작 위치로 `LATEST`를 지정하면 이벤트 소스 매핑이 생성 또는 업데이트 중에 이벤트를 놓칠 수 있음을 의미합니다. 누락된 이벤트가 없도록 스트림 시작 위치를 `TRIM_HORIZON`으로 지정하세요.

DynamoDB Streams 내 샤드의 동시 리더

글로벌 테이블이 아닌 단일 리전 테이블의 경우 최대 2개의 Lambda 함수가 동시에 동일 DynamoDB Streams 샤드에서 읽기 작업을 수행하도록 설계할 수 있습니다. 이 제한을 초과하면 요청 병목이 발생할 수 있습니다. 글로벌 테이블의 경우 요청 제한을 피하기 위해 동시 함수 수를 1로 제한하는 것이 좋습니다.

실행 역할 권한

AWSLambdaDynamoDBExecutionRole AWS 관리형 정책에는 Lambda가 DynamoDB 스트림에서 읽는 데 필요한 권한이 포함되어 있습니다. 함수의 실행 역할에 이 관리형 정책을 추가합니다.

실패한 배치의 레코드를 표준 SQS 대기열 또는 표준 SNS 주제로 보내려면 함수에 추가 권한이 필요합니다. 다음과 같이 각 대상 서비스에는 서로 다른 권한이 필요합니다.

- Amazon SQS – sqs:SendMessage
- Amazon SNS – sns:Publish

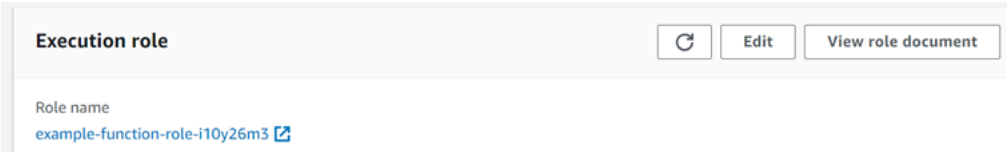
권한 추가 및 이벤트 소스 매핑 생성

이벤트 소스 매핑을 생성하여 Lambda가 스트림의 레코드를 Lambda 함수로 전송하도록 지시합니다. 여러 이벤트 소스 매핑을 생성하여 여러 Lambda 함수로 동일한 데이터를 처리하거나, 단일 함수로 여러 스트림의 항목을 처리할 수 있습니다.

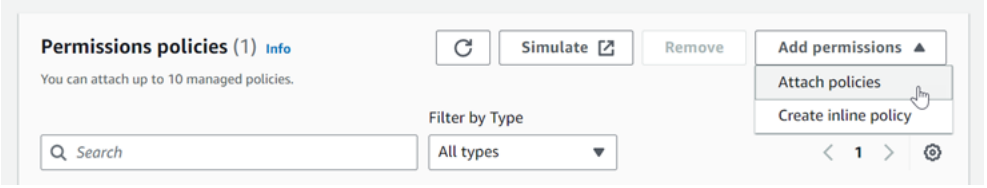
DynamoDB 스트림에서 읽도록 함수를 구성하려면 AWSLambdaDynamoDBExecutionRole AWS 관리형 정책을 실행 역할에 연결한 다음 DynamoDB 트리거를 생성합니다.

권한 추가 및 트리거 생성

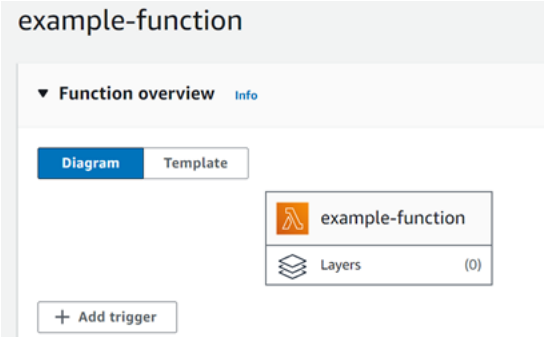
1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 권한(Permissions)을 선택합니다.
4. 역할 이름에서 실행 역할에 대한 링크를 선택합니다. 이 링크를 클릭하면 IAM 콘솔에서 역할이 열립니다.



5. 권한 추가를 선택하고 정책 연결을 선택합니다.



6. 검색 필드에 `AWSLambdaDynamoDBExecutionRole`를 입력합니다. 실행 역할에 이 정책을 추가합니다. 함수가 DynamoDB 스트림에서 읽는 데 필요한 권한을 포함하는 AWS관리형 정책입니다. 이 정책에 대한 자세한 내용은 AWS 관리형 정책 참조의 AWSLambdaDynamoDBExecutionRole을 참조하세요.
7. Lambda 콘솔에서 함수로 돌아갑니다. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.



8. 트리거 유형을 선택합니다.
9. 필요한 옵션을 구성한 다음 추가를 선택합니다.

Lambda는 DynamoDB 이벤트 소스에 대해 다음 옵션을 지원합니다.

이벤트 소스 옵션

- DynamoDB 테이블 – 레코드를 읽을 DynamoDB 테이블입니다.
- 배치 크기(Batch size) - 각 배치에서 함수에 보낼 레코드 수입니다(최대 10,000개). Lambda는 한 번의 호출로 배치의 모든 레코드를 함수에 전달합니다. 단, 이벤트의 총 크기가 동기식 호출에 대한 페이로드 한도(6MB)를 초과하지 않아야 합니다.
- 배치 기간(Batch window) - 함수를 호출하기 전에 레코드를 수집할 최대 기간(단위: 초)를 지정합니다.
- 시작 위치 – 새 레코드만, 또는 기존의 모든 레코드를 처리합니다.
  - 최신 – 스트림에 추가된 새 레코드를 처리합니다.
  - 수평 트리밍 – 스트림의 모든 레코드를 처리합니다.

  기존 레코드 처리 후 함수는 캐치업되고 새 레코드를 계속 처리합니다.
- On-failure destination(실패 시 대상) – 처리할 수 없는 레코드에 대한 표준 SQS 대기열 또는 표준 SNS 주제입니다. 너무 오래되었거나 모든 재시도를 다 사용한 레코드 배치를 폐기할 때, Lambda는 해당 배치에 대한 세부 정보를 대기열 또는 주제로 보냅니다.

- Retry attempts(재시도) - 함수가 오류를 반환할 때 Lambda에서 재시도하는 최대 횟수입니다. 이는 배치가 함수에 도달하지 않은 제한 또는 서비스 오류에 적용되지 않습니다.
- Maximum age of record(최대 레코드 사용 기간) – Lambda에서 함수로 보내는 최대 레코드 사용 기간입니다.
- Split batch on error(오류 시 배치 분할) – 함수에서 오류를 반환하면 재시도하기 전에 배치를 두 개로 분할합니다. 원래 배치 크기 설정은 변경되지 않습니다.
- Concurrent batches per shard(샤드당 동시 배치) – 동일한 샤드의 여러 배치를 동시에 처리합니다.
- 활성화 – 이벤트 소스 매핑을 활성화하려면 true로 설정합니다. 레코드 처리를 중지하려면 false로 설정합니다. Lambda는 마지막으로 처리된 레코드를 추적하여 매핑이 다시 활성화되면 해당 지점부터 처리를 다시 시작합니다.

> **Note**
>
> DynamoDB 트리거의 일부로 Lambda에서 호출한 GetRecords API 호출에 대해서는 요금이 부과되지 않습니다.

나중에 이벤트 소스 구성을 관리하기 위해 디자이너에서 트리거를 선택합니다.

오류 처리

DynamoDB 이벤트 소스 매핑에 대한 오류 처리는 오류가 함수 간접 호출 전에 발생하는지 아니면 함수 간접 호출 중에 발생하는지 여부에 따라 달라집니다.

- 간접 호출 전: Lambda 이벤트 소스 매핑이 제한 또는 기타 문제로 인해 함수를 간접적으로 호출할 수 없는 경우 레코드가 만료되거나 이벤트 소스 매핑에 구성된 최대 기간(MaximumRecordAgeInSeconds)을 초과할 때까지 재시도합니다.
- 간접 호출 중: 함수가 간접적으로 호출되었지만 오류가 반환되는 경우 Lambda는 레코드가 만료되거나 최대 기간(MaximumRecordAgeInSeconds)을 초과하거나 구성된 재시도 할당량(MaximumRetryAttempts)에 도달할 때까지 재시도합니다. 함수 오류의 경우 실패한 배치를 두 개의 작은 배치로 분할하여 잘못된 레코드를 격리하고 시간 초과를 방지하는 BisectBatchOnFunctionError를 구성할 수도 있습니다. 배치를 분할할 때는 재시도 할당량이 소모되지 않습니다.

오류 처리에서 실패를 측정하는 경우 Lambda는 레코드를 폐기하고 스트림에서 배치 처리를 계속합니다. 기본 설정을 사용하는 경우 이는 잘못된 레코드가 영향을 받은 샤드에 대한 처리를 최대 1일 동안

차단할 수 있음을 의미합니다. 이를 방지하려면 함수의 이벤트 소스 매핑을 사용자의 사례에 적합한 최대 레코드 사용 기간 및 합당한 재시도 횟수로 구성합니다.

실패한 간접 호출에 대한 대상 구성

실패한 이벤트 소스 매핑 간접 호출 기록을 보관하려면 함수의 이벤트 소스 매핑에 대상을 추가합니다. 대상으로 전송된 각 레코드는 실패한 간접 호출에 대한 메타데이터가 포함된 JSON 문서입니다. 모든 Amazon SNS 주제 또는 Amazon SQS 대기열을 대상으로 구성할 수 있습니다. 실행 역할에 대상에 대한 권한이 있어야 합니다.

- SQS 대상의 경우: sqs:SendMessage
- SNS 대상의 경우: sns:Publish

이 콘솔을 사용하여 장애 시 대상을 구성하려면 다음 단계를 따르세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 개요(Function overview)에서 대상 추가(Add destination)를 선택합니다.
4. 소스의 경우 이벤트 소스 매핑 간접 호출을 선택합니다.
5. 이벤트 소스 매핑의 경우 이 함수에 대해 구성된 이벤트 소스를 선택합니다.
6. 조건의 경우 실패 시를 선택합니다. 이벤트 소스 매핑 간접 호출의 경우 이 조건만 수락됩니다.
7. 대상 유형의 경우 Lambda가 간접 호출 레코드를 전송할 대상 유형을 선택합니다.
8. Destination(대상)에서 리소스를 선택합니다.
9. 저장을 선택합니다.

AWS Command Line Interface(AWS CLI)를 사용하여 장애 시 대상을 구성할 수도 있습니다. 예를 들어, 다음 create-event-source-mapping 명령은 SQS 장애 시 대상이 있는 이벤트 소스 매핑을 `MyFunction`에 추가합니다.

```
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"}}'
```

다음 update-event-source-mapping 명령은 두 번의 재시도 시도 후 또는 레코드가 1시간 이상 지난 경우 SNS 대상으로 실패한 간접 호출 레코드를 전송하도록 이벤트 소스 매핑을 업데이트합니다.

```
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"}}'
```

업데이트된 설정은 비동기식으로 적용되며 프로세스가 완료된 후에야 출력에 반영됩니다. get-event-source-mapping 명령을 사용하여 현재 상태를 봅니다.

대상을 제거하려면 `destination-config` 파라미터의 인수로 빈 문자열을 제공합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

다음 예제에서는 DynamoDB 스트림의 호출 레코드를 보여 줍니다.

Example 호출 레코드

```
{
    "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"
    }
}
```

이 정보를 사용하여 문제 해결을 위해 스트림에서 영향을 받은 레코드를 검색할 수 있습니다. 실제 레코드는 포함되지 않으므로 이 레코드를 처리하고 실제 레코드가 만료되고 없어지기 전에 스트림에서 해당 레코드를 검색해야 합니다.

Amazon CloudWatch 측정치

Lambda는 해당 함수가 한 배치(batch)의 레코드 처리를 완료하면 `IteratorAge` 지표를 생성합니다. 이 측정치는 처리가 완료되었을 때 배치의 마지막 레코드가 얼마나 오래되었는지를 나타냅니다. 함수가 새 이벤트를 처리하는 경우 반복기 수명을 사용하여 레코드가 추가된 후 함수에서 레코드를 처리할 때까지의 지연 시간을 추정할 수 있습니다.

반복기 수명이 증가하는 추세이면 함수에 문제가 있음을 나타낼 수 있습니다. 자세한 내용은 Lambda 함수 지표 작업 단원을 참조하십시오.

시간 범위

Lambda 함수는 연속 스트림 처리 애플리케이션을 실행할 수 있습니다. 스트림은 애플리케이션을 통해 연속적으로 흐르는 무한 데이터를 나타냅니다. 이 지속적으로 업데이트되는 입력의 정보를 분석하기 위해 시간의 항으로 정의된 윈도우를 사용하여 포함된 레코드를 바인딩할 수 있습니다.

텀블링 기간은 일정한 간격으로 시작되고 끝나는 고유한 시간대입니다. 기본적으로 Lambda 호출은 상태 비저장입니다. 즉, 외부 데이터베이스 없이는 여러 연속 호출에서 데이터를 처리하는 데 사용할 수 없습니다. 그러나 텀블링 기간을 활성화하면 호출 간에 상태를 유지할 수 있습니다. 이 상태에는 현재 윈도우에 대해 이전에 처리된 메시지의 집계 결과가 포함됩니다. 상태는 샤드당 최대 1MB가 될 수 있습니다. 이 크기를 초과하면 Lambda가 윈도우를 조기 종료합니다.

스트림의 각 레코드는 특정 윈도우에 속합니다. Lambda는 각 레코드를 한 번 이상 처리하지만 각 레코드가 한 번만 처리된다고 보장하지는 않습니다. 드물게 오류 처리와 같이 일부 레코드가 두 번 이상 처리될 수 있습니다. 레코드는 항상 처음부터 순서대로 처리됩니다. 레코드가 두 번 이상 처리되는 경우 레코드가 비순차적으로 처리될 수 있습니다.

집계 및 처리

집계 및 해당 집계의 최종 결과 처리를 위해 사용자 관리형 함수가 호출됩니다. Lambda는 윈도우 내에서 수신한 모든 레코드를 집계합니다. 이러한 레코드를 각각 별도의 호출인 여러 배치에서 받을 수 있

습니다. 각 호출은 상태를 수신합니다. 따라서 텀블링 기간을 사용할 때 Lambda 함수 응답에는 `state` 속성을 포함해야 합니다. 응답에 `state` 속성이 포함되지 않은 경우 Lambda는 이 호출이 실패한 것으로 간주합니다. 이 조건을 충족하기 위해 함수는 다음 JSON 모양의 `TimeWindowEventResponse` 객체를 반환할 수 있습니다.

Example **`TimeWindowEventResponse`** 값

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Java 함수의 경우 `Map<String, String>`을 사용하여 상태를 나타내는 것이 좋습니다.

윈도우 끝에서 `isFinalInvokeForWindow` 플래그는 이것이 최종 상태이며 처리 준비가 되었음을 나타내기 위해 `true`로 설정됩니다. 처리 후 윈도우가 완료되고 최종 호출이 완료된 다음 상태가 삭제됩니다.

윈도우 끝에서 Lambda가 집계 결과에 대한 작업을 위해 최종 처리를 사용합니다. 최종 처리는 동기식으로 호출됩니다. 성공적인 호출 후 함수가 시퀀스 번호를 검사하고 스트림 처리가 계속됩니다. 호출이 실패하면 Lambda 함수는 호출이 성공할 때까지 추가 처리를 일시 중단합니다.

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
}
```

구성

이벤트 소스 매핑을 생성하거나 업데이트할 때 텀블링 윈도우를 구성할 수 있습니다. 텀블링 윈도우를 구성하려면 윈도우를 초 단위로 지정합니다(TumblingWindowInSeconds). 다음 예제 AWS Command Line Interface(AWS CLI) 명령은 텀블링 윈도우가 120초인 스트리밍 이벤트 소스 매핑을 생성합니다. 집계 및 처리를 위해 정의된 Lambda 함수는 `tumbling-window-example-function`입니다.

```
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
```

Lambda는 레코드가 스트림에 삽입된 시간을 기준으로 텀블링 윈도우 경계를 결정합니다. 모든 레코드에는 Lambda가 경계 결정에서 사용하는 대략적인 타임스탬프가 있습니다.

텀블링 윈도우 집계는 리샤딩을 지원하지 않습니다. 샤드가 끝나면 Lambda는 윈도우가 닫힌 것으로 간주하며 자식 샤드는 새 상태로 고유한 윈도우를 시작합니다.

텀블링 윈도우는 기존 재시도 정책 `maxRetryAttempts` 및 `maxRecordAge`를 완벽하게 지원합니다.

Example Handler.py - 집계 및 처리

다음 Python 함수는 최종 상태를 집계한 다음 처리하는 방법을 보여줍니다.

```
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}
```

배치 항목 실패 보고

이벤트 소스에서 스트리밍 데이터를 사용하고 처리할 때 기본적으로 Lambda는 배치가 완전히 성공한 경우에만 배치의 가장 높은 시퀀스 번호로 체크포인트를 수행합니다. Lambda는 다른 모든 결과를 완전한 실패로 처리하고 재시도 제한까지 배치 처리를 재시도합니다. 스트림에서 배치를 처리하는 동안 부분적인 성공을 허용하려면 `ReportBatchItemFailures`를 설정합니다. 부분적인 성공을 허용하면 레코드에 대한 재시도 횟수를 줄이는 데 도움이 되지만 성공한 레코드의 재시도 가능성을 완전히 막지는 못합니다.

`ReportBatchItemFailures`를 켜려면 FunctionResponseTypes 목록에 열거형 값 **`ReportBatchItemFailures`**를 포함시킵니다. 이 목록은 함수에 대해 활성화된 응답 유형을 나타냅니다. 이벤트 소스 매핑을 생성하거나 업데이트할 때 이 목록을 구성할 수 있습니다.

보고서 구문

배치 항목 실패에 대한 보고를 구성할 때 `StreamsEventResponse` 클래스는 배치 항목 실패 목록과 함께 반환됩니다. `StreamsEventResponse` 객체를 사용하여 배치에서 첫 번째 실패한 레코드의 시퀀스 번호를 반환할 수 있습니다. 올바른 응답 구문을 사용하여 고유한 사용자 지정 클래스를 생성할 수도 있습니다. 다음 JSON 구조는 필요한 응답 구문을 보여줍니다.

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> `batchItemFailures` 어레이에 여러 항목이 포함되어 있으면 Lambda는 시퀀스 번호가 가장 낮은 레코드를 체크포인트로 사용합니다. 그런 다음 Lambda는 해당 체크포인트에서 시작하여 모든 레코드를 다시 시도합니다.

성공 및 실패 조건

Lambda는 다음 중 하나를 반환할 경우 배치를 완전한 성공으로 처리합니다.

- 비어 있는 `batchItemFailure` 목록
- null `batchItemFailure` 목록
- 비어 있는 `EventResponse`
- null `EventResponse`

Lambda는 다음 중 하나를 반환할 경우 배치를 완전한 실패로 처리합니다.

- 빈 문자열 `itemIdentifier`
- null `itemIdentifier`
- 키 이름이 잘못된 `itemIdentifier`

Lambda는 재시도 전략에 따라 실패를 재시도합니다.

배치 이등분

호출이 실패하고 `BisectBatchOnFunctionError`가 활성화되어 있으면 `ReportBatchItemFailures` 설정에 관계 없이 배치가 이등분됩니다.

부분적 배치 성공 응답이 수신되고 `BisectBatchOnFunctionError` 및 `ReportBatchItemFailures`가 모두 활성화되면 배치가 반환된 시퀀스 번호에서 이등분되고 Lambda는 나머지 레코드만 재시도합니다.

다음은 일괄적으로 실패한 메시지 ID 목록을 반환하는 함수 코드의 몇 가지 예입니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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: [] };
};
```

TypeScript를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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 구성 파라미터

모든 Lambda 이벤트 소스 유형은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 일부 파라미터만 DynamoDB Streams에 적용됩니다.

DynamoDB Streams 에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
BatchSize	N	100	최대값: 10,000
BisectBatchOnFunctionError	N	false	
DestinationConfig	N		폐기된 레코드의 표준 Amazon SQS 대기열 또는 표준 Amazon SNS 주제 대상
활성	N	true	
EventSourceArn	Y		데이터 스트림 또는 스트림 소비자의 ARN
FilterCriteria	N		Lambda 이벤트 필터링
FunctionName	Y		
FunctionResponseTypes	N		함수가 배치에서 특정 실패를 보고하도록 하려면 `FunctionResponseTypes` 에 `ReportBatchItemFailures` 값을 포함하세요. 자세

파라미터	필수	기본값	참고
			한 내용은 배치 항목 실패 보고 단원을 참조하십시오.
MaximumBatchingWindowInSeconds	N	0	
MaximumRecordAgeInSeconds	N	-1	-1은 무한을 의미하며 실패한 레코드는 레코드가 만료될 때까지 재시도됩니다. DynamoDB 스트림의 데이터 보존 한도는 24시간입니다. 최솟값: -1 최대값: 604,800
MaximumRetryAttempts	N	-1	-1(무한): 레코드가 만료될 때까지 실패한 레코드가 다시 시도됩니다 최솟값: 0 최대값: 10,000
ParallelizationFactor	N	1	최댓값: 10
StartingPosition	Y		TRIM_HIZON 또는 LATEST
TumblingWindowInSeconds	N		최솟값: 0 최댓값: 900

자습서: Amazon DynamoDB Streams와 함께 AWS Lambda 사용

이 자습서에서는 Amazon DynamoDB 스트림의 이벤트를 사용하기 위해 Lambda 함수를 생성합니다.

필수 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 생성

함수에 AWS 리소스에 액세스할 수 있는 권한을 제공하는 실행 역할을 만듭니다.

실행 역할을 만들려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.

- 신뢰할 수 있는 엔터티 – Lambda.
- 권한 – AWSLambdaDynamoDBExecutionRole.
- 역할 이름 – **lambda-dynamodb-role**.

AWSLambdaDynamoDBExecutionRole은 함수가 DynamoDB에서 항목을 읽고 CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다.

함수 생성

DynamoDB 이벤트를 처리하는 Lambda 함수를 생성하세요. 함수 코드는 수신 이벤트 데이터 중 일부를 CloudWatch Logs에 기록합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 DynamoDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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)}`);
};
```

TypeScript를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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(())

}
```

함수를 만들려면

1. 샘플 코드를 `example.js` 파일에 복사합니다.
2. 배포 패키지를 만듭니다.

```
zip function.zip example.js
```

3. `create-function` 명령을 사용해 Lambda 함수를 만듭니다.

```
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 함수 테스트

이 단계에서는 `invoke` AWS Lambda CLI 명령과 다음 샘플 DynamoDB 이벤트를 사용하여 수동으로 Lambda 함수를 호출합니다. 다음을 `input.txt`라는 파일에 복사합니다.

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"
        }
    ]
}
```

다음 invoke 명령을 실행합니다.

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

이 함수는 응답 본문에 문자열 `message`를 반환합니다.

`outputfile.txt` 파일의 출력을 확인합니다.

스트림을 활성화하여 DynamoDB 테이블 생성

스트림을 활성화하여 Amazon DynamoDB 테이블을 생성합니다.

DynamoDB 테이블을 생성하려면

1. DynamoDB 콘솔을 엽니다.
2. [Create table]을 선택합니다.
3. 다음 설정을 사용하여 테이블을 생성합니다.
   - 테이블 이름 – **`lambda-dynamodb-stream`**
   - 기본 키 – **`id`**(문자열)
4. 생성(Create)을 선택합니다.

스트림을 활성화하려면

1. DynamoDB 콘솔을 엽니다.
2. 테이블을 선택합니다.
3. lambda-dynamodb-stream 테이블을 선택합니다.

4. 내보내기 및 스트림에서 DynamoDB 스트림 세부 정보를 선택합니다.
5. 켜기를 선택합니다.
6. 보기 유형에서는 키 속성만을 선택합니다.
7. 스트림 켜기를 선택합니다.

스트림 ARN을 적어둡니다. 다음 단계에서 스트림을 Lambda 함수와 연결할 때 필요합니다. 스트림 활성화에 대한 자세한 내용은 DynamoDB 스트림을 사용하여 Table Activity 캡처를 참조하세요.

AWS Lambda에서 이벤트 소스 추가

AWS Lambda에서 이벤트 소스 매핑을 생성합니다. 이 이벤트 소스 매핑은 DynamoDB 스트림을 Lambda 함수와 연결합니다. 이 이벤트 소스 매핑을 생성하면 AWS Lambda가 스트림 폴링을 시작합니다.

다음 AWS CLI `create-event-source-mapping` 명령을 실행합니다. 명령이 실행된 후 UUID를 적어둡니다. 예를 들어, 이벤트 소스 매핑을 삭제할 때처럼 모든 명령에서 이벤트 소스 매핑을 참조하려면 이 UUID가 필요합니다.

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

이는 지정된 DynamoDB 스트림과 Lambda 함수 사이의 매핑을 생성합니다. DynamoDB 스트림을 여러 Lambda 함수와 연결하고 동일한 Lambda 함수를 여러 스트림과 연결할 수 있습니다. 하지만 Lambda 함수는 공유하는 스트림에 대해 읽기 처리량을 공유합니다.

다음 명령을 실행하여 이벤트 소스 매핑 목록을 가져올 수 있습니다.

```
aws lambda list-event-source-mappings
```

이 목록은 사용자가 생성한 모든 이벤트 소스 매핑을 반환하며 각 매핑에 대해 `LastProcessingResult`를 표시합니다. 이 필드는 문제가 있을 경우 유용한 메시지를 제공하는 데 사용됩니다. `No records processed`(AWS Lambda가 폴링을 시작하지 않았거나 스트림에 레코드가 없음을 나타냄) 및 `OK`(AWS Lambda가 레코드를 성공적으로 읽고 Lambda 함수를 호출함을 나타냄) 같은 값은 아무 문제가 없다는 점을 나타냅니다. 문제가 있는 경우 오류 메시지를 수신합니다.

이벤트 소스 매핑이 많을 경우 함수 이름 파라미터를 사용하여 결과 범위를 좁히세요.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

설정 테스트

종합적 경험을 테스트합니다. 테이블 업데이트를 수행할 때 DynamoDB는 이벤트 레코드를 스트림에 기록합니다. AWS Lambda가 스트림을 폴링하면 스트림의 새 레코드를 감지하고 이벤트를 함수에 전달하여 사용자를 대신하여 Lambda 함수를 호출합니다.

1. DynamoDB 콘솔에서 테이블에 항목을 추가, 업데이트하고 항목을 삭제합니다. DynamoDB는 이러한 작업에 대한 레코드를 스트림에 기록합니다.
2. AWS Lambda가 스트림을 폴링하고 스트림에 대한 업데이트를 감지하면 스트림에서 찾은 이벤트 데이터를 전달하여 Lambda 함수를 호출합니다.
3. 함수는 Amazon CloudWatch에서 로그를 실행하고 생성합니다. Amazon CloudWatch 콘솔에서 보고된 로그를 확인할 수 있습니다.

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

DynamoDB 테이블을 삭제하려면

1. DynamoDB 콘솔의 테이블 페이지를 엽니다.
2. 생성한 테이블을 선택합니다.

3. Delete(삭제)를 선택합니다.
4. 텍스트 상자에 **delete**를 입력합니다.
5. 테이블 삭제를 선택합니다.

샘플 함수 코드

샘플 코드는 다음 언어로 제공됩니다.

주제

- Node.js
- Java 11
- C#
- Python 3
- Go

Node.js

다음 예에서는 DynamoDB의 메시지를 처리하고 해당 메시지의 내용을 로깅합니다.

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");
};
```

샘플 코드를 압축하여 배포 패키지를 생성합니다. 지침은 .zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포 섹션을 참조하세요.

Java 11

다음 예제는 DynamoDB의 메시지를 처리하고, 그 내용을 로깅합니다. `handleRequest`는 AWS Lambda가 이벤트 데이터를 호출하고 제공하는 핸들러입니다. 해당 핸들러는 `DynamodbEvent` 라이브러리에서 사전 정의된 `aws-lambda-java-events` 클래스를 사용합니다.

Example DDB .java EventProcessor

```
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.";
    }
}
```

핸들러가 예외 없이 정상적으로 반환되면 Lambda는 레코드의 입력 배치가 성공적으로 처리된 것으로 간주하고 스트림의 새 레코드 읽기를 시작합니다. 핸들러에 예외가 발생하면 Lambda는 레코드의 입력 배치를 처리하지 않은 것으로 간주하고 동일한 레코드 배치로 함수를 다시 호출합니다.

의존성

- `aws-lambda-java-core`
- `aws-lambda-java-events`

Lambda 라이브러리 종속성으로 코드를 빌드하여 배포 패키지를 만듭니다. 지침은 .zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포 섹션을 참조하세요.

C#

다음 예제는 DynamoDB의 메시지를 처리하고, 그 내용을 로깅합니다. `ProcessDynamoEvent`는 AWS Lambda가 이벤트 데이터를 호출하고 제공하는 핸들러입니다. 해당 핸들러는 `DynamoDbEvent` 라이브러리에서 사전 정의된 `Amazon.Lambda.DynamoDBEvents` 클래스를 사용합니다.

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 프로젝트의 `Program.cs`를 위의 샘플로 바꿉니다. 지침은 .zip 파일 아카이브를 사용하여 C# Lambda 함수를 빌드 및 배포 섹션을 참조하세요.

Python 3

다음 예에서는 DynamoDB의 메시지를 처리하고 해당 메시지의 내용을 로깅합니다.

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'])))
```

샘플 코드를 압축하여 배포 패키지를 생성합니다. 지침은 Python Lambda 함수에 대한 .zip 파일 아카이브 작업 섹션을 참조하세요.

Go

다음 예에서는 DynamoDB의 메시지를 처리하고 해당 메시지의 내용을 로깅합니다.

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 build를 사용하여 실행 파일을 빌드하고 배포 패키지를 생성합니다. 지침은 .zip 파일 아카이브를 사용하여 Go Lambda 함수 배포 단원을 참조하세요.

DynamoDB 애플리케이션을 위한 AWS SAM 템플릿

AWS SAM을(를) 사용하여 이 애플리케이션을 빌드할 수 있습니다. AWS SAM 템플릿을 생성하는 방법은 AWS SAM 개발자 안내서의 AWS Serverless Application Model 템플릿 기본 사항을 참조하세요.

다음은 자습서 애플리케이션을 위한 샘플 AWS SAM 템플릿입니다. 아래 텍스트를 .yaml 파일로 복사하고 이전에 만든 ZIP 패키지 옆에 저장합니다. Handler 및 Runtime 파라미터 값은 이전 단원에서 함수를 생성할 때 사용한 것과 일치해야 합니다.

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
```

패키지 및 배포 명령을 사용하여 서버리스 애플리케이션을 패키징하고 배포하는 방법은 AWS Serverless Application Model 개발자 안내서의 서버리스 애플리케이션 배포 단원을 참조하세요.

Lambda가 Amazon Kinesis Data Streams의 레코드를 처리하는 방법

Lambda 함수를 사용하여 Amazon Kinesis 데이터 스트림의 레코드를 처리할 수 있습니다. Lambda 함수를 Kinesis Data Streams 공유 처리량 소비자(표준 반복기)에 매핑하거나 향상된 팬 아웃 기능이 있는 전용 처리량 소비자에 매핑할 수 있습니다. 표준 반복기의 경우 Lambda는 HTTP 프로토콜을 사용하여 Kinesis 스트림의 각 샤드를 폴링합니다. 이벤트 소스 매핑은 샤드의 다른 소비자와 읽기 처리량을 공유합니다.

Kinesis 데이터 스트림에 대한 자세한 내용은 Amazon Kinesis Data Streams에서 데이터 읽기를 참조하세요.

> **Note**
>
> Kinesis는 각 샤드에 대해 요금을 부과하고 향상된 팬아웃의 경우 스트림에서 읽은 데이터에 대해 요금을 부과합니다. 요금 세부 정보는 Amazon Kinesis 요금을 참조하세요.

폴링 및 배치 처리 스트림

Lambda는 데이터 스트림에서 레코드를 읽고 스트림 레코드를 포함한 이벤트와 동기적으로 함수를 호출합니다. Lambda는 배치의 레코드를 읽고 함수를 호출하여 배치의 레코드를 처리합니다. 각 배치에는 단일 샤드/데이터 스트림의 레코드가 포함됩니다.

기본적으로, Lambda는 레코드가 사용 가능하게 되는 즉시 함수를 호출합니다. Lambda가 이벤트 소스에서 읽는 배치에 하나의 레코드만 있는 경우, Lambda는 함수에 하나의 레코드만 전송합니다. 소수의 레코드로 함수를 호출하는 것을 피하려면 일괄 처리 기간을 구성하여 이벤트 소스가 최대 5분 동안 레코드를 버퍼링하도록 지정할 수 있습니다. 함수를 호출하기 전에 Lambda는 전체 배치가 수집되거나, 일괄 처리 기간이 만료되거나, 배치가 페이로드 한도인 6MB에 도달할 때까지 이벤트 소스에서 레코드를 계속 읽습니다. 자세한 내용은 일괄 처리 동작 단원을 참조하십시오.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

Kinesis 데이터 스트림의 한 샤드와 하나 이상의 Lambda 간접 호출을 동시에 처리하도록 ParallelizationFactor 설정을 구성합니다. Lambda가 병렬화 계수를 통해 샤드에서 폴링하는 동시 배치의 수는 1(기본값)부터 10까지 지정할 수 있습니다. 예를 들어 `ParallelizationFactor`를 2로 설정하는 경우 최대 100개의 Kinesis 데이터 샤드를 처리하기 위한 200번의 동시 Lambda 간접 호출을 보유할 수 있습니다(실제로 `ConcurrentExecutions` 지표의 값은 다를 수 있음). 이는 데이터 볼륨이 일시적이고 `IteratorAge`가 높을 때 처리량을 확장하는 데 도움을 줍니다. 샤드당 동시 배치 수를 높여도 Lambda는 파티션-키 수준에서의 순차 처리를 계속 보장합니다.

Kinesis 집계와 함께 `ParallelizationFactor`를 사용할 수도 있습니다. 이벤트 소스 매핑의 동작은 향상된 팬아웃을 사용하는지 여부에 따라 달라집니다.

- 향상된 팬아웃 사용 안 함: 집계된 이벤트 내의 모든 이벤트는 동일한 파티션 키를 가져야 합니다. 파티션 키도 집계된 이벤트의 파티션 키와 일치해야 합니다. 집계된 이벤트 내의 이벤트가 다른 파티션 키를 가지는 경우 Lambda는 파티션 키를 기준으로 이벤트를 순서대로 처리한다고 보장할 수 없습니다.
- 향상된 팬아웃 사용: 먼저 Lambda는 집계된 이벤트를 개별 이벤트로 디코딩합니다. 집계된 이벤트는 포함된 이벤트와 다른 파티션 키를 가질 수 있습니다. 하지만 파티션 키가 일치하지 않는 이벤트는 삭제되고 손실됩니다. Lambda는 이러한 이벤트를 처리하지 않으며 구성된 장애 대상으로 전송하지도 않습니다.

예제 이벤트

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"
        }
```

```
    ]
}
```

Lambda로 Amazon Kinesis Data Streams 레코드 처리

Lambda로 Amazon Kinesis Data Streams 레코드를 처리하려면 스트림에 대한 소비자를 생성한 다음 Lambda 이벤트 소스 매핑을 생성합니다.

데이터 스트림과 함수 구성

Lambda 함수는 데이터 스트림의 소비자 애플리케이션입니다. 이 함수는 각 샤드에서 한 번에 한 개의 레코드 배치를 처리합니다. Lambda 함수를 공유 처리량 소비자(표준 반복기)에 매핑하거나 향상된 팬아웃 기능이 있는 전용 처리량 소비자에 매핑할 수 있습니다.

- 표준 반복자: Lambda는 초당 1회의 속도로 Kinesis 스트림의 각 샤드에서 레코드를 폴링합니다. 더 많은 레코드를 사용할 수 있는 경우 Lambda는 함수가 스트림을 따라잡을 때까지 배치 처리를 유지합니다. 이벤트 소스 매핑은 샤드의 다른 소비자와 읽기 처리량을 공유합니다.
- 향상된 팬아웃: 지연 시간을 최소화하고 읽기 처리량을 최대화하려면 향상된 팬아웃으로 데이터 스트림 소비자를 생성하세요. 향상된 팬아웃 소비자는 각 샤드에 대해 전용 연결을 설정하므로 스트림에서 읽는 다른 애플리케이션에 영향을 주지 않습니다. 스트림 소비자는 HTTP/2를 사용하여 수명이 긴 연결을 통해 레코드를 Lambda에 푸시하고 요청 헤더를 압축함으로써 지연 시간을 최소화합니다. Kinesis RegisterStreamConsumer API를 사용하여 스트림 소비자를 생성할 수 있습니다.

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

다음 결과가 표시됩니다:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

함수가 레코드를 처리하는 속도를 높이려면 데이터 스트림에 샤드를 추가합니다. Lambda는 각 샤드의 레코드를 순서대로 처리합니다. 함수가 오류를 반환하면 샤드는 추가 레코드 처리를 중지합니다. 샤드가 많을수록 한 번에 더 많은 배치가 처리되므로 동시 실행에 대한 오류의 영향이 줄어듭니다.

함수가 총 동시 배치 수를 처리하기 위해 확장할 수 없는 경우 함수에 대한 할당량 증가를 요청하거나 동시성을 예약합니다.

이벤트 소스 매핑을 생성하여 Lambda 함수 호출

데이터 스트림의 레코드와 함께 Lambda 함수를 호출하려면 이벤트 소스 매핑을 생성합니다. 여러 이벤트 소스 매핑을 생성하여 여러 Lambda 함수로 동일한 데이터를 처리하거나, 단일 함수로 여러 데이터 스트림의 항목을 처리할 수 있습니다. 여러 스트림에서 항목을 처리할 때 각 배치에는 단일 샤드 또는 스트림의 레코드만 포함됩니다.

다른 AWS 계정에서 스트림의 레코드를 처리하도록 이벤트 소스 매핑을 구성할 수 있습니다. 자세한 내용은 the section called “크로스 계정 매핑”을 참조하십시오.

이벤트 소스 매핑을 생성하기 전에 Kinesis 데이터 스트림에서 읽을 수 있는 권한을 Lambda 함수에 부여해야 합니다. Lambda는 Kinesis 데이터 스트림 관련 리소스를 관리하기 위해 다음 권한이 필요합니다.

- kinesis:DescribeStream
- kinesis:DescribeStreamSummary
- kinesis:GetRecords
- kinesis:GetShardIterator
- kinesis:ListShards
- kinesis:ListStreams
- kinesis:SubscribeToShard

AWS 관리형 정책 AWSLambdaKinesisExecutionRole에는 다음 권한이 포함됩니다. 다음 절차의 설명에 따라 함수에 해당 관리형 정책을 추가합니다.

AWS Management Console

함수에 Kinesis 권한을 추가

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 구성 탭을 선택한 다음 사용 권한을 선택합니다.

3. 실행 역할 창의 역할 이름에서 함수의 실행 역할 링크를 선택합니다. 이 링크를 클릭하면 IAM 콘솔에서의 해당 역할 페이지가 열립니다.
4. 권한 정책 창에서 권한 추가를 선택한 다음 정책 연결을 선택합니다.
5. 검색 필드에 **AWSLambdaKinesisExecutionRole**를 입력합니다.
6. 정책 옆의 확인란을 선택하고 권한 추가를 선택합니다.

AWS CLI

함수에 Kinesis 권한을 추가

- 함수의 실행 역할에 AWSLambdaKinesisExecutionRole 정책을 연결하려면 다음 CLI 명령을 실행합니다.

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

함수에 Kinesis 권한을 추가

- 함수 정의에서 다음 예제처럼 Policies 속성을 추가합니다.

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs20.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

필요한 권한을 구성한 후 이벤트 소스 매핑을 생성합니다.

AWS Management Console

Kinesis 이벤트 소스 매핑 생성

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 함수 개요 창에서 트리거 추가를 선택합니다.
3. 트리거 구성에서 소스로 Kinesis를 선택합니다.
4. 이벤트 소스 매핑을 생성할 Kinesis 스트림과 필요에 따라 스트림 소비자를 선택합니다.
5. (선택 사항)이벤트 소스 매핑에 대해 배치 크기, 시작 위치 및 배치 창을 편집합니다.
6. 추가를 선택합니다.

콘솔에서 이벤트 소스 매핑을 생성하는 경우 IAM 역할에는 kinesis:ListStreams 및 kinesis:ListStreamConsumers 권한이 있어야 합니다.

AWS CLI

Kinesis 이벤트 소스 매핑 생성

- 다음 CLI 명령을 실행하여 Kinesis 이벤트 소스 매핑을 생성합니다. 사용 사례에 따라 배치 크기와 시작 위치를 직접 선택합니다.

```
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
```

배치 기간을 지정하려면 `--maximum-batching-window-in-seconds` 옵션을 추가합니다. 이 파라미터와 다른 파라미터에 대한 자세한 내용은 AWS CLI 명령 참조의 create-event-source mapping을 참조하세요.

AWS SAM

Kinesis 이벤트 소스 매핑 생성

- 함수 정의에서 다음 예제처럼 `KinesisEvent` 속성을 추가합니다.

```
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 SAM에서 Kinesis Data Streams의 이벤트 소스 매핑을 생성하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Kinesis를 참조하세요.

폴링 및 스트리밍 시작 위치

이벤트 소스 매핑 생성 및 업데이트 중 스트림 폴링은 최종적으로 일관됩니다.

- 이벤트 소스 매핑 생성 중 스트림에서 이벤트 폴링을 시작하는 데 몇 분 정도 걸릴 수 있습니다.
- 이벤트 소스 매핑 업데이트 중 스트림에서 이벤트 폴링을 중지했다가 다시 시작하는 데 몇 분 정도 걸릴 수 있습니다.

이 동작은 스트림의 시작 위치로 `LATEST`를 지정하면 이벤트 소스 매핑이 생성 또는 업데이트 중에 이벤트를 놓칠 수 있음을 의미합니다. 누락된 이벤트가 없도록 하기 위해서는 스트림 시작 위치를 `TRIM_HORIZON` 또는 `AT_TIMESTAMP`로 지정하세요.

계정 간 이벤트 소스 매핑 생성

Amazon Kinesis Data Streams는 리소스 기반 정책을 지원합니다. 따라서 다른 계정의 Lambda 함수를 사용하여 스트림으로 수집된 데이터를 하나의 AWS 계정에서 처리할 수 있습니다.

다른 AWS 계정에서 Kinesis 스트림을 사용하여 Lambda 함수에 대한 이벤트 소스 매핑을 생성하려면 리소스 기반 정책을 사용하여 스트림을 구성해서 Lambda 함수에 항목 읽기 권한을 부여합니다. 크로스 계정 액세스를 허용하도록 스트림을 구성하는 방법을 알아보려면 Amazon Kinesis Streams 개발자 안내서의 교차 계정 AWS Lambda 함수를 통한 액세스 공유를 참조하세요.

Lambda 함수에 필요한 권한을 부여하는 리소스 기반 정책으로 스트림을 구성하고 나면 이전 섹션에서 설명한 방법 중 하나를 사용하여 이벤트 소스 매핑을 생성합니다.

Lambda 콘솔을 사용하여 이벤트 소스 매핑을 생성하기로 선택한 경우 스트림의 ARN을 입력 필드에 직접 붙여넣기합니다. 스트림의 소비자를 지정하려는 경우 소비자의 ARN을 붙여넣기하면 스트림 필드가 자동으로 채워집니다.

Kinesis Data Streams 및 Lambda로 부분 배치 응답 구성

이벤트 소스에서 스트리밍 데이터를 사용하고 처리할 때 기본적으로 Lambda는 배치가 완전히 성공한 경우에만 배치의 가장 높은 시퀀스 번호로 체크포인트를 수행합니다. Lambda는 다른 모든 결과를 완전한 실패로 처리하고 재시도 제한까지 배치 처리를 재시도합니다. 스트림에서 배치를 처리하는 동안 부분적인 성공을 허용하려면 `ReportBatchItemFailures`를 설정합니다. 부분적인 성공을 허용하면 레코드에 대한 재시도 횟수를 줄이는 데 도움이 되지만 성공한 레코드의 재시도 가능성을 완전히 막지는 못합니다.

`ReportBatchItemFailures`를 켜려면 FunctionResponseTypes 목록에 열거형 값 **`ReportBatchItemFailures`**를 포함시킵니다. 이 목록은 함수에 대해 활성화된 응답 유형을 나타냅니다. 이벤트 소스 매핑을 생성하거나 업데이트할 때 이 목록을 구성할 수 있습니다.

보고서 구문

배치 항목 실패에 대한 보고를 구성할 때 `StreamsEventResponse` 클래스는 배치 항목 실패 목록과 함께 반환됩니다. `StreamsEventResponse` 객체를 사용하여 배치에서 첫 번째 실패한 레코드의 시퀀스 번호를 반환할 수 있습니다. 올바른 응답 구문을 사용하여 고유한 사용자 지정 클래스를 생성할 수도 있습니다. 다음 JSON 구조는 필요한 응답 구문을 보여줍니다.

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> batchItemFailures 어레이에 여러 항목이 포함되어 있으면 Lambda는 시퀀스 번호가 가장 낮은 레코드를 체크포인트로 사용합니다. 그런 다음 Lambda는 해당 체크포인트에서 시작하여 모든 레코드를 다시 시도합니다.

성공 및 실패 조건

Lambda는 다음 중 하나를 반환할 경우 배치를 완전한 성공으로 처리합니다.

- 비어 있는 batchItemFailure 목록
- null batchItemFailure 목록
- 비어 있는 EventResponse
- null EventResponse

Lambda는 다음 중 하나를 반환할 경우 배치를 완전한 실패로 처리합니다.

- 빈 문자열 itemIdentifier
- null itemIdentifier
- 키 이름이 잘못된 itemIdentifier

Lambda는 재시도 전략에 따라 실패를 재시도합니다.

배치 이등분

호출이 실패하고 BisectBatchOnFunctionError가 활성화되어 있으면 ReportBatchItemFailures 설정에 관계 없이 배치가 이등분됩니다.

부분적 배치 성공 응답이 수신되고 BisectBatchOnFunctionError 및 ReportBatchItemFailures가 모두 활성화되면 배치가 반환된 시퀀스 번호에서 이등분되고 Lambda는 나머지 레코드만 재시도합니다.

다음은 일괄적으로 실패한 메시지 ID 목록을 반환하는 함수 코드의 몇 가지 예입니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Javascript를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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;
}
```

TypeScript를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Kinesis 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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
}
```

Lambda에서 Kinesis 데이터 스트림 이벤트 소스에 대한 폐기된 배치 레코드 유지

Kinesis 이벤트 소스 매핑에 대한 오류 처리는 오류가 함수 간접 호출 전에 발생하는지 아니면 함수 간접 호출 중에 발생하는지 여부에 따라 달라집니다.

- 간접 호출 전: Lambda 이벤트 소스 매핑이 제한 또는 기타 문제로 인해 함수를 간접적으로 호출할 수 없는 경우 레코드가 만료되거나 이벤트 소스 매핑에 구성된 최대 기간(MaximumRecordAgeInSeconds)을 초과할 때까지 재시도합니다.
- 간접 호출 중: 함수가 간접적으로 호출되었지만 오류가 반환되는 경우 Lambda는 레코드가 만료되거나 최대 기간(MaximumRecordAgeInSeconds)을 초과하거나 구성된 재시도 할당량(MaximumRetryAttempts)에 도달할 때까지 재시도합니다. 함수 오류의 경우 실패한 배치를 두 개의 작은 배치로 분할하여 잘못된 레코드를 격리하고 시간 초과를 방지하는 BisectBatchOnFunctionError를 구성할 수도 있습니다. 배치를 분할할 때는 재시도 할당량이 소모되지 않습니다.

오류 처리에서 실패를 측정하는 경우 Lambda는 레코드를 폐기하고 스트림에서 배치 처리를 계속합니다. 기본 설정을 사용하는 경우 이는 잘못된 레코드가 영향을 받은 샤드에 대한 처리를 최대 1주 동안 차단할 수 있음을 의미합니다. 이를 방지하려면 함수의 이벤트 소스 매핑을 사용자의 사례에 적합한 최대 레코드 사용 기간 및 합당한 재시도 횟수로 구성합니다.

실패한 간접 호출에 대한 대상 구성

실패한 이벤트 소스 매핑 간접 호출 기록을 보관하려면 함수의 이벤트 소스 매핑에 대상을 추가합니다. 대상으로 전송된 각 레코드는 실패한 간접 호출에 대한 메타데이터가 포함된 JSON 문서입니다. 모든 Amazon SNS 주제 또는 Amazon SQS 대기열을 대상으로 구성할 수 있습니다. 실행 역할에 대상에 대한 권한이 있어야 합니다.

- SQS 대상의 경우: sqs:SendMessage
- SNS 대상의 경우: sns:Publish

이 콘솔을 사용하여 장애 시 대상을 구성하려면 다음 단계를 따르세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.

3. 함수 개요(Function overview)에서 대상 추가(Add destination)를 선택합니다.
4. 소스의 경우 이벤트 소스 매핑 간접 호출을 선택합니다.
5. 이벤트 소스 매핑의 경우 이 함수에 대해 구성된 이벤트 소스를 선택합니다.
6. 조건의 경우 실패 시를 선택합니다. 이벤트 소스 매핑 간접 호출의 경우 이 조건만 수락됩니다.
7. 대상 유형의 경우 Lambda가 간접 호출 레코드를 전송할 대상 유형을 선택합니다.
8. Destination(대상)에서 리소스를 선택합니다.
9. 저장을 선택합니다.

AWS Command Line Interface(AWS CLI)를 사용하여 장애 시 대상을 구성할 수도 있습니다. 예를 들어, 다음 create-event-source-mapping 명령은 SQS 장애 시 대상이 있는 이벤트 소스 매핑을 `MyFunction`에 추가합니다.

```
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"}}'
```

다음 update-event-source-mapping 명령은 두 번의 재시도 시도 후 또는 레코드가 1시간 이상 지난 경우 SNS 대상으로 실패한 간접 호출 레코드를 전송하도록 이벤트 소스 매핑을 업데이트합니다.

```
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"}}'
```

업데이트된 설정은 비동기식으로 적용되며 프로세스가 완료된 후에야 출력에 반영됩니다. get-event-source-mapping 명령을 사용하여 현재 상태를 봅니다.

대상을 제거하려면 `destination-config` 파라미터의 인수로 빈 문자열을 제공합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

다음 예는 실패한 Kinesis 이벤트 소스 간접 호출에 대해 Lambda가 SQS 대기열 또는 SNS 주제로 전송하는 내용을 보여줍니다. Lambda는 이러한 대상 유형에 대한 메타데이터만 전송하기 때문에 `streamArn`, `shardId`, `startSequenceNumber` 및 `endSequenceNumber` 필드를 사용하여 전체 원본 레코드를 확보합니다.

```
{
    "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"
    }
}
```

이 정보를 사용하여 문제 해결을 위해 스트림에서 영향을 받은 레코드를 검색할 수 있습니다. 실제 레코드는 포함되지 않으므로 이 레코드를 처리하고 실제 레코드가 만료되고 없어지기 전에 스트림에서 해당 레코드를 검색해야 합니다.

Lambda에서 상태 저장 Kinesis Data Streams 처리 구현

Lambda 함수는 연속 스트림 처리 애플리케이션을 실행할 수 있습니다. 스트림은 애플리케이션을 통해 연속적으로 흐르는 무한 데이터를 나타냅니다. 이 지속적으로 업데이트되는 입력의 정보를 분석하기 위해 시간의 항으로 정의된 윈도우를 사용하여 포함된 레코드를 바인딩할 수 있습니다.

텀블링 기간은 일정한 간격으로 시작되고 끝나는 고유한 시간대입니다. 기본적으로 Lambda 호출은 상태 비저장입니다. 즉, 외부 데이터베이스 없이는 여러 연속 호출에서 데이터를 처리하는 데 사용할 수 없습니다. 그러나 텀블링 기간을 활성화하면 호출 간에 상태를 유지할 수 있습니다. 이 상태에는 현재 윈도우에 대해 이전에 처리된 메시지의 집계 결과가 포함됩니다. 상태는 샤드당 최대 1MB가 될 수 있습니다. 이 크기를 초과하면 Lambda가 윈도우를 조기 종료합니다.

스트림의 각 레코드는 특정 윈도우에 속합니다. Lambda는 각 레코드를 한 번 이상 처리하지만 각 레코드가 한 번만 처리된다고 보장하지는 않습니다. 드물게 오류 처리와 같이 일부 레코드가 두 번 이상 처리될 수 있습니다. 레코드는 항상 처음부터 순서대로 처리됩니다. 레코드가 두 번 이상 처리되는 경우 레코드가 비순차적으로 처리될 수 있습니다.

집계 및 처리

집계 및 해당 집계의 최종 결과 처리를 위해 사용자 관리형 함수가 호출됩니다. Lambda는 윈도우 내에서 수신한 모든 레코드를 집계합니다. 이러한 레코드를 각각 별도의 호출인 여러 배치에서 받을 수 있습니다. 각 호출은 상태를 수신합니다. 따라서 텀블링 기간을 사용할 때 Lambda 함수 응답에는 `state` 속성을 포함해야 합니다. 응답에 `state` 속성이 포함되지 않은 경우 Lambda는 이 호출이 실패한 것으로 간주합니다. 이 조건을 충족하기 위해 함수는 다음 JSON 모양의 `TimeWindowEventResponse` 객체를 반환할 수 있습니다.

Example **`TimeWindowEventResponse` 값**

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> Java 함수의 경우 `Map<String, String>`을 사용하여 상태를 나타내는 것이 좋습니다.

윈도우 끝에서 `isFinalInvokeForWindow` 플래그는 이것이 최종 상태이며 처리 준비가 되었음을 나타내기 위해 `true`로 설정됩니다. 처리 후 윈도우가 완료되고 최종 호출이 완료된 다음 상태가 삭제됩니다.

윈도우 끝에서 Lambda가 집계 결과에 대한 작업을 위해 최종 처리를 사용합니다. 최종 처리는 동기식으로 호출됩니다. 성공적인 호출 후 함수가 시퀀스 번호를 검사하고 스트림 처리가 계속됩니다. 호출이 실패하면 Lambda 함수는 호출이 성공할 때까지 추가 처리를 일시 중단합니다.

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
}
```

구성

이벤트 소스 매핑을 생성하거나 업데이트할 때 텀블링 윈도우를 구성할 수 있습니다. 텀블링 윈도우를 구성하려면 윈도우를 초 단위로 지정합니다(TumblingWindowInSeconds). 다음 예제 AWS Command Line Interface(AWS CLI) 명령은 텀블링 윈도우가 120초인 스트리밍 이벤트 소스 매핑을 생성합니다. 집계 및 처리를 위해 정의된 Lambda 함수는 `tumbling-window-example-function`입니다.

```
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
```

Lambda는 레코드가 스트림에 삽입된 시간을 기준으로 텀블링 윈도우 경계를 결정합니다. 모든 레코드에는 Lambda가 경계 결정에서 사용하는 대략적인 타임스탬프가 있습니다.

텀블링 윈도우 집계는 리샤딩을 지원하지 않습니다. 샤드가 끝나면 Lambda는 현재 윈도우가 닫힌 것으로 간주하며 자식 샤드는 새 상태로 고유한 윈도우를 시작합니다. 현재 윈도우에 새 레코드가 추가되지 않으면 Lambda는 최대 2분간 기다린 후 윈도우가 끝난 것으로 간주합니다. 이렇게 하면 레코드가 간헐적으로 추가되더라도 함수가 현재 윈도우의 모든 레코드를 읽을 수 있습니다.

텀블링 윈도우는 기존 재시도 정책 `maxRetryAttempts` 및 `maxRecordAge`를 완벽하게 지원합니다.

Example Handler.py - 집계 및 처리

다음 Python 함수는 최종 상태를 집계한 다음 처리하는 방법을 보여줍니다.

```
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 이벤트 소스 매핑을 위한 Lambda 파라미터

모든 Lambda 이벤트 소스 매핑은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 일부 파라미터만 Kinesis에 적용됩니다.

Kinesis에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
BatchSize	N	100	최대값: 10,000
BisectBatchOnFunctionError	N	false	
DestinationConfig	N		폐기된 레코드의 Amazon SQS 대기열 또는 Amazon SNS 주제 대상. 자세한 내용은 실패한 간접 호출에 대한 대상 구성 단원을 참조하십시오.
활성화됨	N	true	
EventSourceArn	Y		데이터 스트림 또는 스트림 소비자의 ARN
FunctionName	Y		
FunctionResponseTypes	N		함수가 배치에서 특정 실패를 보고하도록 하려면 FunctionR

파라미터	필수	기본값	참고
			esponseTypes 에 ReportBatchItemFailures 값을 포함하세요. 자세한 내용은 Kinesis Data Streams 및 Lambda로 부분 배치 응답 구성 단원을 참조하십시오.
MaximumBatchingWindowInSeconds	N	0	
MaximumRecordAgeInSeconds	N	-1	-1은 무한을 의미: Lambda는 레코드를 버리지 않습니다 (Kinesis Data Streams 데이터 보존 설정은 계속 적용됨). 최솟값: -1 최대값: 604,800
MaximumRetryAttempts	N	-1	-1(무한): 레코드가 만료될 때까지 실패한 레코드가 다시 시도됩니다 최솟값: -1 최대값: 10,000
ParallelizationFactor	N	1	최댓값: 10

파라미터	필수	기본값	참고
StartingPosition	Y		AT_TIMESTAMP, TRIM_HORIZON, 또는 LATEST
StartingPositionTimestamp	N		StartingPosition이 AT_TIMESTAMP로 설정된 경우에만 유효합니다. 읽기를 시작하는 시간(Unix 시간 초)
TumblingWindowInSeconds	N		최솟값: 0 최댓값: 900

자습서: Kinesis Data Streams에서 Lambda 사용

이 자습서에서는 Amazon Kinesis 데이터 스트림의 이벤트를 소비하기 위한 Lambda 함수를 생성합니다.

1. 사용자 지정 앱은 스트림에 레코드를 기록합니다.
2. AWS Lambda는 스트림에서 새로운 레코드가 감지될 때마다 스트림을 폴링하고 Lambda 함수를 호출합니다.
3. AWS Lambda는 Lambda 함수를 생성할 때 지정한 실행 역할을 수임하여 Lambda 함수를 실행합니다.

필수 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 생성

함수에 AWS 리소스에 액세스할 수 있는 권한을 제공하는 실행 역할을 만듭니다.

실행 역할을 만들려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.
   - 신뢰할 수 있는 엔터티 – AWS Lambda.
   - 권한 – AWSLambdaKinesisExecutionRole.
   - 역할 이름 – **`lambda-kinesis-role`**.

AWSLambdaKinesisExecutionRole 정책은 함수가 Kinesis에서 항목을 읽고 CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다.

함수 생성

Kinesis 메시지를 처리하는 Lambda 함수를 생성합니다. 함수 코드는 Kinesis 레코드의 이벤트 ID 및 이벤트 데이터를 CloudWatch 로그에 기록합니다.

이 자습서에서는 Node.js 18.x 런타임을 사용하지만 다른 런타임 언어의 예제 코드도 제공했습니다. 다음 상자에서 탭을 선택하여 관심 있는 런타임에 대한 코드를 볼 수 있습니다. 이 단계에서 사용할 JavaScript 코드는 JavaScript 탭에 표시된 첫 번째 예제입니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda에서 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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;
}
```

TypeScript를 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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
}
```

함수를 만들려면

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. 샘플 JavaScript 코드를 새 `index.js` 파일에 복사합니다.
3. 배포 패키지를 만듭니다.

```
zip function.zip index.js
```

4. `create-function` 명령을 사용해 Lambda 함수를 만듭니다.

```
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 함수 테스트

invoke AWS Lambda CLI 명령 및 샘플 Kinesis 이벤트를 사용하여 Lambda 함수를 수동으로 호출합니다.

Lambda 함수를 테스트하려면

1. 다음 JSON을 파일에 복사하고 input.txt로 저장합니다.

```
{
    "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 명령을 사용하여 함수로 이벤트를 전송합니다.

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세

요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

응답이 `out.txt`로 저장됩니다.

Kinesis 스트림 생성

`create-stream` 명령을 사용하여 스트림을 만듭니다.

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

다음 `describe-stream` 명령을 실행하여 스트림 ARN을 가져옵니다.

```
aws kinesis describe-stream --stream-name lambda-stream
```

다음 결과가 표시됩니다:

```
{
    "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
    }
}
```

다음 단계에서 스트림 ARN을 사용하여 스트림을 Lambda 함수와 연결합니다.

AWS Lambda에서 이벤트 소스 추가

다음 AWS CLI `add-event-source` 명령을 실행합니다.

```
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
```

나중에 사용할 수 있도록 매핑 ID를 기록해 둡니다. `list-event-source-mappings` 명령을 실행하여 이벤트 소스 매핑 목록을 가져올 수 있습니다.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

응답에서 상태 값이 `enabled`인지 확인할 수 있습니다. 이벤트 소스 매핑을 비활성화하여 폴링을 일시적으로 중지시켜 레코드 손실을 방지할 수 있습니다.

설정 테스트

이벤트 소스 매핑을 테스트하려면 Kinesis 스트림에 이벤트 레코드를 추가합니다. `--data` 값은 Kinesis로 전송하기 전에 CLI가 base64로 인코딩하는 문자열입니다. 동일한 명령을 두 번 이상 실행하여 여러 레코드를 스트림에 추가할 수 있습니다.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda에서는 실행 역할을 사용하여 스트림에서 레코드를 읽습니다. 그런 다음 Lambda 함수를 호출해 레코드 배치를 전달합니다. 함수는 각 레코드에서 데이터를 디코딩하고 로깅해 CloudWatch Logs에 출력을 전송합니다. 로그는 CloudWatch 콘솔에서 확인합니다.

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**를 입력하고 Delete(삭제)를 선택합니다.

Kinesis 스트림을 삭제하려면

1. AWS Management Console에 로그인하여 https://console.aws.amazon.com/kinesis에서 Kinesis 콘솔을 엽니다.
2. 생성한 스트림을 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**을 입력합니다.
5. Delete(삭제)를 선택합니다.

Amazon MQ에서 Lambda 사용

> **Note**
>
> Lambda 함수 이외의 대상으로 데이터를 전송하거나 데이터를 전송하기 전에 데이터를 보강하려는 경우 Amazon EventBridge 파이프를 참조하세요.

Amazon MQ는 Apache ActiveMQ 및 RabbitMQ를 위한 관리형 메시지 브로커 서비스입니다. 메시지 브로커를 사용하면 소프트웨어 애플리케이션 및 구성 요소가 토픽 또는 대기열 이벤트 대상을 통해 다양한 프로그래밍 언어, 운영 체제 및 공식 메시징 프로토콜을 사용하여 통신할 수 있습니다.

Amazon MQ는 ActiveMQ 또는 RabbitMQ 브로커를 설치하고 다른 네트워크 토폴로지 및 기타 인프라 요구 사항을 제공하여 사용자 대신 Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스를 관리할 수도 있습니다.

Lambda 함수를 사용하여 Amazon MQ 메시지 브로커의 레코드를 처리할 수 있습니다. Lambda는 이벤트 소스 매핑을 통해 함수를 호출합니다. 이 매핑은 브로커로부터 메시지를 읽고 함수를 동기식으로 호출하는 Lambda 리소스입니다.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

Amazon MQ 이벤트 소스 매핑에는 다음과 같은 구성 제한 사항이 있습니다.

- 동시성 — Amazon MQ 이벤트 소스 매핑을 사용하는 Lambda 함수에는 기본 최대 동시성 설정이 있습니다. ActiveMQ의 경우 Lambda 서비스는 동시 실행 환경 수를 다섯으로 제한합니다. RabbitMQ의 경우 동시 실행 환경 수는 1개로 제한됩니다. 함수의 예약되거나 프로비저닝된 동시성 설정을 변경하더라도 Lambda 서비스가 더 많은 실행 환경을 사용할 수 있게 해 주지는 않습니다. 기본 최대 동시성 증가를 요청하려면 AWS Support에 문의하세요.
- 교차 계정 - Lambda는 교차 계정 처리를 지원하지 않습니다. Lambda를 사용하여 다른 AWS 계정에 있는 Amazon MQ 메시지 브로커의 레코드를 처리할 수 없습니다.
- 인증 - ActiveMQ의 경우 ActiveMQ SimpleAuthenticationPlugin만 지원됩니다. RabbitMQ의 경우 PLAIN 인증 메커니즘만 지원됩니다. 사용자는 AWS Secrets Manager를 사용해 자신의 자격 증명을 관리해야 합니다. ActiveMQ 인증에 대한 자세한 내용은 Amazon MQ 개발자 안내서의 ActiveMQ 브로커와 LDAP 통합을 참조하세요.
- 연결 할당량 - 브로커에는 와이어 레벨 프로토콜당 허용되는 최대 연결 수가 있습니다. 이 할당량은 브로커 인스턴스 유형에 따라 결정됩니다. 자세한 내용은 Amazon MQ 개발자 안내서에서 Amazon MQ의 할당량의 브로커 단원을 참조하세요.
- 연결성 - 퍼블릭 또는 프라이빗 Virtual Private Cloud(VPC)에서 브로커를 생성할 수 있습니다. 프라이빗 VPC의 경우 레코드와 상호 작용하려면 Lambda 함수에게 메시지를 수신하는 VPC에 대한 액세스 권한이 있어야 합니다. 자세한 내용은 이 주제의 후반부에서 the section called “네트워크 구성” 단원을 참조하세요.

- 이벤트 대상 - 대기열 대상만 지원됩니다. 그러나 Lambda와 상호 작용하는 동안 내부적으로 토픽으로 동작하는 가상 토픽을 대기열로 사용할 수 있습니다. 자세한 내용은 Apache ActiveMQ 웹사이트의 가상 대상과 RabbitMQ의 가상 호스트를 참조하세요.
- 네트워크 토폴로지 - ActiveMQ의 경우 이벤트 소스 매핑당 하나의 단일 인스턴스 또는 대기 브로커만 지원됩니다. RabbitMQ의 경우 이벤트 소스 매핑당 하나의 단일 인스턴스 브로커 또는 클러스터 배포만 지원됩니다. 단일 인스턴스 브로커에는 장애 조치 엔드포인트가 필요합니다. 이러한 브로커 배포 모드에 대한 자세한 내용은 Amazon MQ 개발자 안내서에서 Active MQ 브로커 아키텍쳐 및 Rabbit MQ 브로커 아키텍쳐를 참조하세요.
- 프로토콜 - 지원되는 프로토콜은 Amazon MQ 통합 유형에 따라 다릅니다.
  - ActiveMQ 통합의 경우 Lambda는 OpenWire/Java Message Service(JMS) 프로토콜을 사용하여 메시지를 소비합니다. 메시지 소비를 위한 다른 프로토콜은 지원되지 않습니다. JMS 프로토콜 내에서 `TextMessage` 및 `BytesMessage`만 지원됩니다. Lambda는 JMS 사용자 지정 속성도 지원합니다. OpenWire 프로토콜에 대한 자세한 내용은 Apache ActiveMQ 웹 사이트에서 OpenWire를 참조하세요.
  - RabbitMQ 통합의 경우 Lambda는 AMQP 0-9-1 프로토콜을 사용하여 메시지를 소비합니다. 메시지 소비를 위한 다른 프로토콜은 지원되지 않습니다. RabbitMQ의 AMQP 0-9-1 프로토콜 구현에 대한 자세한 내용은 RabbitMQ 웹사이트의 AMQP 0-9-1 전체 참조 가이드를 참조하세요.

Lambda는 Amazon MQ가 지원하는 최신 버전의 ActiveMQ 및 RabbitMQ를 자동으로 지원합니다. 지원되는 최신 버전은 Amazon MQ 개발자 가이드의 Amazon MQ 릴리스 노트를 참조하세요

> **Note**
>
> 기본적으로 Amazon MQ에는 브로커에 대한 주별 유지 관리 기간이 있습니다. 해당 기간 동안에는 브로커를 사용할 수 없습니다. 예비 인스턴스가 없는 브로커의 경우 Lambda가 해당 기간 동안 메시지를 처리할 수 없습니다.

단원

- Lambda 소비자 그룹
- 실행 역할 권한
- 네트워크 구성
- 권한 추가 및 이벤트 소스 매핑 생성
- 이벤트 소스 매핑 업데이트
- 이벤트 소스 매핑 오류

- Amazon MQ 및 RabbitMQ 구성 파라미터

Lambda 소비자 그룹

Amazon MQ와 상호 작용하기 위해 Lambda는 Amazon MQ 브로커에서 읽을 수 있는 소비자 그룹을 생성합니다. 소비자 그룹은 이벤트 소스 매핑 UUID와 동일한 ID를 사용하여 생성됩니다.

Amazon MQ 이벤트 소스의 경우 Lambda에서 레코드를 일괄 처리하여 단일 페이로드로 함수에 전송합니다. 동작을 제어하려면 일괄 처리 기간 및 배치 크기를 구성할 수 있습니다. Lambda는 페이로드 크기 최댓값인 6MB이 처리되거나 일괄 처리 기간이 만료되거나 레코드 수가 전체 배치 크기에 도달할 때까지 메시지를 가져옵니다. 자세한 내용은 일괄 처리 동작 단원을 참조하십시오.

소비자 그룹은 메시지를 바이트의 BLOB로 검색하고 단일 JSON 페이로드에 base64로 인코딩한 다음, 함수를 호출합니다. 함수가 배치의 어떤 메시지에 대해 오류를 반환하면 Lambda는 처리가 성공하거나 메시지가 만료될 때까지 전체 메시지 배치를 다시 시도합니다.

> **Note**
>
> Lambda 함수의 최대 제한 시간은 일반적으로 15분이지만 Amazon MSK, 자체 관리형 Apache Kafka, Amazon DocumentDB, ActiveMQ 및 RabbitMQ용 Amazon MQ에 대한 이벤트 소스 매핑은 최대 제한 시간이 14분인 함수만 지원합니다. 이 제약 조건에 따라 이벤트 소스 매핑에서 함수 오류 및 재시도를 적절히 처리할 수 있습니다.

Amazon CloudWatch의 `ConcurrentExecutions` 지표를 사용하여 지정된 함수의 동시성 사용량을 모니터링할 수 있습니다. 동시성에 대한 자세한 내용은 the section called “예약된 동시성 구성” 단원을 참조하세요.

Example Amazon MQ 레코드 이벤트

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 예제에서 `pizzaQueue`는 RabbitMQ 대기열의 이름이고 `/`는 가상 호스트의 이름입니다. 메시지를 받을 때 이벤트 소스는 `pizzaQueue::/`에 메시지를 나열합니다.

실행 역할 권한

Amazon MQ 브로커에서 레코드를 읽으려면 Lambda 함수의 실행 역할에 다음 권한이 추가되어야 합니다.

- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup

- logs:CreateLogStream
- logs:PutLogEvents

> **Note**
>
> 암호화된 고객 관리형 키를 사용하는 경우 `kms:Decrypt` 권한도 추가합니다.

네트워크 구성

이벤트 소스 매핑을 통해 Lambda에 브로커에 대한 모든 액세스 권한을 부여하려면 브로커가 퍼블릭 엔드포인트(퍼블릭 IP 주소)를 사용하거나 브로커를 생성한 Amazon VPC에 대한 액세스 권한을 제공해야 합니다.

기본적으로 Amazon MQ 브로커를 생성하면 `PubliclyAccessible` 플래그가 false로 설정됩니다. 브로커가 퍼블릭 IP 주소를 수신하려면 `PubliclyAccessible` 플래그를 true로 설정해야 합니다.

Lambda와 함께 Amazon MQ를 사용하는 모범 사례는 AWS PrivateLink VPC 엔드포인트를 사용하고 Lambda 함수에 브로커의 VPC에 대한 액세스 권한을 부여하는 것입니다. Lambda용 엔드포인트를 배포하고 ActiveMQ의 경우에만 AWS Security Token Service(AWS STS)용 엔드포인트를 배포합니다. 브로커에서 인증을 사용하는 경우 AWS Secrets Manager용 엔드포인트도 배포합니다. 자세한 내용은 the section called “VPC 엔드포인트 작업”을 참조하십시오.

또는 Amazon MQ 브로커가 포함된 VPC의 각 퍼블릭 서브넷에 NAT 게이트웨이를 구성합니다. 자세한 내용은 the section called “VPC 함수에 대한 인터넷 액세스” 단원을 참조하십시오.

Amazon MQ 브로커에 대한 이벤트 소스 매핑을 생성하는 경우 Lambda는 브로커 VPC의 서브넷과 보안 그룹에 대해 탄력적 네트워크 인터페이스(ENI)가 이미 존재하는지 확인합니다. Lambda가 기존 ENI를 찾으면 이를 재사용하려고 시도합니다. 그렇지 않으면 Lambda가 이벤트 소스에 연결하고 함수를 호출하기 위해 새 ENI를 생성합니다.

> **Note**
>
> Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. 이러한 VPC는 서비스에 의해 자동으로 유지 관리되며 고객에게는 표시되지 않습니다. 또한 함수를 Amazon VPC에 연결할 수도 있습니다. 어느 경우든 함수의 VPC 구성은 이벤트 소스 매핑에 영향을 미치지 않습니다. 이벤트 소스의 VPC 구성에 따라 Lambda가 이벤트 소스에 연결되는 방식이 결정됩니다.

VPC 보안 그룹 규칙

최소한 다음 규칙을 사용하여 클러스터가 포함된 Amazon VPC의 보안 그룹을 구성합니다.

- 인바운드 규칙 – 자체 보안 그룹 내에서 이벤트 소스에 대해 지정된 보안 그룹에 대한 브로커 포트의 모든 트래픽을 허용합니다. ActiveMQ는 기본적으로 포트 61617을 사용하고 RabbitMQ는 기본적으로 포트 5671을 사용합니다.
- 아웃바운드 규칙 - 모든 대상에 대해 포트 443의 모든 트래픽을 허용합니다. 자체 보안 그룹 내에서 브로커 포트의 모든 트래픽을 허용합니다. ActiveMQ는 기본적으로 포트 61617을 사용하고 RabbitMQ는 기본적으로 포트 5671을 사용합니다.
- NAT 게이트웨이 대신 VPC 엔드포인트를 사용하는 경우 VPC 엔드포인트와 연결된 보안 그룹은 이벤트 소스의 보안 그룹에서 포트 443의 모든 인바운드 트래픽을 허용해야 합니다.

VPC 엔드포인트 작업

VPC 엔드포인트를 사용하는 경우 함수를 호출하는 API 직접 호출은 ENI를 사용하여 이러한 엔드포인트를 통해 라우팅됩니다. Lambda 서비스 보안 주체는 해당 ENI를 사용하는 모든 함수에서 `lambda:InvokeFunction`을 호출해야 합니다. 또한 ActiveMQ의 경우 Lambda 서비스 보안 주체는 ENI를 사용하는 역할에서 `sts:AssumeRole`을 직접적으로 호출해야 합니다.

기본적으로 VPC 엔드포인트에는 개방적인 IAM 정책이 있습니다. 모범 사례는 특정 보안 주체만 해당 엔드포인트를 사용하여 필요한 작업을 수행할 수 있도록 이러한 정책을 제한하는 것입니다. 이벤트 소스 매핑이 Lambda 함수를 호출할 수 있도록 하려면 VPC 엔드포인트 정책에서 Lambda 서비스 원칙이 `lambda:InvokeFunction`을, ActiveMQ의 경우 `sts:AssumeRole`을 호출하도록 허용해야 합니다. 조직 내에서 발생하는 API 직접 호출만 허용하도록 VPC 엔드포인트 정책을 제한하면 이벤트 소스 매핑이 제대로 작동하지 않습니다.

다음 예제 VPC 엔드포인트 정책은 AWS STS 및 Lambda 엔드포인트에 필요한 액세스 권한을 부여하는 방법을 설명합니다.

Example VPC 엔드포인트 정책 - AWS STS 엔드포인트(ActiveMQ만 해당)

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
```

```
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 엔드포인트 정책 - Lambda 엔드포인트

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Amazon MQ 브로커 클러스터가 인증을 사용하는 경우 Secrets Manager 엔드포인트에 대한 VPC 엔드포인트 정책을 제한할 수도 있습니다. Secrets Manager API를 호출하기 위해 Lambda는 Lambda 서비스 보안 주체가 아닌 함수 역할을 사용합니다. 다음 예제는 Secrets Manager 엔드포인트 정책을 보여줍니다.

Example VPC 엔드포인트 정책 – Secrets Manager 엔드포인트

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
```

```
        }
    ]
}
```

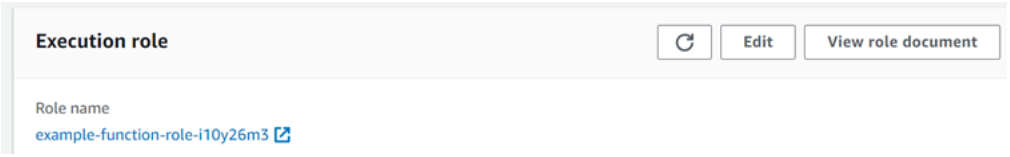
권한 추가 및 이벤트 소스 매핑 생성

이벤트 소스 매핑을 생성하여 Lambda가 Amazon MQ 브로커의 레코드를 Lambda 함수로 전송하도록 지시합니다. 여러 이벤트 소스 매핑을 생성하여 여러 함수로 동일한 데이터를 처리하거나, 단일 함수로 여러 소스의 항목을 처리할 수 있습니다.

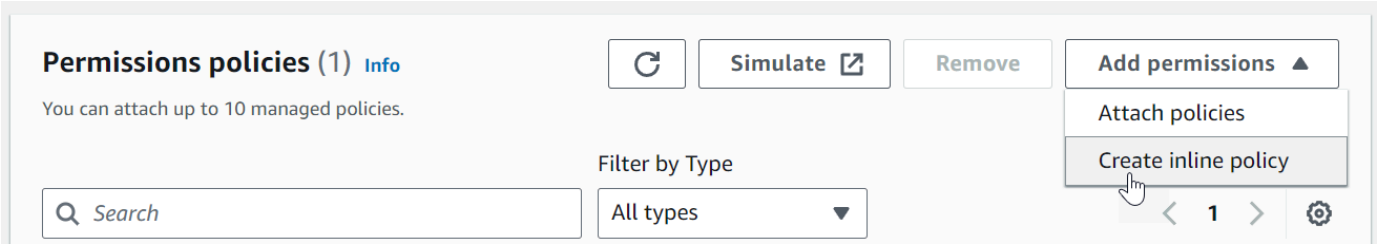
Amazon MQ에서 읽도록 함수를 구성하려면 필요한 권한을 추가하고 Lambda 콘솔에서 MQ 트리거를 생성합니다.

권한 추가 및 트리거 생성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 권한(Permissions)을 선택합니다.
4. 역할 이름에서 실행 역할에 대한 링크를 선택합니다. 이 링크를 클릭하면 IAM 콘솔에서 역할이 열립니다.



5. 권한 추가를 선택하고 인라인 정책 생성을 선택합니다.



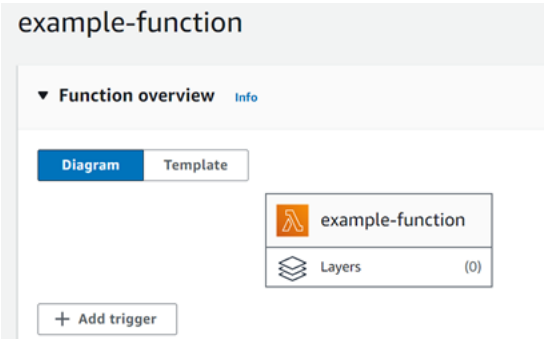
6. 정책 편집기 섹션에서 JSON을 선택합니다. 다음 정책을 입력합니다. 함수가 Amazon MQ 브로커에서 읽으려면 이러한 권한이 필요합니다.

```
{
  "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**
>
> 암호화된 고객 관리형 키를 사용하는 경우 `kms:Decrypt` 권한도 추가해야 합니다.

7. Next(다음)를 선택합니다. 정책 이름을 입력한 후 정책 생성을 선택합니다.
8. Lambda 콘솔에서 함수로 돌아갑니다. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.



9. MQ 트리거 유형을 선택합니다.
10. 필요한 옵션을 구성한 다음 추가를 선택합니다.

Lambda는 Amazon MQ 이벤트 소스에 대해 다음과 같은 옵션을 지원합니다.

- 브로커 - Amazon MQ 브로커를 선택합니다.

- 배치 크기(Batch size) - 단일 배치에서 검색할 최대 메시지 수를 설정합니다.
- 대기열 이름(Queue name) - 사용할 Amazon MQ 대기열을 입력합니다.
- 소스 액세스 구성 - 가상 호스트 정보 및 브로커 자격 증명을 저장하는 Secrets Manager 암호를 입력합니다.
- 트리거 활성화 - 레코드 처리를 중지하려면 트리거를 비활성화합니다.

트리거를 활성화하거나 비활성화(또는 삭제)하려면 디자이너에서 MQ 트리거를 선택합니다. 트리거를 재구성하려면 이벤트 소스 매핑 API 작업을 사용합니다.

이벤트 소스 매핑 업데이트

이벤트 소스 매핑을 업데이트하려면 update-event-source-mapping 명령을 사용하세요. 다음 예제 명령은 배치 크기가 2인 이벤트 소스 매핑을 업데이트합니다.

```
aws lambda update-event-source-mapping \
--uuid 91eaeb7e-c976-1234-9451-8709db01f137 \
--batch-size 2
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```

Lambda는 이러한 설정을 비동기적으로 업데이트합니다. 이 프로세스가 완료될 때까지 출력에 변경 사항이 반영되지 않습니다. 리소스의 현재 상태를 보려면 get-event-source-mapping 명령을 사용합니다.

```
aws lambda get-event-source-mapping \
```

```
--uuid 91eaeb7e-c976-4939-9451-8709db01f137
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```

이벤트 소스 매핑 오류

Lambda 함수에 복구할 수 없는 오류가 발생하면 Amazon MQ 소비자가 레코드 처리를 중지합니다. 다른 소비자는 동일한 오류가 발생하지 않는 한 처리를 계속할 수 있습니다. 중지된 소비자의 잠재적 원인을 확인하려면 `StateTransitionReason`의 반환 세부 정보에서 다음 코드 중 하나에 대한 `EventSourceMapping` 필드를 확인하세요.

`ESM_CONFIG_NOT_VALID`

이벤트 소스 매핑 구성이 잘못되었습니다.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda가 이벤트 소스를 인증하지 못했습니다.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda에게 이벤트 소스에 액세스하는 데 필요한 권한이 없습니다.

`FUNCTION_CONFIG_NOT_VALID`

함수의 구성이 유효하지 않습니다.

Lambda가 크기 때문에 레코드를 버리는 경우에도 레코드가 처리되지 않습니다. Lambda 레코드의 크기 제한은 6MB입니다. 함수 오류 시 메시지를 다시 전달하려면 배달 못한 편지 대기열(DLQ)을 사

용할 수 있습니다. 자세한 내용은 Apache ActiveMQ 웹 사이트에서 Message Redelivery and DLQ Handling을, RabbitMQ에서 Reliability Guide를 참조하세요.

> **Note**
>
> Lambda는 사용자 정의 재전달 정책을 지원하지 않습니다. 그 대신 Lambda는 Apache ActiveMQ 웹사이트의 정책 재전달 페이지에 있는 기본값(`maximumRedeliveries`를 6으로 설정)을 사용한 정책을 사용합니다.

Amazon MQ 및 RabbitMQ 구성 파라미터

모든 Lambda 이벤트 소스 유형은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 일부 파라미터만 Amazon MQ 및 RabbitMQ에 적용됩니다.

Amazon MQ 및 RabbitMQ에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
BatchSize	N	100	최대값: 10,000
활성	N	true	
FunctionName	Y		
FilterCriteria	N		Lambda 이벤트 필터링
MaximumBatchingWindowInSeconds	N	500ms	일괄 처리 동작
대기열	N		소비할 Amazon MQ 브로커 대상 대기열의 이름입니다.
SourceAccessConfigurations	N		ActiveMQ의 BASIC_AUTH 자격 증명입니다. RabbitMQ는 BASIC_AUTH 자격

파라미터	필수	기본값	참고
			증명과 VIRTUAL_HOST 정보를 모두 포함할 수 있습니다.

Amazon MSK에서 Lambda 사용

> **Note**
>
> Lambda 함수 이외의 대상으로 데이터를 전송하거나 데이터를 전송하기 전에 데이터를 보강하려는 경우 Amazon EventBridge 파이프를 참조하세요.

Amazon Managed Streaming for Apache Kafka(Amazon MSK)는 Apache Kafka를 사용하여 스트리밍 데이터를 처리하는 애플리케이션의 구축 및 실행을 위해 사용할 수 있는 완전관리형 서비스입니다. Amazon MSK는 Kafka를 실행하는 클러스터의 설정, 크기 조정, 관리를 간소화합니다. 또한 Amazon MSK는 여러 Availability Zones에 맞게, 그리고 AWS Identity and Access Management(IAM)를 통한 보안을 위해 애플리케이션을 쉽게 구성할 수 있도록 도와줍니다. Amazon MSK는 Kafka의 여러 오픈 소스 버전을 지원합니다.

이벤트 소스로서 Amazon MSK는 Amazon Simple Queue Service(Amazon SQS) 또는 Amazon Kinesis를 사용하는 것과 유사하게 작동합니다. Lambda는 이벤트 소스의 새 메시지를 내부적으로 폴링한 다음 대상 Lambda 함수를 동기적으로 호출합니다. Lambda는 메시지를 배치 단위로 읽고 이를 함수에 이벤트 페이로드로 제공합니다. 최대 배치 크기를 구성할 수 있습니다. 기본값은 메시지 100건입니다. 자세한 내용은 일괄 처리 동작 단원을 참조하십시오.

> **Note**
>
> Lambda 함수의 최대 제한 시간은 일반적으로 15분이지만 Amazon MSK, 자체 관리형 Apache Kafka, Amazon DocumentDB, ActiveMQ 및 RabbitMQ용 Amazon MQ에 대한 이벤트 소스 매핑은 최대 제한 시간이 14분인 함수만 지원합니다. 이 제약 조건에 따라 이벤트 소스 매핑에서 함수 오류 및 재시도를 적절히 처리할 수 있습니다.

Lambda는 각 파티션에 대해 순차적으로 메시지를 읽습니다. 단일 Lambda 페이로드에는 여러 파티션의 메시지가 포함될 수 있습니다. Lambda는 각 배치를 처리한 후 해당 배치에 있는 메시지의 오프셋을

커밋합니다. 함수가 배치의 어떤 메시지에 대해 오류를 반환하면 Lambda는 처리가 성공하거나 메시지가 만료될 때까지 전체 메시지 배치를 다시 시도합니다.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

Amazon MSK를 이벤트 소스로 구성하는 방법의 예는 AWS 컴퓨팅 블로그에서 Using Amazon MSK as an event source for AWS Lambda를 참조하세요. 전체 자습서는 Amazon MSK Labs에서 Amazon MSK Lambda 통합을 참조하세요.

주제

자습서: Amazon MSK 이벤트 소스 매핑을 사용하여 간접적으로 Lambda 함수 호출

이 자습서에서는 다음을 수행합니다.

- 기존 Amazon MSK 클러스터와 동일한 AWS 계정에서 Lambda 함수를 생성합니다.

- Lambda가 Amazon MSK와 통신할 수 있도록 네트워킹과 인증을 구성합니다.
- 주제에 이벤트가 표시될 때 Lambda 함수를 실행하는 Lambda Amazon MSK 이벤트 소스 매핑을 설정합니다.

이 단계를 완료한 후 Amazon MSK로 이벤트가 전송되면 사용자 지정 Lambda 코드를 사용하여 해당 이벤트를 자동으로 처리하도록 Lambda 함수를 설정할 수 있습니다.

이 기능으로 무엇을 할 수 있나요?

예제 솔루션: MSK 이벤트 소스 매핑을 사용하여 고객에게 실시간 점수를 제공하세요.

다음 시나리오를 생각해 보세요. 회사에서 고객이 스포츠 경기와 같은 실시간 이벤트에 대한 정보를 볼 수 있는 웹 애플리케이션을 호스팅하고 있습니다. 게임의 정보 업데이트는 Amazon MSK의 Kafka 주제를 통해 팀에 제공됩니다. 개발하는 애플리케이션 내에서 고객에게 라이브 이벤트의 업데이트된 보기를 제공하기 위해 MSK 주제의 업데이트를 사용하는 솔루션을 설계하려고 합니다. 다음과 같은 설계 접근 방식을 결정했습니다. 클라이언트 애플리케이션은 AWS에서 호스팅되는 서버리스 백엔드와 통신합니다. 클라이언트는 Amazon API Gateway WebSocket API를 사용하여 WebSocket 세션을 통해 연결됩니다.

이 솔루션에서는 MSK 이벤트를 읽고 일부 사용자 지정 로직을 수행하여 애플리케이션 계층에 맞게 해당 이벤트를 준비한 다음 해당 정보를 API Gateway API로 전달하는 구성 요소가 필요합니다. Lambda 함수에 사용자 지정 로직을 제공한 다음 AWS Lambda Amazon MSK 이벤트 소스 매핑을 통해 직접적으로 호출하여 AWS Lambda로 이 구성 요소를 구현할 수 있습니다.

Amazon API Gateway WebSocket API를 사용하여 솔루션을 구현하는 방법에 대한 자세한 내용은 API Gateway 설명서의 WebSocket API 자습서를 참조하세요.

필수 조건

다음과 같은 사전 구성된 리소스가 있는 AWS 계정:

이러한 사전 요구 사항을 충족하려면 Amazon MSK 설명서의 Getting started using Amazon MSK를 따르는 것이 좋습니다.

- Amazon MSK 클러스터. Getting started using Amazon MSK의 Create an Amazon MSK cluster를 참조하세요.
- 다음 구성:
  - 클러스터 보안 설정에서 IAM 역할 기반 인증이 활성화됨인지 확인합니다. 이렇게 하면 필요한 Amazon MSK 리소스에만 액세스하도록 Lambda 함수를 제한하여 보안이 강화됩니다. 이는 새 Amazon MSK 클러스터에서 기본적으로 활성화됩니다.

  - 클러스터 네트워킹 설정에서 퍼블릭 액세스가 꺼져 있는지 확인합니다. Amazon MSK 클러스터의 인터넷 액세스를 제한하면 데이터를 처리하는 중개자 수가 제한되어 보안이 강화됩니다. 이는 새 Amazon MSK 클러스터에서 기본적으로 활성화됩니다.
- 이 솔루션에 사용할 Amazon MSK 클러스터의 Kafka 주제. Getting started using Amazon MSK의 Create a topic을 참조하세요.
- Kafka 클러스터에서 정보를 검색하고 테스트를 위해 Kafka 이벤트를 주제에 전송하도록 설정된 Kafka 관리자 호스트(예: Kafka 관리 CLI와 Amazon MSK IAM 라이브러리가 설치된 Amazon EC2 인스턴스). Getting started using Amazon MSK의 Create a client machine을 참조하세요.

이러한 리소스를 설정한 후에는 AWS 계정에서 다음 정보를 수집하여 계속할 준비가 되었는지 확인하세요.

- Amazon MSK 클러스터의 이름. 이 정보는 Amazon MSK 콘솔에서 확인할 수 있습니다.
- Amazon MSK 클러스터용 ARN의 일부인 클러스터 UUID(Amazon MSK 콘솔에서 찾을 수 있음). 이 정보를 찾으려면 Amazon MSK 설명서의 Listing clusters에 나와 있는 절차를 따르세요.
- Amazon MSK 클러스터와 연결된 보안 그룹. 이 정보는 Amazon MSK 콘솔에서 확인할 수 있습니다. 다음 단계에서는 이를 *clusterSecurityGroups*라고 합니다.
- Amazon MSK 클러스터를 포함하는 Amazon VPC의 ID. Amazon MSK 콘솔에서 Amazon MSK 클러스터와 연결된 서브넷을 식별한 다음 Amazon VPC 콘솔에서 해당 서브넷과 연결된 Amazon VPC를 식별하여 이 정보를 찾을 수 있습니다.
- 솔루션에 사용되는 Kafka 주제의 이름. Kafka 관리 호스트에서 Kafka `topics` CLI로 Amazon MSK 클러스터를 직접적으로 호출하여 이 정보를 찾을 수 있습니다. 주제 CLI에 대한 자세한 내용은 Kafka 설명서의 Adding and removing topics를 참조하세요.
- Lambda 함수에서 사용하기에 적합한 Kafka 주제에 대한 소비자 그룹의 이름. 이 그룹은 Lambda에서 자동으로 생성할 수 있으므로 Kafka CLI를 사용하여 생성할 필요가 없습니다. 소비자 그룹을 관리해야 하는 경우 소비자 그룹 CLI에 대한 자세한 내용은 Kafka 설명서의 Managing Consumer Groups를 참조하세요.

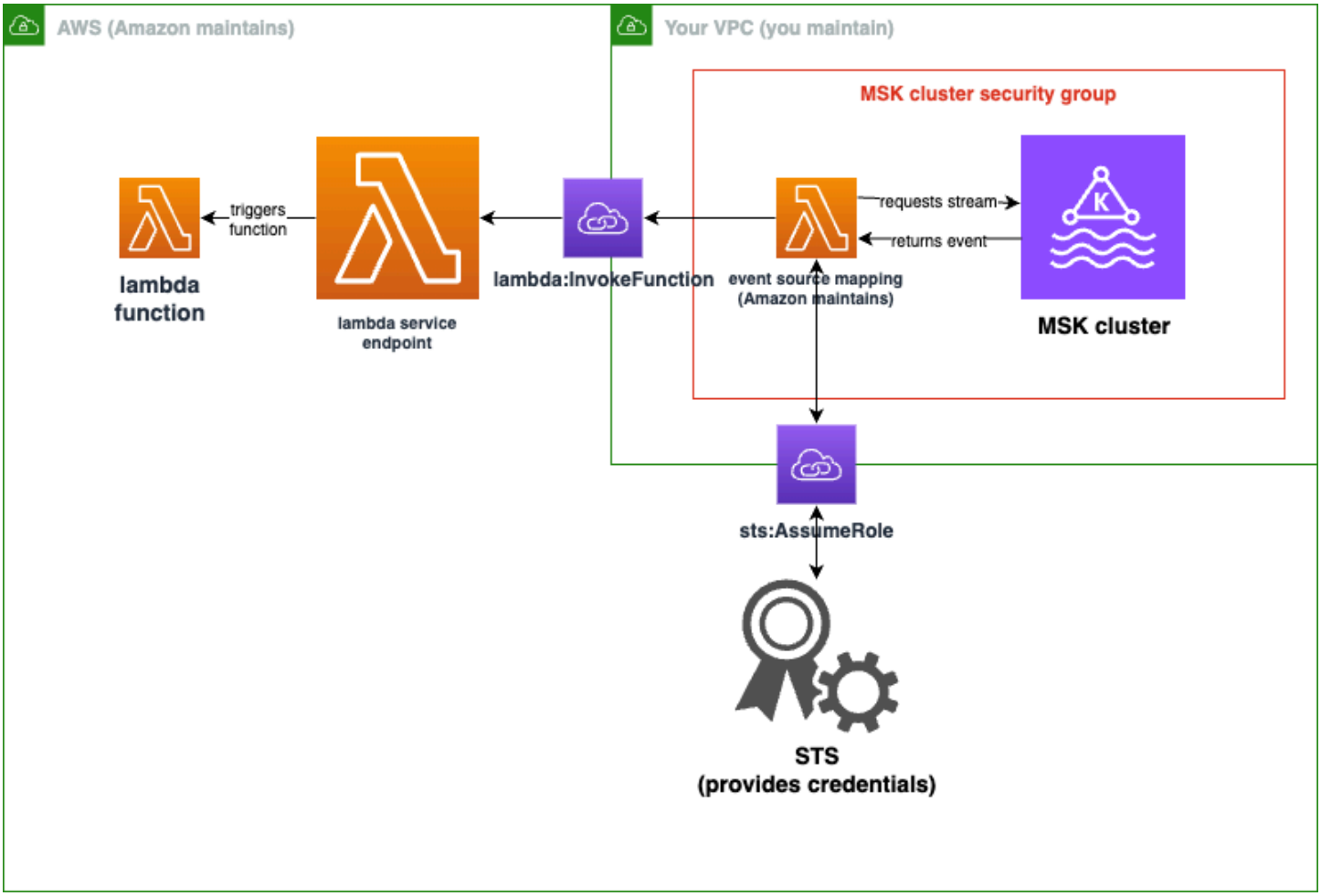
AWS 계정의 다음 권한:

- Lambda 함수를 생성하고 관리할 수 있는 권한
- IAM 정책을 생성하고 Lambda 함수와 연결할 수 있는 권한
- Amazon MSK 클러스터를 호스팅하는 Amazon VPC에서 Amazon VPC 엔드포인트를 생성하고 네트워킹 구성을 변경할 수 있는 권한

Lambda가 Amazon MSK와 통신할 수 있도록 네트워크 연결 구성

AWS PrivateLink를 사용하여 Lambda와 Amazon MSK를 연결합니다. 이를 위해 Amazon VPC 콘솔에서 인터페이스 Amazon VPC 엔드포인트를 생성할 수 있습니다. 네트워크 구성에 대한 자세한 내용은 the section called “네트워크 구성” 섹션을 참조하세요.

Amazon MSK 이벤트 소스 매핑이 Lambda 함수를 대신하여 실행되는 경우 Lambda 함수의 실행 역할을 수임합니다. 이 IAM 역할은 Amazon MSK 클러스터와 같이 IAM으로 보호되는 리소스에 액세스할 수 있도록 매핑 권한을 부여합니다. 구성 요소는 실행 역할을 공유하지만 다음 다이어그램과 같이 Amazon MSK 매핑과 Lambda 함수에는 해당 작업에 대한 별도의 연결 요구 사항이 있습니다.



이벤트 소스 매핑은 Amazon MSK 클러스터 보안 그룹에 속합니다. 이 네트워킹 단계에서는 Amazon MSK 클러스터 VPC에서 Amazon VPC 엔드포인트를 생성하여 Lambda 및 STS 서비스에 이벤트 소스 매핑을 연결합니다. 이러한 엔드포인트를 보호하여 Amazon MSK 클러스터 보안 그룹의 트래픽을 허용하세요. 그런 다음 Amazon MSK 클러스터 보안 그룹을 조정하여 이벤트 소스 매핑이 Amazon MSK 클러스터와 통신할 수 있도록 합니다.

AWS Management Console을 사용하여 다음 단계를 구성할 수 있습니다.

Lambda와 Amazon MSK를 연결하도록 인터페이스 Amazon VPC 엔드포인트를 구성하려면 다음을 수행하세요.

1. 443에서 *clusterSecurityGroups*의 인바운드 TCP 트래픽을 허용하는 인터페이스 Amazon VPC 엔드포인트에 대한 보안 그룹인 *endpointSecurityGroup*을 생성합니다. Amazon EC2 설명서의 보안 그룹 생성에 나와 있는 절차에 따라 보안 그룹을 생성합니다. 그런 다음 Amazon EC2 설명서의 보안 그룹에 규칙 추가 절차에 따라 적절한 규칙을 추가합니다.

   다음 정보로 보안 그룹 생성:

   인바운드 규칙을 추가할 때 *clusterSecurityGroups*에서 각 보안 그룹에 대한 규칙을 생성합니다. 각 규칙에 대해 다음을 수행합니다.

   - 타입에 대해 HTTPS를 선택합니다.
   - 소스로 *clusterSecurityGroups* 중 하나를 선택합니다.

2. Amazon MSK 클러스터가 포함된 Amazon VPC에 Lambda 서비스를 연결하는 엔드포인트를 생성합니다. 인터페이스 엔드포인트 생성 절차를 따릅니다.

   다음 정보로 인터페이스 엔드포인트 생성:

   - 서비스 이름으로 `com.amazonaws.`*regionName*`.lambda`를 선택합니다. 여기서 *regionName*은 Lambda 함수를 호스팅합니다.
   - VPC로 Amazon MSK 클러스터가 포함된 Amazon VPC를 선택합니다.
   - 보안 그룹으로 앞에서 생성한 *endpointSecurityGroup*을 선택합니다.
   - 서브넷으로 Amazon MSK 클러스터를 호스팅하는 서브넷을 선택합니다.
   - 정책으로 Lambda 서비스 주체가 `lambda:InvokeFunction` 작업에 사용할 엔드포인트를 보호하는 다음 정책 문서를 제공합니다.

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
```

```
        }
    ]
}
```

- DNS 이름 활성화가 설정된 상태로 유지되는지 확인합니다.

3. Amazon MSK 클러스터가 포함된 Amazon VPC에 AWS STS 서비스를 연결하는 엔드포인트를 생성합니다. 인터페이스 엔드포인트 생성 절차를 따릅니다.

   다음 정보로 인터페이스 엔드포인트 생성:

   - 서비스 이름에서 AWS STS를 선택합니다.
   - VPC로 Amazon MSK 클러스터가 포함된 Amazon VPC를 선택합니다.
   - 보안 그룹으로 *endpointSecurityGroup*을 선택합니다.
   - 서브넷으로 Amazon MSK 클러스터를 호스팅하는 서브넷을 선택합니다.
   - 정책으로 Lambda 서비스 주체가 `sts:AssumeRole` 작업에 사용할 엔드포인트를 보호하는 다음 정책 문서를 제공합니다.

   ```
   {
       "Statement": [
           {
               "Action": "sts:AssumeRole",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - DNS 이름 활성화가 설정된 상태로 유지되는지 확인합니다.

4. Amazon MSK 클러스터와 연결된 각 보안 그룹, 즉 *clusterSecurityGroups*에 대해 다음을 허용합니다.

   - 9098의 모든 인바운드 및 아웃바운드 TCP 트래픽을 자체 내부를 포함하여 모든 *clusterSecurityGroups*에 허용합니다.
   - 443에서 모든 아웃바운드 TCP 트래픽을 허용합니다.

이 트래픽 중 일부는 기본 보안 그룹 규칙에 의해 허용되므로 클러스터가 단일 보안 그룹에 연결되어 있고 해당 그룹에 기본 규칙이 있는 경우 추가 규칙이 필요하지 않습니다. 보안 그룹 규칙을 조정하려면 Amazon EC2 설명서의 보안 그룹에 규칙 추가 절차를 따릅니다.

다음 정보로 보안 그룹에 규칙 추가

- 포트 9098에 대한 각 인바운드 규칙 또는 아웃바운드 규칙에 대해 다음을 제공합니다.
  - 유형에서 Custom TCP(사용자 지정 TCP)를 선택합니다.
  - 포트 범위에 9098을 입력합니다.
  - 소스로 *clusterSecurityGroups* 중 하나를 제공합니다.
- 포트 443에 대한 각 인바운드 규칙에 대해 유형으로 HTTPS를 선택합니다.

Lambda가 Amazon MSK 주제에서 읽을 수 있도록 IAM 역할 생성

Lambda가 Amazon MSK 주제에서 읽기 위한 인증 요구 사항을 파악한 다음 정책에서 정의합니다. Lambda가 이러한 권한을 사용할 수 있도록 승인하는 역할인 *lambdaAuthRole*을 생성합니다. `kafka-cluster` IAM 작업을 사용하여 Amazon MSK 클러스터에서 작업 권한을 부여합니다. 그런 다음, Lambda가 Amazon MSK 클러스터를 검색하고 연결하는 데 필요한 Amazon MSK `kafka` 및 Amazon EC2 작업을 수행하도록 권한을 부여하고, Lambda가 수행한 작업을 기록할 수 있도록 CloudWatch 작업을 수행하도록 권한을 부여합니다.

Lambda가 Amazon MSK에서 읽기 위한 인증 요구 사항을 설명하려면 다음을 수행하세요.

1. Lambda가 Kafka 소비자 그룹을 사용하여 Amazon MSK 클러스터의 Kafka 주제에서 읽을 수 있도록 허용하는 IAM 정책 문서(JSON 문서)인 *clusterAuthPolicy*를 작성합니다. Lambda는 읽을 때 Kafka 소비자 그룹을 설정해야 합니다.

   사전 요구 사항에 맞게 다음 템플릿 변경

```
{
    "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"
            ]
        }
    ]
}
```

자세한 내용은 the section called “IAM 역할 기반 인증”을 참조하세요. 정책을 작성할 때

- *region*과 *account-id*로 Amazon MSK 클러스터를 호스팅하는 항목을 제공합니다.
- *mskClusterName*으로 Amazon MSK 클러스터의 이름을 제공합니다.
- *cluster-uuid*로 Amazon MSK 클러스터의 ARN에 UUID를 제공합니다.
- *mskTopicName*으로 Kafka 주제의 이름을 제공합니다.
- *mskGroupName*으로 Kafka 소비자 그룹의 이름을 제공합니다.

2. Lambda가 Amazon MSK 클러스터를 검색 및 연결하고 해당 이벤트를 기록하는 데 필요한 Amazon MSK, Amazon EC2 및 CloudWatch 권한을 식별합니다.

   `AWSLambdaMSKExecutionRole` 관리형 정책은 필요한 권한을 허용적으로 정의합니다. 다음 단계에서 사용합니다.

   프로덕션 환경에서는 `AWSLambdaMSKExecutionRole`을 평가하여 최소 권한 원칙에 따라 실행 역할 정책을 제한한 다음 이 관리형 정책을 대체하는 역할에 대한 정책을 작성합니다.

IAM 정책 언어에 대한 자세한 내용은 IAM 설명서를 참조하세요.

정책 문서를 작성했으니 이제 IAM 정책을 생성하여 역할에 연결할 수 있습니다. 다음 절차에 따라 콘솔을 사용하여 이 작업을 수행할 수 있습니다.

정책 문서에서 IAM 정책을 생성하려면 다음을 수행하세요.

1. AWS Management Console에 로그인하여 https://console.aws.amazon.com/iam/ 에서 IAM 콘솔을 엽니다.
2. 왼쪽의 탐색 창에서 정책을 선택합니다.
3. 정책 생성을 선택합니다.
4. 정책 편집기 섹션에서 JSON 옵션을 선택합니다.
5. *clusterAuthPolicy*를 붙여넣습니다.
6. 정책에 권한 추가를 완료했으면 다음을 선택합니다.
7. 검토 및 생성 페이지에서 생성하는 정책에 대한 정책 이름과 설명(선택 사항)을 입력합니다. 이 정책에 정의된 권한을 검토하여 정책이 부여한 권한을 확인합니다.
8. 정책 생성을 선택하고 새로운 정책을 저장합니다.

자세한 내용은 IAM 설명서의 IAM 정책 생성을 참조하세요.

이제 적절한 IAM 정책이 있으므로 역할을 생성하고 이에 IAM 정책을 연결합니다. 다음 절차에 따라 콘솔을 사용하여 이 작업을 수행할 수 있습니다.

IAM 콘솔에서 실행 역할을 생성하려면

1. IAM 콘솔에서 역할(Roles) 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형에서 AWS 서비스를 선택합니다.
4. 사용 사례에서 Lambda를 선택합니다.
5. Next(다음)를 선택합니다.
6. 다음 정책을 선택합니다.
   - *clusterAuthPolicy*
   - `AWSLambdaMSKExecutionRole`
7. Next(다음)를 선택합니다.
8. 역할 이름에 *lambdaAuthRole*을 입력한 다음 역할 생성을 선택합니다.

자세한 내용은 the section called “실행 역할(함수가 다른 리소스에 액세스할 수 있는 권한)” 단원을 참조하십시오.

Amazon MSK 주제에서 읽을 Lambda 함수 생성

IAM 역할을 사용하도록 구성된 Lambda 함수를 생성합니다. 콘솔을 사용하여 Lambda 함수를 생성할 수 있습니다.

인증 구성을 사용하여 Lambda 함수를 생성하려면 다음을 수행하세요.

1. Lambda 콘솔을 열고 헤더에서 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.
3. 함수 이름으로 원하는 적절한 이름을 제공합니다.
4. 런타임으로 이 자습서에 제공된 코드를 사용하려면 지원되는 최신 버전의 `Node.js`를 선택합니다.
5. 기본 실행 역할 변경을 선택합니다.
6. 기존 역할 사용을 선택합니다.
7. 기존 역할로 *`lambdaAuthRole`*을 선택합니다.

프로덕션 환경에서는 일반적으로 Amazon MSK 이벤트를 의미 있게 처리하기 위해 Lambda 함수의 실행 역할에 추가 정책을 추가해야 합니다. 역할에 정책을 추가하는 방법에 대한 자세한 내용은 IAM 설명서의 ID 권한 추가 또는 제거를 참조하세요.

Lambda 함수에 대한 이벤트 소스 매핑 생성

Amazon MSK 이벤트 소스 매핑은 적절한 Amazon MSK 이벤트가 발생할 때 Lambda를 간접적으로 호출하는 데 필요한 정보를 Lambda 서비스에 제공합니다. 콘솔을 사용하여 Amazon MSK 매핑을 생성할 수 있습니다. Lambda 트리거를 생성하면 이벤트 소스 매핑이 자동으로 설정됩니다.

Lambda 트리거 및 이벤트 소스 매핑을 생성하려면 다음을 수행하세요.

1. Lambda 함수의 개요 페이지로 이동합니다.
2. 함수 개요 섹션의 왼쪽 하단에서 트리거 추가를 선택합니다.
3. 소스 선택 드롭다운에서 Amazon MSK를 선택합니다.
4. 인증을 설정하지 마세요.
5. MSK 클러스터로 클러스터의 이름을 선택합니다.
6. 배치 크기로 1을 입력합니다. 이 단계를 수행하면 이 기능을 더 쉽게 테스트할 수 있지만 프로덕션에서는 이상적인 값이 아닙니다.
7. 주제 이름으로 Kafka 주제의 이름을 입력합니다.

8. 소비자 그룹 ID로 Kafka 소비자 그룹의 ID를 제공합니다.

스트리밍 데이터를 읽도록 Lambda 함수 업데이트

Lambda는 이벤트 메서드 파라미터를 통해 Kafka 이벤트에 대한 정보를 제공합니다. Amazon MSK 이벤트의 예제 구조는 the section called “ 예제 이벤트” 섹션을 참조하세요. Lambda가 전달한 Amazon MSK 이벤트를 해석하는 방법을 이해한 후에는 제공된 정보를 사용하도록 Lambda 함수 코드를 변경할 수 있습니다.

테스트 목적으로 Lambda Amazon MSK 이벤트의 내용을 기록하려면 Lambda 함수에 다음 코드를 제공합니다.

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)
      })
    }
}
```

콘솔을 사용하여 Lambda에 함수 코드를 제공할 수 있습니다.

Lambda 함수 코드를 업데이트하려면 다음을 수행하세요.

1. Lambda 함수의 개요 페이지로 이동합니다.
2. 코드 탭을 선택합니다.
3. 제공된 코드를 코드 소스 IDE에 입력합니다.
4. 코드 소스 탐색 바에서 배포를 선택합니다.

Lambda 함수를 테스트하여 Amazon MSK 주제에 연결되어 있는지 확인합니다.

이제 CloudWatch 이벤트 로그를 검사하여 Lambda가 이벤트 소스에 의해 간접적으로 호출되고 있는지 여부를 확인할 수 있습니다.

Lambda 함수가 간접적으로 호출되고 있는지 확인하려면 다음을 수행하세요.

1. Kafka 관리 호스트를 사용하여 `kafka-console-producer` CLI를 통해 Kafka 이벤트를 생성합니다. 자세한 내용은 Kafka 설명서의 Write some events into the topic을 참조하세요. 이전 단계에서 정의된 이벤트 소스 매핑에 대해 배치 크기로 정의된 배치를 채우기에 충분한 이벤트를 전송합니다. 그렇지 않으면 Lambda가 추가 정보가 간접적으로 호출될 때까지 기다립니다.
2. 함수가 실행되면 Lambda는 CloudWatch에 발생한 일을 기록합니다. 콘솔에서 Lambda 함수의 세부 정보 페이지로 이동합니다.
3. [구성(Configuration)] 탭을 선택합니다.
4. 사이드바에서 모니터링 및 운영 도구를 선택합니다.
5. 로깅 구성에서 CloudWatch 로그 그룹을 식별합니다. 로그 그룹은 `/aws/lambda`로 시작해야 합니다. 로그 그룹의 링크를 선택합니다.
6. CloudWatch 콘솔의 로그 이벤트에서 Lambda가 로그 스트림으로 전송한 로그 이벤트가 있는지 검사합니다. 다음 이미지와 같이 Kafka 이벤트의 메시지가 포함된 로그 이벤트가 있는지 확인합니다. 있으면 Lambda 이벤트 소스 매핑을 사용하여 Lambda 함수를 Amazon MSK에 성공적으로 연결한 것입니다.

```
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
```

예제 이벤트

Lambda는 함수를 호출할 때 이벤트 파라미터의 메시지 배치를 보냅니다. 이벤트 페이로드에는 메시지 배열이 포함됩니다. 각 배열 항목에는 Amazon MSK 주제 및 파티션 식별자에 대한 세부 정보와 함께 타임스탬프 및 base64로 인코딩된 메시지가 포함됩니다.

```
{
```

```
   "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 클러스터 인증

Lambda는 Amazon MSK 클러스터에 액세스하고 레코드를 검색하고 다른 태스크를 수행할 수 있는 권한이 필요합니다. Amazon MSK는 MSK 클러스터에 대한 클라이언트 액세스를 제어하는 몇 가지 옵션을 지원합니다.

클러스터 액세스 옵션

- 인증되지 않은 액세스
- SASL/SCRAM 인증
- IAM 역할 기반 인증
- 상호 TLS 인증
- mTLS 암호 구성
- Lambda이 부트스트랩 브로커를 선택하는 방법

인증되지 않은 액세스

인터넷을 통해 클러스터에 액세스하는 클라이언트가 없는 경우 인증되지 않은 액세스를 사용할 수 있습니다.

SASL/SCRAM 인증

Amazon MSK는 전송 계층 보안(TLS) 암호화를 통해 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism(SASL/SCRAM) 인증을 지원합니다. Lambda가 클러스터에 연결하려면 인증 자격 증명 (사용자 이름 및 암호)을 AWS Secrets Manager 비밀 정보에 저장합니다.

Secrets Manager 사용에 대한 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 AWS Secrets Manager를 사용한 사용자 이름 및 암호 인증을 참조하세요.

Amazon MSK는 SASL/PLAIN 인증을 지원하지 않습니다.

IAM 역할 기반 인증

IAM을 사용하여 MSK 클러스터에 연결하는 클라이언트의 자격 증명을 인증할 수 있습니다. IAM 인증이 MSK 클러스터에서 활성 상태이고 인증을 위한 암호를 제공하지 않으면 Lambda는 자동으로 IAM 인증을 사용하도록 기본 설정됩니다. 사용자 또는 역할 기반 정책을 생성하고 배포하려면 IAM 콘솔 또는 API를 사용합니다. 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 IAM 액세스 제어를 참조하세요.

Lambda가 MSK 클러스터에 연결하고 레코드를 읽고 기타 필수 작업을 수행할 수 있도록 하려면 함수의 실행 역할에 다음 권한을 추가합니다.

```
{
    "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"
            ]
        }
    ]
}
```

이러한 권한의 범위를 특정 클러스터, 주제 및 그룹으로 설정할 수 있습니다. 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 Amazon MSK Kafka 작업을 참조하세요.

상호 TLS 인증

상호 TLS(mTLS)는 클라이언트와 서버 간의 양방향 인증을 제공합니다. 클라이언트는 서버가 클라이언트를 확인할 수 있도록 서버에 인증서를 보내고, 서버는 클라이언트가 서버를 확인할 수 있도록 클라이언트에 인증서를 보냅니다.

Amazon MSK의 경우 Lambda가 클라이언트 역할을 합니다. 클라이언트 인증서(Secrets Manager의 비밀 정보)를 구성하여 MSK 클러스터의 브로커로 Lambda를 인증합니다. 클라이언트 인증서는 서버의 신뢰 저장소에 있는 CA에서 서명해야 합니다. MSK 클러스터는 서버 인증서를 Lambda로 전송하여 Lambda로 브로커를 인증합니다. 서버 인증서는 AWS 신뢰 저장소에 있는 인증 기관(CA)에서 서명해야 합니다.

클라이언트 인증서를 생성하는 방법에 대한 지침은 Introducing mutual TLS authentication for Amazon MSK as an event source를 참조하세요.

Amazon MSK는 자체 서명된 서버 인증서를 지원하지 않습니다. Amazon MSK의 모든 브로커는 기본적으로 Lambda가 신뢰하는 Amazon Trust Services CA에서 서명한 퍼블릭 인증서를 사용하기 때문입니다.

Amazon MSK용 mTLS에 대한 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 상호 TLS 인증을 참조하세요.

mTLS 암호 구성

CLIENT_CERTIFICATE_TLS_AUTH 비밀 정보에 인증서 필드와 프라이빗 키 필드가 필요합니다. 암호화된 프라이빗 키의 경우 비밀 정보에 프라이빗 키 암호가 필요합니다. 인증서와 프라이빗 키는 모두 PEM 형식이어야 합니다.

> **Note**
>
> Lambda는 PBES1(PBES2가 아님) 프라이빗 키 암호화 알고리즘을 지원합니다.

인증서 필드에는 클라이언트 인증서부터 시작하여 중간 인증서가 이어지고 루트 인증서로 끝나는 인증서 목록이 포함되어야 합니다. 각 인증서는 다음 구조의 새 줄에서 시작해야 합니다.

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager는 최대 65,536바이트의 보안 정보를 지원하므로 긴 인증서 체인을 위한 충분한 공간입니다.

프라이빗 키는 다음 구조의 PKCS #8 형식이어야 합니다.

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

암호화된 프라이빗 키의 경우 다음 구조를 사용합니다.

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

다음 예제에서는 암호화된 프라이빗 키를 사용한 mTLS 인증용 비밀 정보 콘텐츠를 표시합니다. 암호화된 개인 키의 경우 비밀 정보에 프라이빗 키 암호를 포함합니다.

```
{
 "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-----"
}
```

Lambda이 부트스트랩 브로커를 선택하는 방법

Lambda는 클러스터에서 사용할 수 있는 인증 방법 및 인증을 위한 암호 제공 여부를 기반으로 부트스트랩 브로커를 선택합니다. mTLS 또는 SASL/SCRAM에 대한 암호를 제공하면 Lambda가 자동으로 해당 인증 방법을 선택합니다. 암호를 제공하지 않으면 Lambda가 클러스터에서 활성화된 가장 강력한 인증 방법을 선택합니다. 다음은 Lambda가 가장 강력한 인증부터 가장 약한 인증까지 브로커를 선택하는 우선 순위입니다.

- mTLS(MTL에 제공되는 암호)
- SASL/SCRAM(SASL/SCRAM에 대해 제공되는 암호)
- SASL IAM(암호가 제공되지 않았으며 IAM 인증이 활성화됨)
- 인증되지 않은 TLS(암호가 제공되지 않고 IAM 인증이 활성화되지 않음)
- 일반 텍스트(암호가 제공되지 않고 IAM 인증 및 인증되지 않은 TLS가 모두 활성화되지 않음)

> **Note**
>
> Lambda가 가장 안전한 브로커 유형에 연결할 수 없는 경우 Lambda는 다른 (약한) 브로커 유형에 연결을 시도하지 않습니다. Lambda가 더 약한 브로커 유형을 선택하게 하려면 클러스터에서 더 강력한 인증 방법을 모두 비활성화하십시오.

API 액세스 및 권한 관리

Amazon MSK 클러스터에 액세스하는 것 외에도 함수에는 다양한 Amazon MSK API 작업을 수행할 수 있는 권한이 필요합니다. 이러한 권한을 함수의 실행 역할에 추가합니다. 사용자가 Amazon MSK API 작업에 액세스해야 하는 경우 사용자 또는 역할에 대한 자격 증명 정책에 필요한 권한을 추가합니다.

실행 역할에 다음과 같은 권한을 각각 수동으로 추가할 수 있습니다. 또는 AWS 관리형 정책 AWSLambdaMSKExecutionRole을 실행 역할에 연결할 수도 있습니다. `AWSLambdaMSKExecutionRole` 정책에는 아래 나열된 모든 필수 API 작업 및 VPC 권한이 포함되어 있습니다.

필요한 Lambda 함수 실행 역할 권한

Amazon CloudWatch Logs의 로그 그룹에 로그를 생성하고 저장하려면 Lambda 함수의 실행 역할에 다음 권한이 있어야 합니다.

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Lambda가 사용자를 대신하여 Amazon MSK 클러스터에 액세스하기 위해서는 Lambda 함수의 실행 역할에 다음 권한이 있어야 합니다.

- kafka:DescribeCluster
- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers
- kafka:DescribeVpcConnection: 계정 간 이벤트 소스 매핑에만 필요합니다.
- kafka:ListVpcConnections: 실행 역할에는 필요하지 않지만 계정 간 이벤트 소스 매핑을 생성하는 IAM 보안 주체에는 필요합니다.

`kafka:DescribeCluster`과 `kafka:DescribeClusterV2` 중 하나만 추가하면 됩니다. 프로비저닝된 MSK 클러스터의 경우 두 권한 모두 작동합니다. 서버리스 MSK 클러스터의 경우 `kafka:DescribeClusterV2`을 사용해야 합니다.

> **Note**
>
> Lambda는 결국 관련된 `AWSLambdaMSKExecutionRole` 관리 정책에서 `kafka:DescribeCluster` 권한을 제거할 계획입니다. 이 정책을 사용하면, 대신 `kafka:DescribeClusterV2`을 사용하기 위해서는 `kafka:DescribeCluster`을 이용하는 애플리케이션을 모두 마이그레이션해야 합니다.

VPC 권한

VPC 내의 사용자만 Amazon MSK 클러스터에 액세스할 수 있는 경우 Lambda 함수에는 Amazon VPC 리소스에 액세스할 수 있는 권한이 있어야 합니다. 이러한 리소스에는 VPC, 서브넷, 보안 그룹 및 네트워크 인터페이스가 있습니다. 리소스에 액세스하려면 함수의 실행 역할에 다음 권한이 주어져야 합니다. 이러한 권한은 AWSLambdaMSKExecutionRole AWS 관리형 정책에 포함됩니다.

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

선택적 Lambda 함수 권한

Lambda 함수에 또한 다음 권한이 필요할 수 있습니다.

- SASL/SCRAM 인증을 사용하는 경우 SCRAM 암호에 액세스하십시오.
- Secrets Manager 비밀 정보를 설명합니다.
- AWS Key Management Service(AWS KMS) 고객 관리형 키에 액세스합니다.
- 실패한 간접 호출 기록을 대상으로 전송합니다.

Secrets Manager 및 AWS KMS 권한

Amazon MSK 브로커에 대해 구성하는 액세스 제어 유형에 따라 Lambda 함수에는 SCRAM 비밀(SASL/SCRAM 인증 이용 시) 또는 Secrets Manager 암호 정보에 액세스하여 AWS KMS 고객 관리형

키를 복호화할 수 있는 권한이 필요할 수도 있습니다. 리소스에 액세스하려면 함수의 실행 역할에 다음 권한이 주어져야 합니다.

- kafka:ListScramSecrets
- secretsmanager:GetSecretValue
- kms:Decrypt

실행 역할에 권한 추가

IAM 콘솔을 사용하여 AWS 관리형 정책 AWSLambdaMSKExecutionRole을 실행 역할에 추가하려면 다음 단계를 따릅니다.

AWS 관리형 정책을 추가하려면

1. IAM 콘솔에서 정책 페이지를 엽니다.
2. 검색 상자에 정책 이름(`AWSLambdaMSKExecutionRole`)을 입력합니다.
3. 목록에서 정책을 선택한 다음 정책 작업(Policy actions), 연결(Attach)을 선택합니다.
4. 정책 연결(Attach policy) 페이지의 목록에서 실행 역할을 선택한 다음 정책 연결을 선택합니다.

IAM 정책을 사용하여 사용자에게 액세스 권한 부여

기본적으로 사용자 및 역할은 Amazon MSK API 작업을 수행할 수 있는 권한이 없습니다. 조직 또는 계정의 사용자에게 액세스 권한을 부여하려면 자격 증명 기반 정책을 추가 혹은 업데이트할 수 있습니다. 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 Amazon MSK 자격 증명 기반 정책 예제를 참조하세요.

인증 및 권한 부여 오류

Amazon MSK 클러스터의 데이터를 사용하는 데 필요한 권한이 누락된 경우 Lambda는 LastProcessingResult 아래의 이벤트 소스 매핑에 다음 오류 메시지 중 하나를 표시합니다.

오류 메시지

- 클러스터가 Lambda를 인증하지 못함
- SASL 인증 실패
- 서버가 Lambda를 인증하지 못함
- 제공된 인증서 또는 프라이빗 키가 잘못됨

클러스터가 Lambda를 인증하지 못함

SASL/SCRAM 또는 mMTS의 경우 이 오류는 제공된 사용자에게 다음 필수 Kafka 액세스 제어 목록(ACL) 권한이 모두 있지는 않음을 나타냅니다.

- DescribeConfigs 클러스터
- 그룹 설명
- 그룹 읽기
- 주제 설명
- Thread-Topic

IAM 액세스 제어의 경우 함수의 실행 역할에 그룹 또는 주제에 액세스하는 데 필요한 하나 이상의 권한이 없습니다. the section called “IAM 역할 기반 인증”에서 필요한 권한 목록을 검토합니다.

필수 Kafka 클러스터 권한이 있는 Kafka ACL 또는 IAM 정책을 생성할 때 주제와 그룹을 리소스로 지정합니다. 주제 이름은 이벤트 소스 매핑의 주제와 일치해야 합니다. 그룹 이름은 이벤트 소스 매핑의 UUID와 일치해야 합니다.

실행 역할에 필요한 권한을 추가한 후 변경 사항이 적용되기까지 몇 분 정도 소요될 수 있습니다.

SASL 인증 실패

SASL/SCRAM의 경우 이 오류는 제공된 사용자 이름과 암호가 유효하지 않음을 나타냅니다.

IAM 액세스 제어의 경우 실행 역할에 MSK 클러스터에 대한 `kafka-cluster:Connect` 권한이 없습니다. 역할에 이 권한을 추가하고 클러스터의 Amazon 리소스 이름(ARN)을 리소스로 지정합니다.

이 오류가 간헐적으로 발생하는 경우가 있습니다. TCP 연결 수가 Amazon MSK 서비스 할당량을 초과하면 클러스터는 연결을 거부합니다. Lambda는 연결이 성공할 때까지 취소하고 다시 시도합니다. Lambda가 클러스터에 연결하고 레코드를 폴링하면 마지막 처리 결과가 `OK`로 변경됩니다.

서버가 Lambda를 인증하지 못함

이 오류는 Amazon MSK Kafka 브로커가 Lambda로 인증하지 못했음을 나타냅니다. 이 오류는 다음과 같은 이유로 발생할 수 있습니다.

- mTLS 인증을 위한 클라이언트 인증서를 제공하지 않았습니다.
- 클라이언트 인증서를 제공했지만 브로커가 mTLS를 사용하도록 구성되지 않았습니다.
- 브로커가 클라이언트 인증서를 신뢰하지 않습니다.

제공된 인증서 또는 프라이빗 키가 잘못됨

이 오류는 Amazon MSK 소비자가 제공된 인증서 또는 프라이빗 키를 사용할 수 없음을 나타냅니다. 인증서와 키가 PEM 형식을 사용하고 프라이빗 키 암호화가 PBES1 알고리즘을 사용하는지 확인합니다.

네트워크 구성

Lambda가 Kafka 클러스터를 이벤트 소스로 사용하려면 클러스터가 있는 Amazon VPC에 액세스해야 합니다. Lambda가 VPC에 액세스할 수 있도록 AWS PrivateLink VPC 엔드포인트를 배포하는 것이 좋습니다. Lambda 및 AWS Security Token Service(AWS STS)에 대한 엔드포인트를 배포합니다. 브로커가 인증을 사용하는 경우 Secrets Manager용 VPC 엔드포인트도 배포합니다. 장애 시 대상을 구성한 경우 대상 서비스를 위한 VPC 엔드포인트도 배포합니다.

또는 Kafka 클러스터와 연결된 VPC에 퍼블릭 서브넷당 하나의 NAT 게이트웨이가 포함되는지 확인합니다. 자세한 내용은 the section called “VPC 함수에 대한 인터넷 액세스” 단원을 참조하십시오.

또한 VPC 엔드포인트를 사용하는 경우 프라이빗 DNS 이름을 활성화하도록 구성해야 합니다.

MSK 클러스터에 대한 이벤트 소스 매핑을 생성하는 경우 Lambda는 클러스터 VPC의 서브넷 및 보안 그룹에 대해 탄력적 네트워크 인터페이스(ENI)가 이미 존재하는지 확인합니다. Lambda가 기존 ENI를 찾으면 이를 재사용하려고 시도합니다. 그렇지 않으면 Lambda가 이벤트 소스에 연결하고 함수를 호출하기 위해 새 ENI를 생성합니다.

> **Note**
>
> Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. 이러한 VPC는 서비스에 의해 자동으로 유지 관리되며 고객에게는 표시되지 않습니다. 또한 함수를 Amazon VPC에 연결할 수도 있습니다. 어느 경우든 함수의 VPC 구성은 이벤트 소스 매핑에 영향을 미치지 않습니다. 이벤트 소스의 VPC 구성에 따라 Lambda가 이벤트 소스에 연결되는 방식이 결정됩니다.

Amazon VPC 구성은 Amazon MSK API를 통해 검색할 수 있습니다. create-event-source-mapping 명령을 사용하여 설정하는 동안 이 옵션을 구성할 필요가 없습니다.

네트워크 구성에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Setting up AWS Lambda with an Apache Kafka cluster within a VPC를 참조하세요.

VPC 보안 그룹 규칙

최소한 다음 규칙을 사용하여 클러스터가 포함된 Amazon VPC의 보안 그룹을 구성합니다.

- 인바운드 규칙 - 이벤트 소스에 지정된 보안 그룹에 대해 Amazon MSK 브로커 포트(일반 텍스트의 경우 9092, TLS의 경우 9094, SASL의 경우 9096, IAM의 경우 9098)의 모든 트래픽을 허용합니다.
- 아웃바운드 규칙 - 모든 대상에 대해 포트 443의 모든 트래픽을 허용합니다. 이벤트 소스에 지정된 보안 그룹에 대해 Amazon MSK 브로커 포트(일반 텍스트의 경우 9092, TLS의 경우 9094, SASL의 경우 9096, IAM의 경우 9098)의 모든 트래픽을 허용합니다.
- NAT 게이트웨이 대신 VPC 엔드포인트를 사용하는 경우 VPC 엔드포인트와 연결된 보안 그룹은 이벤트 소스의 보안 그룹에서 포트 443의 모든 인바운드 트래픽을 허용해야 합니다.

VPC 엔드포인트 작업

VPC 엔드포인트를 사용하는 경우 함수를 호출하는 API 직접 호출은 ENI를 사용하여 이러한 엔드포인트를 통해 라우팅됩니다. Lambda 서비스 보안 주체는 해당 ENI를 사용하는 모든 역할과 함수에서 `sts:AssumeRole`과 `lambda:InvokeFunction`을 호출해야 합니다.

기본적으로 VPC 엔드포인트에는 개방적인 IAM 정책이 있습니다. 모범 사례는 특정 보안 주체만 해당 엔드포인트를 사용하여 필요한 작업을 수행할 수 있도록 이러한 정책을 제한하는 것입니다. 이벤트 소스 매핑이 Lambda 함수를 호출할 수 있도록 하려면 VPC 엔드포인트 정책에서 Lambda 서비스 원칙이 `sts:AssumeRole`과 `lambda:InvokeFunction`을 호출할 수 있도록 허용해야 합니다. 조직 내에서 발생하는 API 직접 호출만 허용하도록 VPC 엔드포인트 정책을 제한하면 이벤트 소스 매핑이 제대로 작동하지 않습니다.

다음 예제 VPC 엔드포인트 정책은 AWS STS 및 Lambda 엔드포인트에 대해 Lambda 서비스 보안 주체에 필요한 액세스 권한을 부여하는 방법을 안내합니다.

Example VPC 엔드포인트 정책 - AWS STS 엔드포인트

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
```

```
}
```

Example VPC 엔드포인트 정책 - Lambda 엔드포인트

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Kafka 브로커 클러스터가 인증을 사용하는 경우 Secrets Manager 엔드포인트에 대한 VPC 엔드포인트 정책을 제한할 수도 있습니다. Secrets Manager API를 호출하기 위해 Lambda는 Lambda 서비스 보안 주체가 아닌 함수 역할을 사용합니다. 다음 예제는 Secrets Manager 엔드포인트 정책을 보여줍니다.

Example VPC 엔드포인트 정책 – Secrets Manager 엔드포인트

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

장애 발생 시 대상을 구성한 경우 Lambda는 함수의 역할을 사용하여 Lambda 관리형 ENI를 사용하여 `s3:PutObject`, `sns:Publish` 또는 `sqs:sendMessage`를 호출합니다.

Amazon MSK를 이벤트 소스로 추가

이벤트 소스 매핑을 생성하려면 Lambda 콘솔, AWS SDK 또는 AWS Command Line Interface(AWS CLI)를 사용해 Amazon MSK를 Lambda 함수 트리거로 추가합니다. Amazon MSK를 트리거로 추가하면 Lambda는 Lambda 함수의 VPC 설정이 아니라 Amazon MSK 클러스터의 VPC 설정을 가정합니다.

이 단원에서는 Lambda 콘솔 및 AWS CLI를 사용해 이벤트 소스 매핑을 생성하는 방법을 설명합니다.

필수 조건

- Amazon MSK 클러스터와 Kafka 주제. 자세한 내용은 Amazon Managed Streaming for Apache Kafka 개발자 안내서의 Amazon MSK 사용 시작하기를 참조하세요.
- MSK 클러스터에서 사용하는 AWS 리소스에 액세스할 수 있는 권한이 있는 실행 역할입니다.

사용자 지정이 가능한 소비자 그룹 ID

Kafka를 이벤트 소스로 설정할 때 소비자 그룹 ID를 지정할 수 있습니다. 이 소비자 그룹 ID는 Lambda 함수에 가입하려는 Kafka 소비자 그룹의 기존 식별자입니다. 이 기능을 사용하여 진행 중인 모든 Kafka 레코드 처리 설정을 다른 소비자에서 Lambda로 원활하게 마이그레이션할 수 있습니다.

소비자 그룹 ID를 지정하고 해당 소비자 그룹 내에 다른 활성 폴러가 있는 경우 Kafka는 모든 소비자에게 메시지를 배포합니다. 즉, Lambda는 Kafka 주제에 대한 모든 메시지를 수신하지는 않습니다. Lambda가 주제에 있는 모든 메시지를 처리하도록 하려면 해당 소비자 그룹의 다른 모든 폴러를 끄십시오.

또한 소비자 그룹 ID를 지정했는데 Kafka가 동일한 ID를 가진 유효한 기존 소비자 그룹을 찾으면 Lambda는 이벤트 소스 매핑을 위한 `StartingPosition` 파라미터를 무시합니다. 대신 Lambda는 소비자 그룹의 커밋된 오프셋에 따라 레코드 처리를 시작합니다. 소비자 그룹 ID를 지정했는데 Kafka가 기존 소비자 그룹을 찾을 수 없는 경우, Lambda는 `StartingPosition`에서 지정된 대로 이벤트 소스를 구성합니다.

지정하는 소비자 그룹 ID는 모든 Kafka 이벤트 소스 중에서 고유해야 합니다. 지정된 소비자 그룹 ID로 Kafka 이벤트 소스 매핑을 생성한 후에는 이 값을 업데이트할 수 없습니다.

Amazon MSK 트리거 추가(콘솔)

다음 단계에 따라 Amazon MSK 클러스터 및 Kafka 주제를 Lambda 함수의 트리거로 추가합니다.

Lambda 함수에 Amazon MSK 트리거를 추가하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.

2. Lambda 함수의 이름을 선택합니다.
3. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.
4. 트리거 구성에서 다음을 수행합니다.

   a. MSK 트리거 유형을 선택합니다.
   b. MSK 클러스터에 해당 클러스터를 선택합니다.
   c. 배치 크기(Batch size)에 단일 배치에서 검색할 최대 메시지 수를 입력합니다.
   d. Batch window에서 Lambda가 함수를 호출하기 전에 레코드를 수집하는 데 걸리는 최대 시간(초)을 입력합니다.
   e. 주제 이름에 Kafka 주제의 이름을 입력합니다.
   f. (선택 사항) Consumer group ID에서 가입할 Kafka 소비자 그룹의 ID를 입력합니다.
   g. (선택 사항) 시작 위치의 경우 최신 레코드에서 스트림 읽기를 시작하려면 최신을 선택하고, 사용 가능한 가장 빠른 레코드에서 시작하려면 수평 트리밍을 선택하고, 읽기를 시작할 타임스탬프를 지정하려면 타임스탬프를 선택합니다.
   h. (선택 사항) 인증(Authentication)에서 MSK 클러스터의 브로커로 인증하기 위한 비밀 키를 선택합니다.
   i. 테스트를 위해 트리거를 비활성화된 상태에서 생성하려면(권장됨) 트리거 사용을 선택 해제합니다. 또는 트리거를 즉시 사용하려면 트리거 사용을 선택합니다.
5. 트리거를 생성하려면 추가를 선택합니다.

Amazon MSK 트리거 추가(AWS CLI)

다음 예제 AWS CLI 명령을 사용하여 Lambda 함수에 대한 Amazon MSK 트리거를 생성하고 확인합니다.

AWS CLI를 사용하여 트리거 생성

Example — IAM 인증을 사용하는 클러스터의 이벤트 소스 매핑 생성

다음 예제에서는 `create-event-source-mapping` AWS CLI 명령을 사용해 `my-kafka-function`이라는 Lambda 함수를 `AWSKafkaTopic`이라는 Kafka 주제에 매핑합니다. 주제의 시작 위치를 `LATEST`으로 설정합니다. 클러스터가 IAM 역할 기반 인증을 사용하는 경우 SourceAccessConfiguration 객체가 필요하지 않습니다. 예제

```
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 인증을 사용하는 클러스터의 이벤트 소스 매핑 생성

클러스터가 SASL/SCRAM 인증을 사용하는 경우 SourceAccessConfiguration 객체를 포함하며 이는 SASL_SCRAM_512_AUTH 및 Secrets Manager 비밀 ARN을 지정합니다.

```
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 인증을 사용하는 클러스터의 이벤트 소스 매핑 생성

클러스터가 mTLS을 사용하는 경우 SourceAccessConfiguration 객체를 포함하며 이는 CLIENT_CERTIFICATE_TLS_AUTH 및 Secrets Manager 비밀 ARN을 지정합니다.

```
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"}]'
```

자세한 내용은 CreateEventSourceMapping API 참조 문서를 참조하세요.

AWS CLI를 사용하여 상태 확인

다음 예제에서는 `get-event-source-mapping` AWS CLI 명령을 사용해 생성한 이벤트 소스 매핑의 상태를 설명합니다.

```
aws lambda get-event-source-mapping \
  --uuid 6d9bce8e-836b-442c-8070-74e77903c815
```

계정 간 이벤트 소스 매핑 생성

다중 VPC 프라이빗 연결을 사용하여 Lambda 함수를 다른 AWS 계정의 클러스터에 프로비저닝된 MSK 클러스터에 연결할 수 있습니다. 다중 VPC 연결은 AWS PrivateLink를 사용하며, 이는 모든 트래픽을 AWS 네트워크 내에 유지합니다.

> **Note**
>
> 서버리스 MSK 클러스터에는 계정 간 이벤트 소스 매핑을 생성할 수 없습니다.

계정 간 이벤트 소스 매핑을 생성하려면 먼저 MSK 클러스터의 다중 VPC 연결을 구성해야 합니다. 이벤트 소스 매핑을 생성할 때는 다음 예에서 표시된 것처럼 클러스터 ARN 대신 관리형 VPC 연결 ARN을 사용합니다. CreateEventSourceMapping 작업도 MSK 클러스터가 사용하는 인증 유형에 따라 달라집니다.

Example — IAM 인증을 사용하는 클러스터의 계정 간 이벤트 소스 매핑 생성

클러스터가 IAM 역할 기반 인증을 사용하는 경우 SourceAccessConfiguration 객체가 필요하지 않습니다. 예제

```
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 인증을 사용하는 클러스터의 계정 간 이벤트 소스 매핑 생성

클러스터가 SASL/SCRAM 인증을 사용하는 경우 SourceAccessConfiguration 객체를 포함하며 이는 SASL_SCRAM_512_AUTH 및 Secrets Manager 비밀 ARN을 지정합니다.

SASL/SCRAM 인증을 통한 계정 간 Amazon MSK 이벤트 소스 매핑에 암호를 사용하는 방법에는 두 가지가 있습니다.

- Lambda 함수 계정에서 시크릿을 생성하고 클러스터 암호와 동기화합니다. 로테이션을 생성하여 두 암호가 동기화된 상태를 유지합니다. 이 옵션을 사용하면 함수 계정에서 시크릿을 제어할 수 있습니다.

- MSK 클러스터와 연결된 암호를 사용합니다. 이 암호는 Lambda 함수 계정에 대한 교차 계정 액세스를 허용해야 합니다. 자세한 내용은 다른 계정에 있는 사용자의 AWS Secrets Manager 암호에 대한 권한을 참조하세요.

```
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 인증을 사용하는 클러스터의 계정 간 이벤트 소스 매핑 생성

클러스터가 mTLS을 사용하는 경우 SourceAccessConfiguration 객체를 포함하며 이는 CLIENT_CERTIFICATE_TLS_AUTH 및 Secrets Manager 비밀 ARN을 지정합니다. 암호는 클러스터 계정 또는 Lambda 함수 계정에 저장할 수 있습니다.

```
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"}]'
```

장애 시 대상

실패한 이벤트 소스 매핑 간접 호출 기록을 보관하려면 함수의 이벤트 소스 매핑에 대상을 추가합니다. 대상으로 전송된 각 레코드는 실패한 간접 호출에 대한 메타데이터가 포함된 JSON 문서입니다. 모든 Amazon SNS 주제, Amazon SQS 대기열 또는 S3 버킷을 대상으로 구성할 수 있습니다. 실행 역할에 대상에 대한 권한이 있어야 합니다.

- SQS 대상의 경우: sqs:SendMessage
- SNS 대상의 경우: sns:Publish
- S3 버킷 대상의 경우: s3:PutObject 및 s3:ListBuckets

또한 대상에 KMS 키를 구성한 경우 대상 유형에 따라 Lambda에는 다음과 같은 권한이 필요합니다.

- S3 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:GenerateDataKey가 필요합니다. KMS 키와 S3 버킷 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:GenerateDataKey를 허용합니다.
- SQS 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:Decrypt 및 kms:GenerateDataKey가 필요합니다. KMS 키와 SQS 대기열 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:DescribeKey 및 kms:ReEncrypt를 허용합니다.
- SNS 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:Decrypt 및 kms:GenerateDataKey가 필요합니다. KMS 키와 SNS 주제 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:DescribeKey 및 kms:ReEncrypt를 허용합니다.

이 콘솔을 사용하여 장애 시 대상을 구성하려면 다음 단계를 따르세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 개요(Function overview)에서 대상 추가(Add destination)를 선택합니다.
4. 소스의 경우 이벤트 소스 매핑 간접 호출을 선택합니다.
5. 이벤트 소스 매핑의 경우 이 함수에 대해 구성된 이벤트 소스를 선택합니다.
6. 조건의 경우 실패 시를 선택합니다. 이벤트 소스 매핑 간접 호출의 경우 이 조건만 수락됩니다.
7. 대상 유형의 경우 Lambda가 간접 호출 레코드를 전송할 대상 유형을 선택합니다.
8. Destination(대상)에서 리소스를 선택합니다.
9. 저장을 선택합니다.

AWS CLI를 사용하여 장애 시 대상을 구성할 수도 있습니다. 예를 들어, 다음 create-event-source-mapping 명령은 SQS 장애 시 대상이 있는 이벤트 소스 매핑을 MyFunction에 추가합니다.

```
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"}}'
```

다음 update-event-source-mapping 명령은 uuid 입력과 연결된 이벤트 소스에 S3 장애 시 대상을 추가합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

대상을 제거하려면 destination-config 파라미터의 인수로 빈 문자열을 제공합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 및 SQS 예제 간접 호출 레코드

다음 예는 실패한 Kafka 이벤트 소스 간접 호출에 대해 Lambda가 SNS 주제 또는 SQS 대기열 대상으로 전송하는 내용을 보여줍니다. recordsInfo 아래의 각 키에는 하이픈으로 구분된 Kafka 주제와 파티션이 모두 포함되어 있습니다. 예를 들어, "Topic-0" 키의 경우 Topic은 Kafka 주제이고 0은 파티션입니다. 각 주제 및 파티션에 대해 오프셋과 타임스탬프 데이터를 사용하여 원래의 간접 호출 레코드를 찾을 수 있습니다.

```
{
    "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 대상 예제 간접 호출 레코드

S3 대상의 경우, Lambda는 메타데이터와 함께 전체 간접 호출 레코드를 대상으로 전송합니다. 다음 예는 실패한 Kafka 이벤트 소스 간접 호출에 대해 Lambda가 SNS 주제 또는 S3 버킷 대상으로 전송하는 것을 보여줍니다. SQS 및 SNS 대상에 대한 이전 예제의 모든 필드 외에도 payload 필드에는 원래 간접 호출 레코드가 이스케이프된 JSON 문자열로 포함되어 있습니다.

```
{
    "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**
>
> 대상 버킷에서 S3 버전 관리를 활성화하는 것이 좋습니다.

Amazon MSK 이벤트 소스의 Auto Scaling

Amazon MSK 이벤트 소스를 처음 생성하면 Lambda는 Kafka 주제의 모든 파티션을 처리할 한 명의 소비자를 할당합니다. 각 소비자는 증가한 워크로드를 처리하기 위해 여러 프로세서를 병렬로 실행합니다. 그뿐 아니라 Lambda는 워크로드에 따라 소비자 수를 자동으로 늘리거나 줄입니다. 각 파티션에서 메시지 순서를 유지하기 위해 최대 소비자 수는 주제의 파티션당 하나의 소비자입니다.

Lambda는 1분 간격으로 주제의 모든 파티션의 소비자 오프셋 지연을 평가합니다. 지연이 너무 높으면 파티션은 Lambda가 메시지를 처리할 수 있는 것보다 더 빠르게 메시지를 수신합니다. 필요에 따라 Lambda는 주제에 소비자를 추가하거나 제거합니다. 소비자를 추가 또는 제거하는 크기 조정 프로세스는 평가 후 3분 이내에 진행됩니다.

대상 Lambda 함수가 제한되면 Lambda는 소비자 수를 줄입니다. 이 동작은 소비자가 검색하고 함수에 보낼 수 있는 메시지 수를 줄임으로써 함수의 워크로드를 줄입니다.

Kafka 주제의 처리량을 모니터링하려면 Lambda에서 함수가 레코드를 처리하는 동안 내보내는 오프셋 지연 지표를 확인합니다.

얼마나 많은 함수 호출이 병렬로 발생하는지 확인하기 위해 함수에 대한 동시성 지표를 확인할 수도 있습니다.

폴링 및 스트리밍 시작 위치

이벤트 소스 매핑 생성 및 업데이트 중 스트림 폴링은 최종적으로 일관됩니다.

- 이벤트 소스 매핑 생성 중 스트림에서 이벤트 폴링을 시작하는 데 몇 분 정도 걸릴 수 있습니다.
- 이벤트 소스 매핑 업데이트 중 스트림에서 이벤트 폴링을 중지했다가 다시 시작하는 데 몇 분 정도 걸릴 수 있습니다.

이 동작은 스트림의 시작 위치로 `LATEST`를 지정하면 이벤트 소스 매핑이 생성 또는 업데이트 중에 이벤트를 놓칠 수 있음을 의미합니다. 누락된 이벤트가 없도록 하기 위해서는 스트림 시작 위치를 `TRIM_HORIZON` 또는 `AT_TIMESTAMP`로 지정하세요.

Amazon CloudWatch 지표

Lambda는 함수가 레코드를 처리하는 동안 `OffsetLag` 지표를 내보냅니다. 이 지표의 값은 Kafka 이벤트 소스 주제에 작성된 마지막 레코드와 함수의 소비자 그룹에서 처리한 마지막 레코드 간의 오프셋 차이입니다. 레코드가 추가되는 시점과 소비자 그룹이 이를 처리하는 시점 사이의 지연 시간을 추정하는 데 `OffsetLag`를 사용할 수 있습니다.

`OffsetLag`의 증가 추세는 함수의 소비자 그룹 폴러에 문제가 있음을 나타낼 수 있습니다. 자세한 내용은 Lambda 함수 지표 작업 단원을 참조하십시오.

Amazon MSK 구성 파라미터

모든 Lambda 이벤트 소스 유형은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 일부 파라미터만 Amazon MSK에 적용됩니다.

Amazon MSK에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
AmazonManagedKafkaEventSourceConfig	N	기본적으로 고유한 값으로 설정되는 소비자 그룹 ID 필드를 포함합니다.	생성 시에만 설정할 수 있음
BatchSize	N	100	최대값: 10,000
활성	N	활성	
EventSourceArn	Y		생성 시에만 설정할 수 있음
FunctionName	Y		
FilterCriteria	N		Lambda 이벤트 필터링
MaximumBatchingWindowInSeconds	N	500ms	일괄 처리 동작
SourceAccessConfigurations	N	자격 증명 없음	이벤트 소스에 대한 CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) 또는 SASL/SCRAM
StartingPosition	Y		AT_TIMESTAMP, TRIM_HORIZON, 또는 LATEST 생성 시에만 설정할 수 있음
StartingPositionTimestamp	N		StartingPosition이 AT_TIMESTAMP로 설

파라미터	필수	기본값	참고
			정된 경우에만 필요합니다.
주제	Y		Kafka 주제 이름 생성 시에만 설정할 수 있음

자체 관리형 Apache Kafka에서 Lambda 사용

> **Note**
>
> Lambda 함수 이외의 대상으로 데이터를 전송하거나 데이터를 전송하기 전에 데이터를 보강하려는 경우 Amazon EventBridge 파이프를 참조하세요.

Lambda는 이벤트 소스로 Apache Kafka를 지원합니다. Apache Kafka는 데이터 파이프라인 및 스트리밍 분석과 같은 워크로드를 지원하는 오픈 소스 이벤트 스트리밍 플랫폼입니다.

AWS 관리형 Kafka 서비스인 Amazon Managed Streaming for Apache Kafka(Amazon MSK) 또는 자체 관리형 Kafka 클러스터를 사용할 수 있습니다. 자세한 내용은 Amazon MSK에서 Lambda 사용에 대한 자세한 내용은 Amazon MSK에서 Lambda 사용을 참조하세요.

이 주제에서는 자체 관리형 Kafka 클러스터에서 Lambda를 사용하는 방법을 설명합니다. AWS 용어에서 자체 관리형 클러스터는 AWS가 아닌 다른 서비스에 호스팅된 Kafka 클러스터를 포함합니다. 예를 들어, Confluent Cloud와 같은 클라우드 공급업체를 통해 Kafka 클러스터를 호스팅할 수 있습니다.

이벤트 소스로서 Apache Kafka는 Amazon Simple Queue Service(Amazon SQS) 또는 Amazon Kinesis를 사용하는 것과 유사하게 작동합니다. Lambda는 이벤트 소스의 새 메시지를 내부적으로 폴링한 다음 대상 Lambda 함수를 동기적으로 호출합니다. Lambda는 메시지를 배치 단위로 읽고 이를 함수에 이벤트 페이로드로 제공합니다. 최대 배치 크기는 구성 가능합니다. (기본값은 100개의 메시지입니다.)

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니

다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

Kafka 기반 이벤트 소스의 경우 Lambda는 일괄 처리 기간 및 배치 크기와 같은 처리 제어 파라미터를 지원합니다. 자세한 정보는 일괄 처리 동작 섹션을 참조하세요.

자체 관리형 Kafka를 이벤트 소스로 사용하는 방법의 예는 AWS 컴퓨팅 블로그에서 Using self-hosted Apache Kafka as an event source for AWS Lambda를 참조하세요.

주제

예제 이벤트

Lambda는 Lambda 함수를 호출할 때 이벤트 파라미터의 메시지 배치를 보냅니다. 이벤트 페이로드에는 메시지 배열이 포함됩니다. 각 배열 항목에는 Kafka 주제 및 Kafka 파티션 식별자에 대한 세부 정보와 함께 타임스탬프 및 base64로 인코딩된 메시지가 포함됩니다.

```
{
   "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 클러스터 인증

Lambda는 자체 관리형 Apache Kafka 클러스터를 통해 인증하는 여러 가지 방법을 지원합니다. 지원되는 인증 방법 중 하나를 사용하도록 Kafka 클러스터를 구성해야 합니다. Kafka 보안에 대한 자세한 내용은 Kafka 설명서의 보안 섹션을 참조하세요.

VPC 액세스

VPC 내의 Kafka 사용자만 Kafka 브로커에 액세스하는 경우 Amazon Virtual Private Cloud(Amazon VPC) 액세스에 대해 Kafka 이벤트 소스를 구성해야 합니다.

SASL/SCRAM 인증

Lambda는 전송 계층 보안(TLS) 암호화(SASL_SSL)를 통해 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism(SASL/SCRAM) 인증을 지원합니다. Lambda는 암호화된 자격 증명을 전송하여 클러스터에서 인증합니다. Lambda는 일반 텍스트(SASL_PLAINTEXT)가 포함된 SASL/SCRAM을 지원하지 않습니다. SASL/SCRAM 인증에 관한 자세한 내용은 RFC 5802를 참조하세요.

Lambda는 SASL/PLAIN 인증도 지원합니다. 이 메커니즘은 일반 텍스트 보안 인증을 사용하므로, 서버에 연결할 때 TLS 암호화를 사용하여 보안 인증 정보를 보호해야 합니다.

SASL 인증의 경우 로그인 자격 증명을 AWS Secrets Manager에 보안 암호로 저장합니다. Secrets Manager 사용에 대한 자세한 내용은 AWS Secrets Manager사용 설명서의 자습서: 비밀 정보 생성 및 검색을 참조하세요.

> **Important**
>
> 인증에 Secrets Manager를 사용하려면 Lambda 함수와 동일한 AWS 리전에 보안 암호를 저장해야 합니다.

상호 TLS 인증

상호 TLS(mTLS)는 클라이언트와 서버 간의 양방향 인증을 제공합니다. 클라이언트는 서버가 클라이언트를 확인할 수 있도록 서버에 인증서를 보내고, 서버는 클라이언트가 서버를 확인할 수 있도록 클라이언트에 인증서를 보냅니다.

자체 관리형 Apache Kafka에서 Lambda는 클라이언트 역할을 수행합니다. 클라이언트 인증서(Secrets Manager의 비밀 정보)를 구성하여 Kafka 브로커로 Lambda를 인증합니다. 클라이언트 인증서는 서버의 신뢰 저장소에 있는 CA에서 서명해야 합니다.

Kafka 클러스터는 서버 인증서를 Lambda로 전송하여 Lambda로 Kafka 브로커를 인증합니다. 서버 인증서는 퍼블릭 CA 인증서 또는 프라이빗 CA/자체 서명 인증서일 수 있습니다. 퍼블릭 CA 인증서는 Lambda 신뢰 저장소에 있는 인증 기관(CA)에서 서명해야 합니다. 프라이빗 CA/자체 서명 인증서의 경우 서버 루트 CA 인증서(Secrets Manager의 비밀 정보로)를 구성합니다. Lambda는 루트 인증서를 사용하여 Kafka 브로커를 확인합니다.

mTLS에 대한 자세한 내용은 이벤트 소스로 Amazon MSK에 대한 상호 TLS 인증 도입을 참조하세요.

클라이언트 인증서 비밀 정보 구성

CLIENT_CERTIFICATE_TLS_AUTH 비밀 정보에 인증서 필드와 프라이빗 키 필드가 필요합니다. 암호화된 프라이빗 키의 경우 비밀 정보에 프라이빗 키 암호가 필요합니다. 인증서와 프라이빗 키는 모두 PEM 형식이어야 합니다.

> **Note**
>
> Lambda는 PBES1(PBES2가 아님) 프라이빗 키 암호화 알고리즘을 지원합니다.

인증서 필드에는 클라이언트 인증서부터 시작하여 중간 인증서가 이어지고 루트 인증서로 끝나는 인증서 목록이 포함되어야 합니다. 각 인증서는 다음 구조의 새 줄에서 시작해야 합니다.

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager는 최대 65,536바이트의 보안 정보를 지원하므로 긴 인증서 체인을 위한 충분한 공간입니다.

프라이빗 키는 다음 구조의 PKCS #8 형식이어야 합니다.

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

암호화된 프라이빗 키의 경우 다음 구조를 사용합니다.

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

다음 예제에서는 암호화된 프라이빗 키를 사용한 mTLS 인증용 비밀 정보 콘텐츠를 표시합니다. 암호화된 프라이빗 키의 경우 비밀 정보에 프라이빗 키 암호를 포함합니다.

```
{"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-----"
}
```

서버 루트 CA 인증서 비밀 정보 구성

Kafka 브로커가 프라이빗 CA에서 서명한 인증서로 TLS 암호화를 사용하는 경우 이 비밀 정보를 생성합니다. VPC, SASL/SCRAM, SASL/PLAIN 또는 mTLS 인증에 TLS 암호화를 사용할 수 있습니다.

서버 루트 CA 인증서 비밀 정보에는 PEM 형식의 Kafka 브로커의 루트 CA 인증서가 포함된 필드가 필요합니다. 다음 예제는 비밀 정보의 구조를 보여줍니다.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

API 액세스 및 권한 관리

자체 관리형 Kafka 클러스터에 액세스하는 것 외에도 Lambda 함수에는 다양한 API 작업을 수행할 수 있는 권한이 필요합니다. 함수의 실행 역할에 이러한 권한을 추가합니다. 사용자가 API 작업에 액세스해야 하는 경우 AWS Identity and Access Management(IAM) 사용자 또는 역할에 자격 증명 정책에 필요한 권한을 추가합니다.

필요한 Lambda 함수 권한

Amazon CloudWatch Logs의 로그 그룹에 로그를 생성하고 저장하려면 Lambda 함수의 실행 역할에 다음 권한이 있어야 합니다.

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

선택적 Lambda 함수 권한

Lambda 함수에 또한 다음 권한이 필요할 수 있습니다.

- Secrets Manager 비밀 정보를 설명합니다.
- AWS Key Management Service(AWS KMS) 고객 관리형 키에 액세스합니다.
- Amazon VPC에 액세스합니다.
- 실패한 간접 호출 기록을 대상으로 전송합니다.

Secrets Manager 및 AWS KMS 권한

Kafka 브로커에 대해 구성하는 액세스 제어 유형에 따라 Lambda 함수에는 Secrets Manager 비밀 정보에 액세스하거나 AWS KMS 고객 관리형 키를 복호화할 수 있는 권한이 필요할 수 있습니다. 리소스에 액세스하려면 함수의 실행 역할에 다음 권한이 주어져야 합니다.

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC 권한

VPC 내의 사용자만 자체 관리형 Apache Kafka 클러스터에 액세스할 수 있는 경우 Lambda 함수에 Amazon VPC 리소스에 액세스할 수 있는 권한이 있어야 합니다. 이러한 리소스에는 VPC, 서브넷, 보안 그룹 및 네트워크 인터페이스가 있습니다. 리소스에 액세스하려면 함수의 실행 역할에 다음 권한이 주어져야 합니다.

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs

- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

실행 역할에 권한 추가

자체 관리형 Apache Kafka 클러스터가 사용하는 다른 AWS 서비스에 액세스하기 위해 Lambda는 함수의 실행 역할에 정의된 권한 정책을 사용합니다.

기본적으로 Lambda는 자체 관리형 Apache Kafka 클러스터에 대한 필수 또는 선택적 작업을 수행할 수 없습니다. IAM 신뢰 정책에서 이러한 작업을 생성하고 정의한 다음 정책을 실행 역할에 연결해야 합니다. 이 예제에서는 Lambda가 Amazon VPC 리소스에 액세스하도록 허용하는 정책을 생성하는 방법을 보여줍니다.

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
           }
        ]
     }
```

IAM 콘솔에서 JSON 정책 문서를 생성하는 방법에 대한 자세한 내용은 IAM 사용 설명서의 JSON 탭에서 정책 생성을 참조하세요.

IAM 정책을 사용하여 사용자에게 액세스 권한 부여

기본적으로 사용자 및 역할에는 이벤트 소스 API 작업을 수행할 수 있는 권한이 없습니다. 조직 또는 계정의 사용자에게 액세스 권한을 부여하려면 자격 증명 기반 정책을 생성 혹은 업데이트합니다. 자세한 내용은 IAM 사용 설명서의 정책을 사용하여 AWS 리소스에 대한 액세스 제어를 참조하세요.

인증 및 권한 부여 오류

Kafka 클러스터의 데이터를 사용하는 데 필요한 권한이 누락된 경우 Lambda는 LastProcessingResult 아래의 이벤트 소스 매핑에 다음 오류 메시지 중 하나를 표시합니다.

오류 메시지

- 클러스터가 Lambda를 인증하지 못함
- SASL 인증 실패
- 서버가 Lambda를 인증하지 못함
- Lambda가 서버를 인증하지 못함
- 제공된 인증서 또는 프라이빗 키가 잘못됨

클러스터가 Lambda를 인증하지 못함

SASL/SCRAM 또는 mMTS의 경우 이 오류는 제공된 사용자에게 다음 필수 Kafka 액세스 제어 목록(ACL) 권한이 모두 있지는 않음을 나타냅니다.

- DescribeConfigs 클러스터
- 그룹 설명
- 그룹 읽기
- 주제 설명
- Thread-Topic

필수 `kafka-cluster` 권한으로 Kafka ACL을 생성할 때 주제와 그룹을 리소스로 지정합니다. 주제 이름은 이벤트 소스 매핑의 주제와 일치해야 합니다. 그룹 이름은 이벤트 소스 매핑의 UUID와 일치해야 합니다.

실행 역할에 필요한 권한을 추가한 후 변경 사항이 적용되기까지 몇 분 정도 소요될 수 있습니다.

SASL 인증 실패

SASL/SCRAM 또는 SASL/PLAIN의 경우 이 오류는 제공된 로그인 자격 증명이 유효하지 않음을 나타냅니다.

서버가 Lambda를 인증하지 못함

이 오류는 Kafka 브로커가 Lambda를 인증하지 못했음을 나타냅니다. 이 오류는 다음과 같은 이유로 발생할 수 있습니다.

- mTLS 인증을 위한 클라이언트 인증서를 제공하지 않았습니다.
- 클라이언트 인증서를 제공했지만 Kafka 브로커가 mTLS 인증을 사용하도록 구성되지 않았습니다.
- Kafka 브로커가 클라이언트 인증서를 신뢰하지 않습니다.

Lambda가 서버를 인증하지 못함

이 오류는 Lambda가 Kafka 브로커를 인증하지 못했음을 나타냅니다. 이 오류는 다음과 같은 이유로 발생할 수 있습니다.

- Kafka 브로커는 자체 서명된 인증서 또는 사설 CA를 사용하지만 서버 루트 CA 인증서를 제공하지 않았습니다.
- 서버 루트 CA 인증서가 브로커 인증서에 서명한 루트 CA와 일치하지 않습니다.
- 브로커의 인증서에 브로커의 DNS 이름 또는 IP 주소가 주체 대체 이름으로 포함되어 있지 않기 때문에 호스트 이름 검증에 실패했습니다.

제공된 인증서 또는 프라이빗 키가 잘못됨

이 오류는 Kafka 소비자가 제공된 인증서 또는 프라이빗 키를 사용할 수 없음을 나타냅니다. 인증서와 키가 PEM 형식을 사용하고 프라이빗 키 암호화가 PBES1 알고리즘을 사용하는지 확인합니다.

네트워크 구성

Lambda가 Kafka 클러스터를 이벤트 소스로 사용하려면 클러스터가 있는 Amazon VPC에 액세스해야 합니다. Lambda가 VPC에 액세스할 수 있도록 AWS PrivateLink VPC 엔드포인트를 배포하는 것이 좋습니다. Lambda 및 AWS Security Token Service(AWS STS)에 대한 엔드포인트를 배포합니다. 브로커가 인증을 사용하는 경우 Secrets Manager용 VPC 엔드포인트도 배포합니다. 장애 시 대상을 구성한 경우 대상 서비스를 위한 VPC 엔드포인트도 배포합니다.

또는 Kafka 클러스터와 연결된 VPC에 퍼블릭 서브넷당 하나의 NAT 게이트웨이가 포함되는지 확인합니다. 자세한 내용은 the section called “VPC 함수에 대한 인터넷 액세스” 단원을 참조하십시오.

또한 VPC 엔드포인트를 사용하는 경우 프라이빗 DNS 이름을 활성화하도록 구성해야 합니다.

자체 관리형 Apache Kafka 클러스터에 대한 이벤트 소스 매핑을 생성하는 경우 Lambda는 클러스터 VPC의 서브넷 및 보안 그룹에 대해 탄력적 네트워크 인터페이스(ENI)가 이미 존재하는지 확인합니다. Lambda가 기존 ENI를 찾으면 이를 재사용하려고 시도합니다. 그렇지 않으면 Lambda가 이벤트 소스에 연결하고 함수를 호출하기 위해 새 ENI를 생성합니다.

> **Note**
>
> Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. 이러한 VPC는 서비스에 의해 자동으로 유지 관리되며 고객에게는 표시되지 않습니다. 또한 함수를 Amazon VPC에 연결할 수도 있습니다. 어느 경우든 함수의 VPC 구성은 이벤트 소스 매핑에 영향을 미치지 않습니다. 이벤트 소스의 VPC 구성에 따라 Lambda가 이벤트 소스에 연결되는 방식이 결정됩니다.

네트워크 구성에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Setting up AWS Lambda with an Apache Kafka cluster within a VPC를 참조하세요.

VPC 보안 그룹 규칙

최소한 다음 규칙을 사용하여 클러스터가 포함된 Amazon VPC의 보안 그룹을 구성합니다.

- 인바운드 규칙 – 이벤트 소스에 대해 지정된 보안 그룹에 대해 Kafka 브로커 포트의 모든 트래픽을 허용합니다. Kafka는 기본적으로 포트 9092를 사용합니다.
- 아웃바운드 규칙 - 모든 대상에 대해 포트 443의 모든 트래픽을 허용합니다. 이벤트 소스에 대해 지정된 보안 그룹에 대해 Kafka 브로커 포트의 모든 트래픽을 허용합니다. Kafka는 기본적으로 포트 9092를 사용합니다.
- NAT 게이트웨이 대신 VPC 엔드포인트를 사용하는 경우 VPC 엔드포인트와 연결된 보안 그룹은 이벤트 소스의 보안 그룹에서 포트 443의 모든 인바운드 트래픽을 허용해야 합니다.

VPC 엔드포인트 작업

VPC 엔드포인트를 사용하는 경우 함수를 호출하는 API 직접 호출은 ENI를 사용하여 이러한 엔드포인트를 통해 라우팅됩니다. Lambda 서비스 보안 주체는 해당 ENI를 사용하는 모든 역할과 함수에서 `sts:AssumeRole`과 `lambda:InvokeFunction`을 호출해야 합니다.

기본적으로 VPC 엔드포인트에는 개방적인 IAM 정책이 있습니다. 모범 사례는 특정 보안 주체만 해당 엔드포인트를 사용하여 필요한 작업을 수행할 수 있도록 이러한 정책을 제한하는 것입니다. 이벤트 소스 매핑이 Lambda 함수를 호출할 수 있도록 하려면 VPC 엔드포인트 정책에서 Lambda 서비스 원칙이 `sts:AssumeRole`과 `lambda:InvokeFunction`을 호출할 수 있도록 허용해야 합니다. 조직 내에서 발생하는 API 직접 호출만 허용하도록 VPC 엔드포인트 정책을 제한하면 이벤트 소스 매핑이 제대로 작동하지 않습니다.

다음 예제 VPC 엔드포인트 정책은 AWS STS 및 Lambda 엔드포인트에 대해 Lambda 서비스 보안 주체에 필요한 액세스 권한을 부여하는 방법을 안내합니다.

Example VPC 엔드포인트 정책 - AWS STS 엔드포인트

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 엔드포인트 정책 - Lambda 엔드포인트

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Kafka 브로커 클러스터가 인증을 사용하는 경우 Secrets Manager 엔드포인트에 대한 VPC 엔드포인트 정책을 제한할 수도 있습니다. Secrets Manager API를 호출하기 위해 Lambda는 Lambda 서비스 보안 주체가 아닌 함수 역할을 사용합니다. 다음 예제는 Secrets Manager 엔드포인트 정책을 보여줍니다.

Example VPC 엔드포인트 정책 – Secrets Manager 엔드포인트

```
{
    "Statement": [
```

```
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

장애 발생 시 대상을 구성한 경우 Lambda는 함수의 역할을 사용하여 Lambda 관리형 ENI를 사용하여 `s3:PutObject`, `sns:Publish` 또는 `sqs:sendMessage`를 호출합니다.

Kafka 클러스터를 이벤트 소스로 추가

이벤트 소스 매핑을 생성하려면 Lambda 콘솔, AWS SDK 또는 AWS Command Line Interface(AWS CLI)를 사용해 Kafka를 Lambda 함수 트리거로 추가합니다.

이 단원에서는 Lambda 콘솔 및 AWS CLI를 사용해 이벤트 소스 매핑을 생성하는 방법을 설명합니다.

필수 조건

- 자체 관리형 Apache Kafka 클러스터. Lambda는 Apache Kafka 버전 0.10.1.0 이상을 지원합니다.
- 자체 관리형 Kafka 클러스터에서 사용하는 AWS 리소스에 액세스할 수 있는 권한이 있는 실행 역할입니다.

사용자 지정이 가능한 소비자 그룹 ID

Kafka를 이벤트 소스로 설정할 때 소비자 그룹 ID를 지정할 수 있습니다. 이 소비자 그룹 ID는 Lambda 함수에 가입하려는 Kafka 소비자 그룹의 기존 식별자입니다. 이 기능을 사용하여 진행 중인 모든 Kafka 레코드 처리 설정을 다른 소비자에서 Lambda로 원활하게 마이그레이션할 수 있습니다.

소비자 그룹 ID를 지정하고 해당 소비자 그룹 내에 다른 활성 폴러가 있는 경우 Kafka는 모든 소비자에게 메시지를 배포합니다. 즉, Lambda는 Kafka 주제에 대한 모든 메시지를 수신하지는 않습니다. Lambda가 주제에 있는 모든 메시지를 처리하도록 하려면 해당 소비자 그룹의 다른 모든 폴러를 끄십시오.

또한 소비자 그룹 ID를 지정했는데 Kafka가 동일한 ID를 가진 유효한 기존 소비자 그룹을 찾으면 Lambda는 이벤트 소스 매핑을 위한 `StartingPosition` 파라미터를 무시합니다. 대신 Lambda는 소비자 그룹의 커밋된 오프셋에 따라 레코드 처리를 시작합니다. 소비자 그룹 ID를 지정했는데 Kafka가 기존 소비자 그룹을 찾을 수 없는 경우, Lambda는 `StartingPosition`에서 지정된 대로 이벤트 소스를 구성합니다.

지정하는 소비자 그룹 ID는 모든 Kafka 이벤트 소스 중에서 고유해야 합니다. 지정된 소비자 그룹 ID로 Kafka 이벤트 소스 매핑을 생성한 후에는 이 값을 업데이트할 수 없습니다.

자체 관리형 Kafka 클러스터 추가(콘솔)

다음 단계에 따라 자체 관리형 Apache Kafka 클러스터 및 Kafka 주제를 Lambda 함수의 트리거로 추가합니다.

Lambda 함수에 Apache Kafka 트리거를 추가하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. Lambda 함수의 이름을 선택합니다.
3. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.
4. 트리거 구성에서 다음을 수행합니다.
    a. Apache Kafka 트리거 유형을 선택합니다.
    b. Bootstrap 서버에는 클러스터의 Kafka 브로커 호스트 및 포트 페어 주소를 입력한 다음 추가를 선택합니다. 클러스터의 각 Kafka 브로커에 대해 이를 반복합니다.
    c. 주제 이름에는 클러스터에 레코드를 저장하는 데 사용되는 Kafka 주제의 이름을 입력합니다.
    d. (선택 사항) 배치 크기(Batch size)에 단일 배치에서 검색할 최대 레코드 수를 입력합니다.
    e. Batch window에서 Lambda가 함수를 호출하기 전에 레코드를 수집하는 데 걸리는 최대 시간(초)을 입력합니다.
    f. (선택 사항) Consumer group ID에서 가입할 Kafka 소비자 그룹의 ID를 입력합니다.
    g. (선택 사항) 시작 위치의 경우 최신 레코드에서 스트림 읽기를 시작하려면 최신을 선택하고, 사용 가능한 가장 빠른 레코드에서 시작하려면 수평 트리밍을 선택하고, 읽기를 시작할 타임스탬프를 지정하려면 타임스탬프를 선택합니다.
    h. (선택 사항) VPC에서 Kafka 클러스터용 Amazon VPC를 선택합니다. 그런 다음 VPC 서브넷(VPC subnets)과 VPC 보안 그룹(VPC security groups)을 선택합니다.

        VPC 내의 사용자만 브로커에 액세스하는 경우 이 설정이 필요합니다.

i. (선택 사항) 인증(Authentication)에서 추가(Add)를 선택한 후 다음을 수행합니다.

i. 클러스터에 있는 Kafka 브로커의 액세스 또는 인증 프로토콜을 선택합니다.

- Kafka 브로커가 SASL/PLAIN 인증을 사용하는 경우 BASIC_AUTH를 선택합니다.
- 브로커가 SASL/SCRAM 인증을 사용하는 경우 SASL_SCRAM 프로토콜 중 하나를 선택합니다.
- mTLS 인증을 구성하는 경우 CLIENT_CERTIFICATE_TLS_AUTH 프로토콜을 선택합니다.

ii. SASL/SCRAM 또는 mTLS 인증의 경우 Kafka 클러스터에 대한 자격 증명이 포함된 Secrets Manager 비밀 키를 선택합니다.

j. (선택 사항) 암호화(Encryption)에서 Kafka 브로커가 프라이빗 CA에서 서명한 인증서를 사용하는 경우 Kafka 브로커가 TLS 암호화에 사용하는 루트 CA 인증서가 포함된 Secrets Manager 암호를 선택합니다.

이 설정은 SASL/SCRAM 또는 SASL/PLANE에 대한 TLS 암호화 및 mTLS 인증에 적용됩니다.

k. 테스트를 위해 트리거를 비활성화된 상태에서 생성하려면(권장됨) 트리거 사용을 선택 해제합니다. 또는 트리거를 즉시 사용하려면 트리거 사용을 선택합니다.

5. 트리거를 생성하려면 추가를 선택합니다.

자체 관리형 Kafka 클러스터 추가(AWS CLI)

다음 예제 AWS CLI 명령을 사용하여 Lambda 함수에 대한 자체 관리형 Apache Kafka 트리거를 생성하고 확인합니다.

SASL/SCRAM 사용

Kafka 사용자가 인터넷을 통해 Kafka 브로커에 액세스한다면 SASL/SCRAM 인증용으로 생성한 Secrets Manager 비밀 정보를 지정합니다. 다음 예제에서는 `create-event-source-mapping` AWS CLI 명령을 사용해 `my-kafka-function`이라는 Lambda 함수를 `AWSKafkaTopic`이라는 Kafka 주제에 매핑합니다.

```
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 사용

VPC 내의 Kafka 사용자만 Kafka 브로커에 액세스한다면 VPC, 서브넷 및 VPC 보안 그룹을 지정해야 합니다. 다음 예제에서는 create-event-source-mapping AWS CLI 명령을 사용해 my-kafka-function이라는 Lambda 함수를 AWSKafkaTopic이라는 Kafka 주제에 매핑합니다.

```
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 CLI를 사용하여 상태 확인

다음 예제에서는 get-event-source-mapping AWS CLI 명령을 사용해 생성한 이벤트 소스 매핑의 상태를 설명합니다.

```
aws lambda get-event-source-mapping
          --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

장애 시 대상

실패한 이벤트 소스 매핑 간접 호출 기록을 보관하려면 함수의 이벤트 소스 매핑에 대상을 추가합니다. 대상으로 전송된 각 레코드는 실패한 간접 호출에 대한 메타데이터가 포함된 JSON 문서입니다. 모든 Amazon SNS 주제, Amazon SQS 대기열 또는 S3 버킷을 대상으로 구성할 수 있습니다. 실행 역할에 대상에 대한 권한이 있어야 합니다.

- SQS 대상의 경우: sqs:SendMessage
- SNS 대상의 경우: sns:Publish
- S3 버킷 대상의 경우: s3:PutObject 및 s3:ListBuckets

또한 대상에 KMS 키를 구성한 경우 대상 유형에 따라 Lambda에는 다음과 같은 권한이 필요합니다.

- S3 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:GenerateDataKey가 필요합니다. KMS 키와 S3 버킷 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:GenerateDataKey를 허용합니다.
- SQS 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:Decrypt 및 kms:GenerateDataKey가 필요합니다. KMS 키와 SQS 대기열 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:DescribeKey 및 kms:ReEncrypt를 허용합니다.
- SNS 대상에 대해 자체 KMS 키를 사용하여 암호화를 활성화한 경우 kms:Decrypt 및 kms:GenerateDataKey가 필요합니다. KMS 키와 SNS 주제 대상이 Lambda 함수 및 실행 역할과 다른 계정에 있는 경우 실행 역할을 신뢰하도록 KMS 키를 구성하여 kms:DescribeKey 및 kms:ReEncrypt를 허용합니다.

이 콘솔을 사용하여 장애 시 대상을 구성하려면 다음 단계를 따르세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 개요(Function overview)에서 대상 추가(Add destination)를 선택합니다.
4. 소스의 경우 이벤트 소스 매핑 간접 호출을 선택합니다.
5. 이벤트 소스 매핑의 경우 이 함수에 대해 구성된 이벤트 소스를 선택합니다.
6. 조건의 경우 실패 시를 선택합니다. 이벤트 소스 매핑 간접 호출의 경우 이 조건만 수락됩니다.
7. 대상 유형의 경우 Lambda가 간접 호출 레코드를 전송할 대상 유형을 선택합니다.
8. Destination(대상)에서 리소스를 선택합니다.
9. 저장을 선택합니다.

AWS CLI를 사용하여 장애 시 대상을 구성할 수도 있습니다. 예를 들어, 다음 create-event-source-mapping 명령은 SQS 장애 시 대상이 있는 이벤트 소스 매핑을 `MyFunction`에 추가합니다.

```
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"}}'
```

다음 update-event-source-mapping 명령은 uuid 입력과 연결된 이벤트 소스에 S3 장애 시 대상을 추가합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

대상을 제거하려면 destination-config 파라미터의 인수로 빈 문자열을 제공합니다.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

SNS 및 SQS 예제 간접 호출 레코드

다음 예는 실패한 Kafka 이벤트 소스 간접 호출에 대해 Lambda가 SNS 주제 또는 SQS 대기열 대상으로 전송하는 내용을 보여줍니다. recordsInfo 아래의 각 키에는 하이픈으로 구분된 Kafka 주제와 파티션이 모두 포함되어 있습니다. 예를 들어, "Topic-0" 키의 경우 Topic은 Kafka 주제이고 0은 파티션입니다. 각 주제 및 파티션에 대해 오프셋과 타임스탬프 데이터를 사용하여 원래의 간접 호출 레코드를 찾을 수 있습니다.

```
{
    "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 대상 예제 간접 호출 레코드

S3 대상의 경우, Lambda는 메타데이터와 함께 전체 간접 호출 레코드를 대상으로 전송합니다. 다음 예는 실패한 Kafka 이벤트 소스 간접 호출에 대해 Lambda가 SNS 주제 또는 S3 버킷 대상으로 전송하는 것을 보여줍니다. SQS 및 SNS 대상에 대한 이전 예제의 모든 필드 외에도 payload 필드에는 원래 간접 호출 레코드가 이스케이프된 JSON 문자열로 포함되어 있습니다.

```
{
    "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**
>
> 대상 버킷에서 S3 버전 관리를 활성화하는 것이 좋습니다.

Kafka 클러스터를 이벤트 소스로 사용

Apache Kafka 클러스터를 Lambda 함수의 트리거로 추가하면 클러스터가 이벤트 소스로 사용됩니다.

Lambda는 사용자가 지정한 `StartingPosition`을 기반으로 CreateEventSourceMapping 요청에서 `Topics`로 지정한 Kafka 주제에서 이벤트 데이터를 읽습니다. 성공적인 처리 후, Kafka 토픽은 Kafka 클러스터에 커밋됩니다.

`StartingPosition`을 `LATEST`로 지정하면 Lambda는 주제에 속한 각 파티션의 최신 메시지에서 읽기를 시작합니다. 트리거 구성 후 Lambda가 메시지를 읽기 시작하기까지 약간의 지연이 발생할 수 있으므로 Lambda는 이 기간 중에 생성된 메시지를 읽지 않습니다.

Lambda는 지정한 하나 이상의 Kafka 주제 파티션의 레코드를 처리하고 JSON 페이로드를 함수로 보냅니다. 사용 가능한 레코드가 더 있는 경우 Lambda는 함수가 주제를 따라잡을 때까지 CreateEventSourceMapping 요청에서 지정한 `BatchSize` 값을 기반으로 배치로 레코드를 계속 처리합니다.

함수가 배치의 어떤 메시지에 대해 오류를 반환하면 Lambda는 처리가 성공하거나 메시지가 만료될 때까지 전체 메시지 배치를 다시 시도합니다. 모든 재시도에 실패한 레코드를 실패 시 대상으로 전송하여 나중에 처리하도록 할 수 있습니다.

> **Note**
>
> Lambda 함수의 최대 제한 시간은 일반적으로 15분이지만 Amazon MSK, 자체 관리형 Apache Kafka, Amazon DocumentDB, ActiveMQ 및 RabbitMQ용 Amazon MQ에 대한 이벤트 소스 매핑은 최대 제한 시간이 14분인 함수만 지원합니다. 이 제약 조건에 따라 이벤트 소스 매핑에서 함수 오류 및 재시도를 적절히 처리할 수 있습니다.

폴링 및 스트리밍 시작 위치

이벤트 소스 매핑 생성 및 업데이트 중 스트림 폴링은 최종적으로 일관됩니다.

- 이벤트 소스 매핑 생성 중 스트림에서 이벤트 폴링을 시작하는 데 몇 분 정도 걸릴 수 있습니다.
- 이벤트 소스 매핑 업데이트 중 스트림에서 이벤트 폴링을 중지했다가 다시 시작하는 데 몇 분 정도 걸릴 수 있습니다.

이 동작은 스트림의 시작 위치로 `LATEST`를 지정하면 이벤트 소스 매핑이 생성 또는 업데이트 중에 이벤트를 놓칠 수 있음을 의미합니다. 누락된 이벤트가 없도록 하기 위해서는 스트림 시작 위치를 `TRIM_HORIZON` 또는 `AT_TIMESTAMP`로 지정하세요.

Kafka 이벤트 소스의 Auto Scaling

Apache Kafka 이벤트 소스를 처음 생성하면 Lambda는 한 명의 소비자를 할당하여 Kafka 주제의 모든 파티션을 처리합니다. 각 소비자는 증가한 워크로드를 처리하기 위해 여러 프로세서를 병렬로 실행합니다. 그뿐 아니라 Lambda는 워크로드에 따라 소비자 수를 자동으로 늘리거나 줄입니다. 각 파티션에서 메시지 순서를 유지하기 위해 최대 소비자 수는 주제의 파티션당 하나의 소비자입니다.

Lambda는 1분 간격으로 주제의 모든 파티션의 소비자 오프셋 지연을 평가합니다. 지연이 너무 높으면 파티션은 Lambda가 메시지를 처리할 수 있는 것보다 더 빠르게 메시지를 수신합니다. 필요에 따라

Lambda는 주제에 소비자를 추가하거나 제거합니다. 소비자를 추가 또는 제거하는 크기 조정 프로세스는 평가 후 3분 이내에 진행됩니다.

대상 Lambda 함수가 오버로드되면 Lambda는 소비자 수를 줄입니다. 이 동작은 소비자가 검색하고 함수에 보낼 수 있는 메시지 수를 줄임으로써 함수의 워크로드를 줄입니다.

Kafka 주제의 처리량을 모니터링하려면 `consumer_lag` 및 `consumer_offset` 같은 Apache Kafka 소비자 지표를 확인합니다. 얼마나 많은 함수 호출이 병렬로 발생하는지 확인하기 위해 함수에 대한 동시성 지표를 확인할 수도 있습니다.

이벤트 소스 매핑 오류

Lambda 함수의 이벤트 소스로 Apache Kafka 클러스터를 추가한 경우 함수에 오류가 발생하면 Kafka 소비자가 레코드 처리를 중지합니다. 토픽 파티션의 소비자는 레코드를 구독하고, 읽고, 처리하는 소비자입니다. 다른 Kafka 소비자는 동일한 오류가 발생하지 않는 한 레코드 처리를 계속할 수 있습니다.

중지된 소비자의 원인을 확인하려면 `StateTransitionReason` 응답의 `EventSourceMapping` 필드를 확인하세요. 다음 목록에는 발생할 수 있는 이벤트 소스 오류가 나왔습니다.

`ESM_CONFIG_NOT_VALID`

이벤트 소스 매핑 구성이 잘못되었습니다.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda가 이벤트 소스를 인증하지 못했습니다.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda에 이벤트 소스에 액세스하는 데 필요한 권한이 없습니다.

`FUNCTION_CONFIG_NOT_VALID`

함수의 구성이 유효하지 않습니다.

> **Note**
>
> Lambda 이벤트 레코드가 허용되는 크기 제한인 6MB를 초과하면 처리되지 않을 수 있습니다.

Amazon CloudWatch 지표

Lambda는 함수가 레코드를 처리하는 동안 `OffsetLag` 지표를 내보냅니다. 이 지표의 값은 Kafka 이벤트 소스 주제에 작성된 마지막 레코드와 함수의 소비자 그룹에서 처리한 마지막 레코드 간의 오프셋 차이입니다. 레코드가 추가되는 시점과 소비자 그룹이 이를 처리하는 시점 사이의 지연 시간을 추정하는 데 `OffsetLag`를 사용할 수 있습니다.

`OffsetLag`의 증가 추세는 함수의 소비자 그룹 폴러에 문제가 있음을 나타낼 수 있습니다. 자세한 내용은 Lambda 함수 지표 작업 단원을 참조하십시오.

자체 관리형 Apache Kafka 구성 파라미터

모든 Lambda 이벤트 소스 유형은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 파라미터 중 일부는 Apache Kafka에 적용됩니다.

자체 관리형 Apache Kafka에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
BatchSize	N	100	최대값: 10,000
활성	N	활성	
FunctionName	Y		
FilterCriteria	N		Lambda 이벤트 필터링
MaximumBatchingWindowInSeconds	N	500ms	일괄 처리 동작
SelfManagedEventSource	Y		Kafka 브로커의 목록. 생성 시에만 설정할 수 있음
SelfManagedKafkaEventSourceConfig	N	기본적으로 고유한 값으로 설정되는 소비자 그룹 ID 필드를 포함합니다.	생성 시에만 설정할 수 있음

파라미터	필수	기본값	참고
SourceAccessConfigurations	N	자격 증명 없음	클러스터에 대한 VPC 정보 또는 인증 자격 증명 SASL_PLAIN의 경우 BASIC_AUTH로 설정
StartingPosition	Y		AT_TIMESTAMP, TRIM_HORIZON, 또는 LATEST 생성 시에만 설정할 수 있음
StartingPositionTimestamp	N		StartingPosition이 AT_TIMESTAMP로 설정된 경우에만 필요합니다.
주제	Y		주제 이름 생성 시에만 설정할 수 있음

Amazon SQS에서 Lambda 사용

> **Note**
>
> Lambda 함수 이외의 대상으로 데이터를 전송하거나 데이터를 전송하기 전에 데이터를 보강하려는 경우 Amazon EventBridge 파이프를 참조하세요.

Lambda 함수를 사용하여 Amazon Simple Queue Service(Amazon SQS) 대기열의 메시지를 처리할 수 있습니다. Lambda는 이벤트 소스 매핑을 위해 표준 대기열과 선입선출(FIFO) 대기열을 모두 지원합니다.

Amazon SQS 이벤트 소스 매핑에 대한 폴링 및 배치 동작 이해

Amazon SQS 이벤트 소스 매핑을 사용하면 Lambda가 대기열을 폴링하고 이벤트와 동기적으로 함수를 간접적으로 호출합니다. 각 이벤트에는 대기열의 여러 메시지 배치가 포함될 수 있습니다. Lambda는 이러한 이벤트를 한 번에 한 배치씩 수신하고 각 배치에 대해 함수를 한 번씩 간접적으로 호출합니다. 함수가 배치를 성공적으로 처리하면 Lambda는 대기열에서 메시지를 삭제합니다.

Lambda가 배치를 수신하면 메시지는 대기열에 머무르지만 대기열의 표시 제한 시간 동안 숨겨집니다. 함수가 배치의 모든 메시지를 성공적으로 처리하면 Lambda는 대기열에서 메시지를 삭제합니다. 기본적으로 배치를 처리하는 동안 함수에 오류가 발생하면 표시 제한 시간이 만료된 후 해당 배치의 모든 메시지가 대기열에 다시 표시됩니다. 이러한 이유로 함수 코드는 의도하지 않은 부작용 없이 동일한 메시지를 여러 번 처리할 수 있어야 합니다.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

Lambda가 메시지를 여러 번 처리하지 못하도록 하려면 함수 응답에 배치 항목 실패를 포함하도록 이벤트 소스 매핑을 구성하거나, Lambda 함수가 메시지를 성공적으로 처리할 경우 DeleteMessage API를 사용하여 대기열에서 메시지를 제거할 수 있습니다.

SQS 이벤트 소스 매핑을 위해 Lambda가 지원하는 구성 파라미터에 대한 자세한 내용은 the section called “SQS 이벤트 소스 매핑 생성” 섹션을 참조하세요.

표준 대기열 메시지 이벤트의 예

Example Amazon SQS 메시지 이벤트(표준 대기열)

```
{
    "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"
        }
    ]
}
```

기본적으로 Lambda는 대기열에서 최대 10개의 메시지를 한 번에 폴링하고 해당 배치를 함수로 보냅니다. 소수의 레코드로 함수를 간접적으로 호출하는 것을 피하려면 배치 기간을 구성하여 이벤트 소스가 최대 5분 동안 레코드를 버퍼링하도록 구성할 수 있습니다. 함수를 간접적으로 호출하기 전에 Lambda는 배치 기간이 만료되거나, 간접 호출 페이로드 크기 할당량에 도달하거나, 구성된 최대 배치 크기에 도달할 때까지 표준 대기열에서 메시지를 계속 폴링합니다.

배치 기간을 사용하고 SQS 대기열에 트래픽이 매우 적은 경우, Lambda는 함수를 호출하기 전에 최대 20초까지 기다릴 수 있습니다. 이는 배치 기간을 20초 미만으로 설정한 경우에도 마찬가지입니다.

> **Note**
>
> Java에서는 JSON을 역직렬화할 때 null 포인터 오류가 발생할 수 있습니다. 이는 JSON 객체 매퍼가 'Records' 및 "eventSourceARN"의 대소문자를 변환하는 방식 때문일 수 있습니다.

FIFO 대기열 메시지 이벤트의 예

FIFO 대기열의 경우 레코드에는 중복 제거 및 시퀀싱과 관련된 추가 속성이 포함되어 있습니다.

Example Amazon SQS 메시지 이벤트(FIFO 대기열)

```
{
    "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 이벤트 소스 매핑 생성 및 구성

Lambda로 Amazon SQS 메시지를 처리하려면 적절한 설정으로 대기열을 구성한 다음 Lambda 이벤트 소스 매핑을 생성하세요.

Lambda에서 사용할 수 있도록 대기열 구성

기존 Amazon SQS 대기열이 없으면 Lambda 함수의 이벤트 소스로 사용할 대기열을 생성합니다. 그런 다음 Lambda 함수가 각 이벤트 배치를 처리하는 데 충분한 시간을 허용하도록 대기열을 구성합니다.

함수 시간이 각 레코드 배치를 처리할 수 있도록 하려면 소스 대기열의 표시 제한 시간을 함수에 구성 제한 시간의 6배 이상으로 설정하세요. 이전 배치를 처리하는 동안 함수가 제한되는 경우 추가 시간을 통해 Lambda가 재시도할 수 있습니다.

기본적으로 배치를 처리하는 동안 Lambda에 오류가 발생하면 해당 배치의 모든 메시지가 대기열로 돌아갑니다. 표시 제한 시간 후 메시지가 Lambda에 다시 표시됩니다. 부분 배치 응답을 사용하여 실패한 메시지만 대기열에 반환하도록 이벤트 소스 매핑을 구성할 수 있습니다. 또한 함수가 메시지를 여러 번 처리하지 못하면 Amazon SQS에서 메시지를 DLQ(Dead Letter Queue)로 보낼 수 있습니다. 소스 대기열의 리드라이브 정책에서 `maxReceiveCount`를 5 이상으로 설정하는 것이 좋습니다. 이를 통해 Lambda는 실패한 메시지를 DLQ(Dead Letter Queue)로 직접 전송하기 전에 몇 번의 재시도를 할 수 있습니다.

Lambda 실행 역할 권한 설정

AWSLambdaSQSQueueExecutionRole AWS 관리형 정책에는 Lambda가 Amazon SQS 대기열에서 읽는 데 필요한 권한이 포함되어 있습니다. 함수의 실행 역할에 이 관리형 정책을 추가할 수 있습니다.

선택적으로 암호화된 대기열을 사용하는 경우 실행 역할에 다음 권한도 추가해야 합니다.

- kms:Decrypt

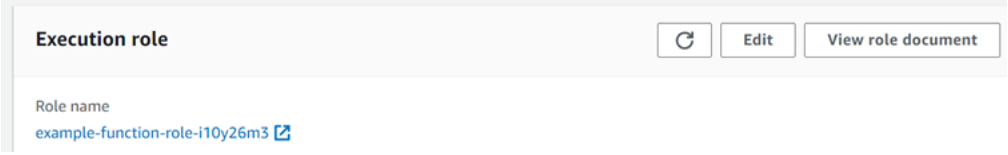
SQS 이벤트 소스 매핑 생성

이벤트 소스 매핑을 생성하여 Lambda가 대기열의 항목을 Lambda 함수로 전송하게 할 수 있습니다. 여러 이벤트 소스 매핑을 생성하여 하나의 함수로 여러 대기열의 항목을 처리할 수 있습니다. Lambda가 대상 함수를 호출하면 이벤트에는 구성 가능한 최대 배치 크기까지 항목이 여러 개 포함될 수 있습니다.

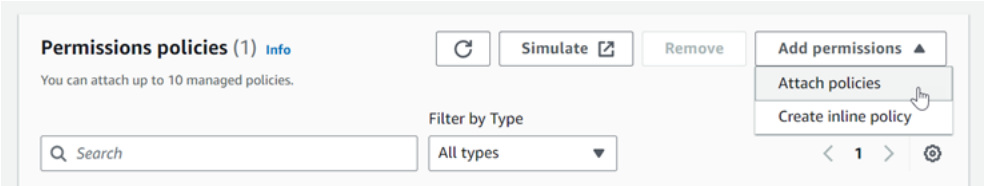
Amazon SQS에서 읽도록 함수를 구성하려면 AWSLambdaSQSQueueExecutionRole AWS 관리형 정책을 실행 역할에 연결합니다. 그리고 다음 단계를 사용하여 콘솔에서 SQS 이벤트 소스 매핑을 생성합니다.

권한 추가 및 트리거 생성

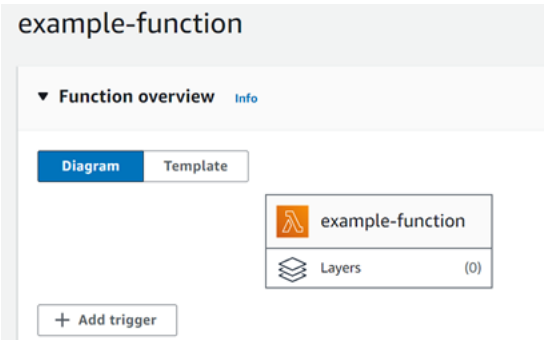
1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 권한(Permissions)을 선택합니다.
4. 역할 이름에서 실행 역할에 대한 링크를 선택합니다. 이 링크를 클릭하면 IAM 콘솔에서 역할이 열립니다.



5. 권한 추가를 선택하고 정책 연결을 선택합니다.



6. 검색 필드에 `AWSLambdaSQSQueueExecutionRole`를 입력합니다. 실행 역할에 이 정책을 추가합니다. 이는 함수가 Amazon SQS 대기열에서 읽어야 하는 권한을 포함하는 AWS 관리형 정책입니다. 이 정책에 대한 자세한 내용은 AWS Managed Policy 참조 안내서의 AWSLambdaSQSQueueExecutionRole을 참조하세요.
7. Lambda 콘솔에서 함수로 돌아갑니다. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.



8. 트리거 유형을 선택합니다.
9. 필요한 옵션을 구성한 다음 추가를 선택합니다.

Lambda는 Amazon SQS 이벤트 소스에 대해 다음과 같은 구성 옵션을 지원합니다.

SQS 대기열

레코드를 읽어 올 Amazon SQS 대기열입니다.

트리거 활성화

이벤트 소스 매핑의 상태입니다. Enable trigger(트리거 활성화)는 기본적으로 선택됩니다.

배치 크기

각 배치에서 함수에 보낼 레코드 최대 수입니다. 표준 대기열의 경우 최대 10,000개의 레코드가 될 수 있습니다. FIFO 대기열의 경우 최대값은 10입니다. 10을 초과하는 배치 크기의 경우 배치 기간(`MaximumBatchingWindowInSeconds`)도 최소 1초로 설정해야 합니다.

함수 제한 시간은 항목의 전체 배치를 처리할 시간이 충분하도록 구성합니다. 항목을 처리하는 데 걸리는 시간이 길면 더 작은 배치 크기를 선택합니다. 배치 크기가 크면 매우 빠르거나 오버헤드가 큰 워크로드에 대한 효율성에 영향을 미칠 수 있습니다. 함수에 예약된 동시성을 구성할 경우 최소 동시성 실행 수를 5로 설정하여 Lambda가 함수를 호출할 때 오류를 제한할 수 있는 가능성을 줄이세요.

이벤트의 총 크기가 동기식 호출에 대한 간접 호출 페이로드 크기 할당량(6MB)을 초과하지 않는 한 Lambda는 단일 직접 호출로 배치의 모든 레코드를 함수에 전달합니다. Lambda와 Amazon SQS는 모두 각 레코드의 메타데이터를 생성합니다. 이 추가 메타데이터는 총 페이로드 크기에 포함되며, 그러면 배치로 전송된 총 레코드 수가 구성된 배치 크기보다 작을 수 있습니다. Amazon SQS에서 전송하는 메타데이터 필드는 길이가 가변적일 수 있습니다. Amazon SQS 메타데이터 필드에 대한 자세한 내용은 Amazon Simple Queue Service API 참조의 ReceiveMessage API 작업 설명서를 참조하세요.

배치 기간

함수를 호출하기 전에 기록을 수집할 최대 기간(단위: 초)입니다. 이 지표는 표준 대기열에만 적용됩니다.

0초보다 큰 배치 기간을 사용하는 경우 대기열의 표시 제한 시간에서 늘어난 처리 시간을 고려해야 합니다. 대기열의 가시성 제한 시간을 함수 제한 시간의 6배에 `MaximumBatchingWindowInSeconds` 값을 더한 값으로 설정하는 것이 좋습니다. 이렇게 하면 Lambda 함수가 각 이벤트 배치를 처리하고 제한 오류가 발생할 경우 다시 시도할 수 있습니다.

메시지를 사용할 수 있게 되면 Lambda는 메시지를 일괄 처리하기 시작합니다. Lambda는 함수를 동시에 5번 호출하여 한 번에 5개의 배치를 처리하기 시작합니다. 메시지를 계속 사용할 수 있는 경우 Lambda는 분당 최대 300개의 함수 인스턴스를 추가해 최대 1,000개까지 추가합니다. 함수 확장 및 동시성에 대한 자세한 내용은 Lambda 함수 크기 조정을 참조하세요.

더 많은 메시지를 처리하기 위해 Lambda 함수를 최적화하여 처리량을 높일 수 있습니다. 자세한 내용은 AWS Lambda의 Amazon SQS 표준 대기열 크기 조정 방식 이해를 참조하세요.

최대 동시성

이벤트 소스가 호출할 수 있는 최대 동시성 함수 수입니다. 자세한 내용은 Amazon SQS 이벤트 소스의 최대 동시성 구성 단원을 참조하십시오.

필터 기준

필터 기준을 추가하여 Lambda가 처리를 위해 함수로 보내는 이벤트를 제어합니다. 자세한 내용은 Lambda 이벤트 필터링 단원을 참조하십시오.

SQS 이벤트 소스 매핑에 대한 규모 조정 동작 구성

표준 대개열의 경우 Lambda는 긴 폴링을 사용하여 대기열이 활성화될 때까지 대기열을 폴링합니다. 메시지를 사용할 수 있는 경우 Lambda는 함수를 동시에 5번 호출하여 한 번에 5개의 배치를 처리하기 시작합니다. 메시지를 계속 사용할 수 있는 경우 Lambda는 배치를 읽는 프로세스의 수를 분당 최대 300개의 추가 인스턴스까지 증가시킵니다. 이벤트 소스 매핑으로 동시에 처리할 수 있는 최대 배치 수는 1,000개입니다.

FIFO 대기열의 경우, Lambda는 메시지를 수신하는 순서대로 함수에 메시지를 보냅니다. FIFO 대기열에 메시지를 전송할 때 메시지 그룹 ID를 지정합니다. Amazon SQS는 동일한 그룹의 메시지가 순서대로 Lambda에 전송되도록 합니다. Lambda가 메시지를 배치로 읽을 때 각 배치에는 둘 이상의 메시지 그룹의 메시지가 포함될 수 있지만 메시지 순서는 그대로 유지됩니다. 함수가 오류를 반환하면 함수는 Lambda가 동일한 그룹에서 추가 메시지를 수신하기 전에 영향을 받는 메시지에 대해 모든 재시도를 시도합니다.

Amazon SQS 이벤트 소스의 최대 동시성 구성

최대 동시성 설정을 사용하여 SQS 이벤트 소스의 규모 조정 동작을 제어할 수 있습니다. 최대 동시성 설정은 Amazon SQS 이벤트 소스가 호출할 수 있는 함수의 동시 인스턴스 수를 제한합니다. 최대 동시성은 이벤트 소스 수준 설정입니다. 여러 Amazon SQS 이벤트 소스가 하나의 함수에 매핑되어 있는 경우, 각 이벤트 소스마다 별도의 최대 동시성 설정이 있을 수 있습니다. 최대 동시성을 사용하여 단일 대기열이 함수의 예약된 동시성 전체를 사용하거나 계정의 나머지 동시성 할당량을 사용하지 못하도록 할 수 있습니다. Amazon SQS 이벤트 소스에 대해 최대 동시성을 구성하는 데는 요금이 부과되지 않습니다.

중요한 것은 최대 동시성과 예약된 동시성은 독립된 두 설정이라는 것입니다. 함수의 예약된 동시성보다 큰 최대 동시성을 설정하지 마세요. 최대 동시성을 구성한 경우, 함수의 예약된 동시성이 함수의 모든 Amazon SQS 이벤트 소스에 대한 총 최대 동시성보다 크거나 같은지 확인합니다. 그렇지 않으면 Lambda가 메시지를 제한할 수도 있습니다.

최대 동시성이 설정되지 않은 경우 Lambda는 Amazon SQS 이벤트 소스를 계정의 총 동시성 할당량(기본값 1,000)까지 확장할 수 있습니다.

> **Note**
>
> FIFO 대기열의 경우 동시 호출은 메시지 그룹 ID(`messageGroupId`) 또는 최대 동시성 설정 중 더 낮은 값으로 제한됩니다. 예를 들어, 메시지 그룹 ID가 6개이고 최대 동시성이 10으로 설정된 경우 함수는 최대 6개의 동시 호출을 가질 수 있습니다.

신규 및 기존 Amazon SQS 이벤트 소스 매핑에 대해 최대 동시성을 구성할 수 있습니다.

Lambda 콘솔을 사용하여 최대 동시성 구성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. Function overview(함수 개요)에서 SQS를 선택합니다. 그러면 Configuration(구성) 탭이 열립니다.
4. Amazon SQS 트리거를 선택하고 Edit(편집)를 선택합니다.
5. Maximum concurrency(최대 동시성)에 2에서 1,000 사이의 숫자를 입력합니다. 최대 동시성을 해제하려면 상자를 비워 둡니다.
6. Save(저장)를 선택합니다.

AWS Command Line Interface(AWS CLI)를 사용하여 최대 동시성 구성

`--scaling-config` 옵션과 함께 update-event-source-mapping 명령을 사용합니다. 예제

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

최대 동시성을 해제하려면 `--scaling-config`에 빈 값을 입력합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Lambda API를 사용하여 최대 동시성 구성

ScalingConfig 객체와 함께 CreateEventSourceMapping 또는 UpdateEventSourceMapping 작업을 사용합니다.

Lambda에서 SQS 이벤트 소스에 대한 오류 처리

SQS 이벤트 소스와 관련된 오류를 처리하기 위해 Lambda는 백오프 전략과 함께 재시도 전략을 자동으로 사용합니다. 부분 배치 응답을 반환하도록 SQS 이벤트 소스 매핑을 구성하여 오류 처리 동작을 사용자 지정할 수도 있습니다.

실패한 호출에 대한 백오프 전략

호출이 실패하면 Lambda는 백오프 전략을 구현하면서 호출을 재시도합니다. 백오프 전략은 Lambda가 함수 코드의 오류로 인해 장애를 겪었는지 아니면 제한으로 인해 장애가 발생했는지에 따라 약간 다릅니다.

- 함수 코드로 인해 오류가 발생한 경우 Lambda는 간접 호출 처리 및 재시도를 중지합니다. 그 동안 Lambda는 점진적으로 백오프를 수행하여 Amazon SQS 이벤트 소스 매핑에 할당되는 동시성의 양을 줄입니다. 대기열의 표시 제한 시간이 다 지나면 메시지가 대기열에 다시 나타납니다.
- 제한으로 인해 호출이 실패하는 경우 Lambda는 Amazon SQS 이벤트 소스 매핑에 할당되는 동시성의 양을 줄여 재시도를 점진적으로 백오프합니다. Lambda는 메시지의 타임스탬프가 대기열의 가시성 제한 시간을 초과할 때(이때 Lambda가 메시지를 삭제)까지 메시지를 계속 재시도합니다.

부분 일괄 응답 구현

배치를 처리하는 동안 Lambda 함수에 오류가 발생하면 Lambda가 성공적으로 처리한 메시지를 포함하여 해당 배치의 모든 메시지가 기본적으로 대기열에 다시 표시됩니다. 따라서 함수가 동일한 메시지를 여러 번 처리할 수 있습니다.

실패한 배치에서 정상 처리된 메시지를 재처리하지 않으려면 실패한 메시지만 다시 표시하도록 이벤트 소스 매핑을 구성할 수 있습니다. 이를 부분 일괄 응답이라고 합니다. 부분 일괄 응답을 켜려면 이벤트 소스 매핑을 구성할 때 FunctionResponseTypes 작업에 대해 `ReportBatchItemFailures`을 지정하십시오. 그러면 함수가 부분적인 성공을 반환할 수 있으므로 레코드에 대한 불필요한 재시도 횟수를 줄일 수 있습니다.

`ReportBatchItemFailures`이 활성화되면 Lambda는 함수 호출이 실패할 때 메시지 폴링을 축소하지 않습니다. 일부 메시지에 오류가 발생할 것으로 예상되고 이러한 오류가 메시지 처리 속도에 영향을 미치지 않도록 하려면 `ReportBatchItemFailures`를 사용하십시오.

> **Note**
>
> 부분 일괄 응답을 사용할 때는 다음 사항에 유의하세요.
>
> - 함수에서 예외가 발생한 경우 전체 배치가 완전한 실패로 간주됩니다.
> - FIFO 대기열과 함께 이 기능을 사용하는 경우 함수는 첫 번째 실패 후 메시지 처리를 중지하고 `batchItemFailures`에서 모든 실패한 메시지와 처리되지 않은 메시지를 반환해야 합니다. 그러면 대기열의 메시지 순서를 유지할 수 있습니다.

부분 배치 보고를 활성화 방법

1. 부분 일괄 응답 구현에 대한 모범 사례를 검토하세요.
2. 다음 명령을 실행하여 함수의 ReportBatchItemFailures을 활성화합니다. 이벤트 소스 매핑의 UUID를 가져오려면 list-event-source-mappings AWS CLI 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. 함수 코드를 업데이트하여 모든 예외를 포착하고 실패한 메시지를 batchItemFailures JSON 응답으로 반환하세요. batchItemFailures 응답에는 메시지 ID 목록이 itemIdentifier JSON 값으로 포함되어야 합니다.

   예를 들어 메시지 ID가 id1, id2, id3, id4, id5인 5개의 메시지로 구성된 배치가 있다고 가정합니다. 함수가 id1, id3, id5를 성공적으로 처리합니다. id2 및 id4 메시지를 대기열에서 다시 볼 수 있도록 하려면 함수가 다음 응답을 리턴해야 합니다.

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

   다음은 일괄적으로 실패한 메시지 ID 목록을 반환하는 함수 코드의 몇 가지 예입니다.

   .NET

   AWS SDK for .NET

   > **Note**
   >
   > GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda에서 SQS 배치 항목 실패를 보고합니다.

```
// 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}`);
}
```

TypeScript를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
// 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
}
```

실패한 이벤트가 대기열로 반환되지 않는 경우 AWS 지식 센터에서 Lambda 함수 SQS ReportBatchItemFailure의 문제를 해결하려면 어떻게 해야 합니까?를 참조하십시오.

성공 및 실패 조건

Lambda는 함수가 다음 중 하나를 반환할 경우 배치를 완전한 성공으로 처리합니다.

- 비어 있는 `batchItemFailures` 목록
- null `batchItemFailures` 목록
- 비어 있는 `EventResponse`
- null `EventResponse`

Lambda는 함수가 다음 중 하나를 반환할 경우 배치를 완전한 실패로 처리합니다.

- 잘못된 JSON 응답
- 빈 문자열 `itemIdentifier`
- null `itemIdentifier`
- 키 이름이 잘못된 `itemIdentifier`
- 존재하지 않는 메시지 ID가 있는 `itemIdentifier` 값

CloudWatch 지표

함수가 배치 항목 실패를 올바르게 보고하는지 확인하려면 Amazon CloudWatch에서 `NumberOfMessagesDeleted` 및 `ApproximateAgeOfOldestMessage` Amazon SQS 지표를 모니터링하면 됩니다.

- `NumberOfMessagesDeleted`는 대기열에서 제거된 메시지 수를 추적합니다. 이 값이 0으로 떨어지면 함수 응답이 실패한 메시지를 올바르게 반환하지 않는다는 신호입니다.
- `ApproximateAgeOfOldestMessage`는 가장 오래된 메시지가 대기열에 머물렀던 시간을 추적합니다. 이 지표가 급격히 증가하면 함수가 실패한 메시지를 올바르게 반환하지 않음을 나타낼 수 있습니다.

Amazon SQS 이벤트 소스 매핑을 위한 Lambda 파라미터

모든 Lambda 이벤트 소스 유형은 동일한 CreateEventSourceMapping 및 UpdateEventSourceMapping API 작업을 공유합니다. 그러나 일부 파라미터만 Amazon SQS에 적용됩니다.

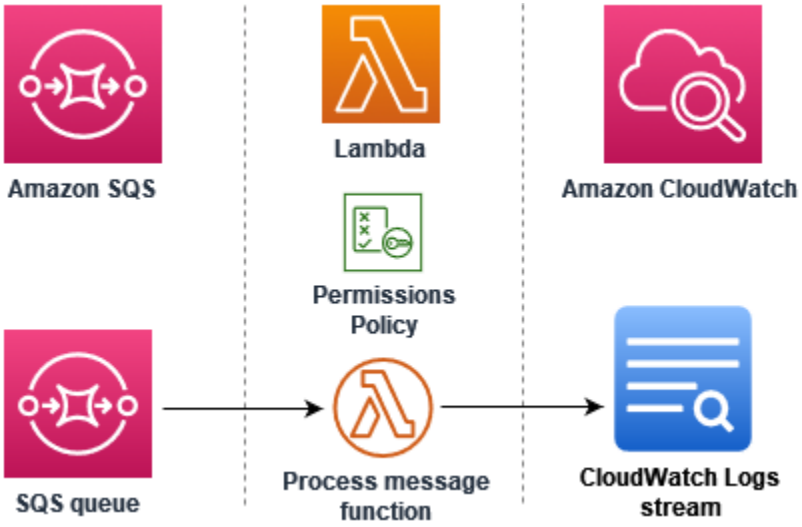
Amazon SQS에 적용되는 이벤트 소스 파라미터

파라미터	필수	기본값	참고
BatchSize	N	10	표준 대기열의 경우 최댓값은 10,000입니다. FIFO 대기열의 경우 최댓값은 10입니다.
활성	N	true	
EventSourceArn	Y		데이터 스트림 또는 스트림 소비자의 ARN
FunctionName	Y		
FilterCriteria	N		Lambda 이벤트 필터링
FunctionResponseTypes	N		함수가 배치에서 특정 실패를 보고하도록 하려면 `FunctionResponseTypes` 에 `ReportBatchItemFailures` 값을 포함하세요. 자세한 내용은 부분 일괄

파라미터	필수	기본값	참고
			응답 구현 단원을 참조하십시오.
MaximumBatchingWindowInSeconds	N	0	
ScalingConfig	N		Amazon SQS 이벤트 소스의 최대 동시성 구성

자습서: Amazon SQS에서 Lambda 사용

이 자습서에서는 Amazon Simple Queue Service(Amazon SQS) 대기열에서 메시지를 사용하는 Lambda 함수를 생성합니다. Lambda 함수는 새 메시지가 대기열에 추가될 때마다 실행됩니다. 이 함수는 메시지를 Amazon CloudWatch Logs 스트림에 기록합니다. 다음 다이어그램은 자습서를 완료하는 데 사용하는 AWS 리소스를 보여줍니다.



이 자습서를 완료하려면 다음 단계를 수행하세요.

1. CloudWatch Logs에 메시지를 작성하는 Lambda 함수를 생성합니다.
2. Amazon SQS 대기열을 생성합니다.
3. Lambda 이벤트 소스 매핑을 생성합니다. 이벤트 소스 매핑은 Amazon SQS 대기열을 읽고 새 메시지가 추가되면 Lambda 함수를 간접적으로 호출합니다.
4. 대기열에 메시지를 추가하고 CloudWatch Logs에서 결과를 모니터링하여 설정을 테스트합니다.

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

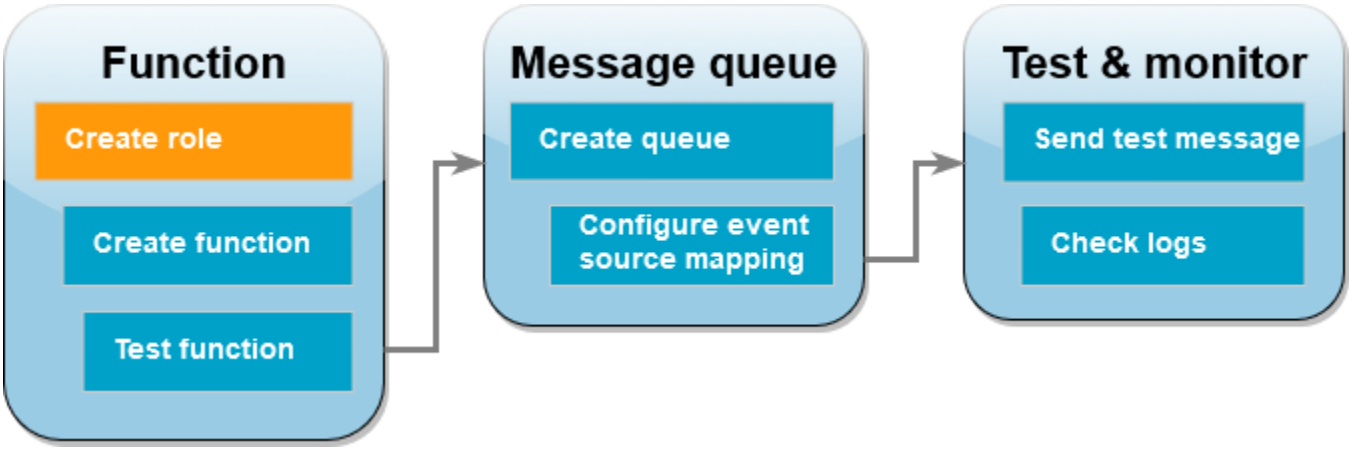
AWS Command Line Interface 설치

아직 AWS Command Line Interface를 설치하지 않은 경우 AWS CLI의 최신 버전 설치 또는 업데이트에서 설명하는 단계에 따라 설치하세요.

이 자습서에서는 명령을 실행할 셸 또는 명령줄 터미널이 필요합니다. Linux 및 macOS에서는 선호하는 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다.

실행 역할 생성



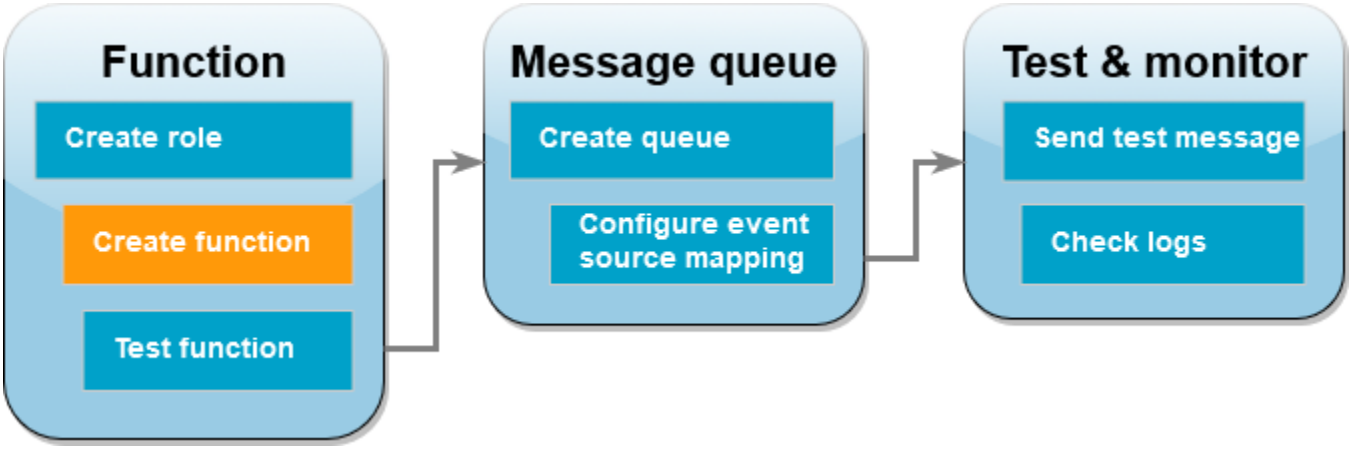
실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 AWS Identity and Access Management(IAM) 역할입니다. 함수가 Amazon SQS에서 항목을 읽을 수 있도록 허용하려면 AWSLambdaSQSQueueExecutionRole 권한 정책을 연결합니다.

실행 역할을 생성하고 Amazon SQS 권한 정책을 연결하려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형에서 AWS 서비스를 선택합니다.
4. 사용 사례에서 Lambda를 선택합니다.
5. 다음을 선택합니다.
6. 권한 정책 검색 상자에 **`AWSLambdaSQSQueueExecutionRole`**을 입력합니다.
7. AWSLambdaSQSQueueExecutionRole 정책을 선택한 후 다음을 선택합니다.
8. 역할 세부 정보에서 역할 이름에 **`lambda-sqs-role`**을 입력한 다음 역할 생성을 선택합니다.

역할을 생성한 후에는 실행 역할의 Amazon 리소스 이름(ARN)을 기록해 둡니다. 이는 이후 단계에서 필요합니다.

함수 생성



Amazon SQS 메시지를 처리하는 Lambda 함수를 생성합니다. 함수 코드는 Amazon SQS 메시지 본문을 CloudWatch Logs에 로그합니다.

이 자습서에서는 Node.js 18.x 런타임을 사용하지만 다른 런타임 언어의 예제 코드도 제공했습니다. 다음 상자에서 탭을 선택하여 관심 있는 런타임에 대한 코드를 볼 수 있습니다. 이 단계에서 사용할 JavaScript 코드는 JavaScript 탭에 표시된 첫 번째 예제입니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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;
  }
}
```

TypeScript를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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 함수를 생성하려면

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir sqs-tutorial
cd sqs-tutorial
```

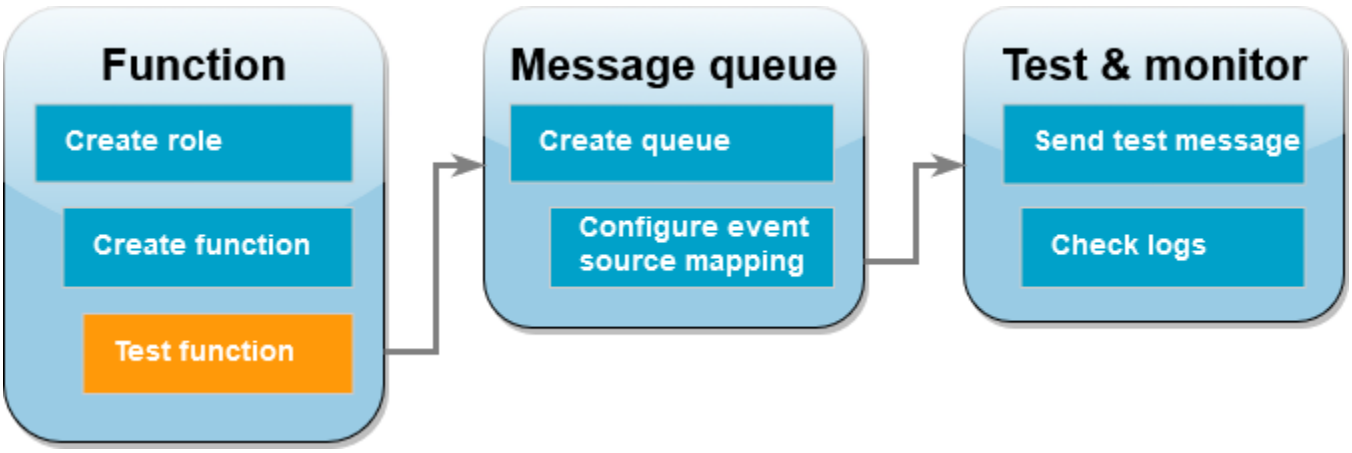
2. 샘플 JavaScript 코드를 새로운 `index.js` 파일에 복사합니다.
3. 다음 `zip` 명령을 사용하여 배포 패키지를 생성합니다.

```
zip function.zip index.js
```

4. create-function AWS CLI 명령을 사용하여 Lambda 함수를 생성합니다. `role` 파라미터에 앞서 생성한 실행 역할의 ARN을 입력합니다.

```
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
```

함수 테스트



`invoke` AWS CLI 명령 및 샘플 Amazon SQS 이벤트를 사용하여 Lambda 함수를 수동으로 호출합니다.

샘플 이벤트를 사용하여 Lambda 함수를 간접적으로 호출하려면

1. 다음 JSON을 `input.json`라는 파일로 저장합니다. 이 JSON은 Amazon SQS가 `"body"`에 대기열의 실제 메시지를 포함하는 Lambda 함수로 보낼 수 있는 이벤트를 시뮬레이션합니다. 이 예제에서 메시지는 `"test"`입니다.

   Example Amazon SQS 이벤트

   이는 테스트 이벤트이므로 메시지나 계정 번호를 변경할 필요가 없습니다.

   ```
   {
       "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. 다음 AWS CLI 간접 호출 명령을 실행합니다. 이 명령은 응답으로 CloudWatch 로그를 반환합니다. 로그 검색에 대한 자세한 내용은 AWS CLI를 사용하여 로그에 액세스 단원을 참조하십시오.

   ```
   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
   ```

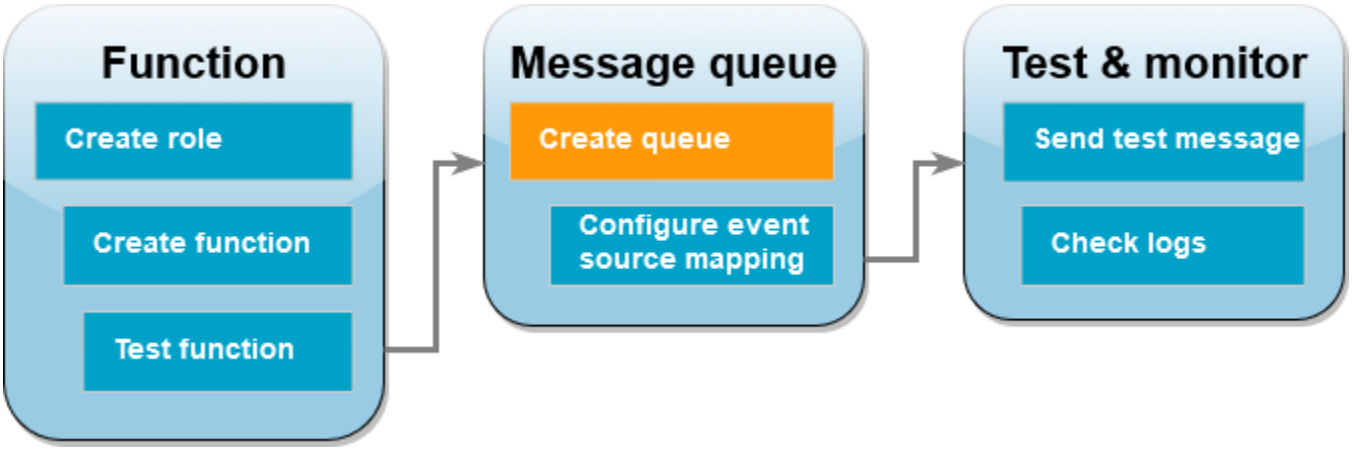
   cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세

요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

3. 응답에서 INFO 로그를 찾습니다. 여기에 Lambda 함수가 메시지 본문을 기록합니다. 다음과 유사한 로그가 표시되어야 합니다.

```
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 대기열 생성



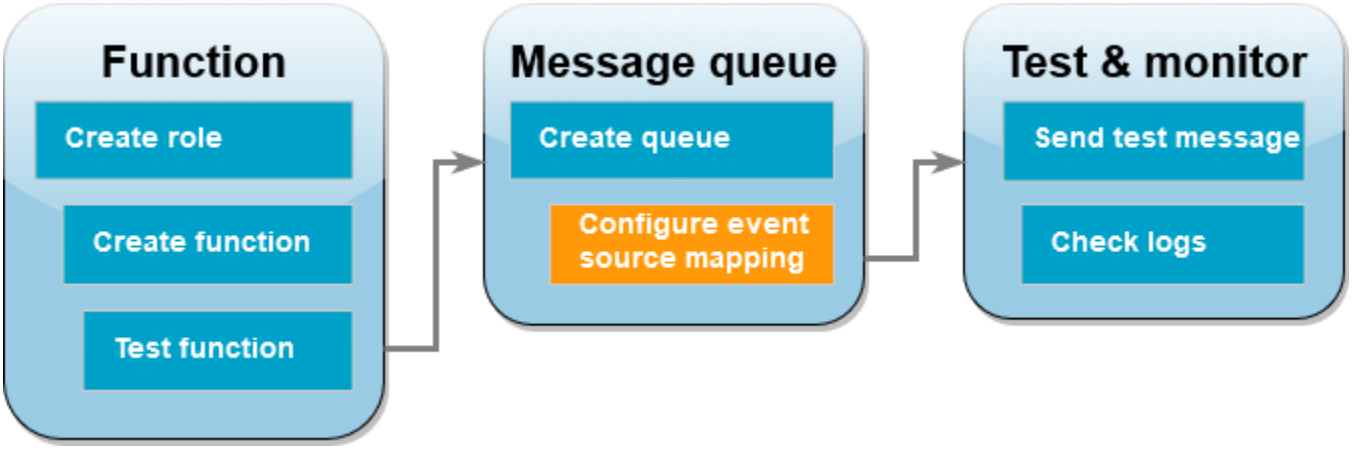
Lambda 함수가 이벤트 소스로 사용할 수 있는 Amazon SQS 대기열을 생성합니다.

대기열을 생성하려면

1. Amazon SQS 콘솔을 엽니다.
2. Create queue(대기열 생성)를 선택합니다.
3. 대기열의 이름을 입력합니다. 다른 모든 옵션은 기본 설정으로 둡니다.
4. 대기열 생성을 선택합니다.

대기열을 생성한 후 해당 ARN을 기록해 둡니다. 다음 단계에서 대기열을 Lambda 함수와 연결할 때 필요합니다.

이벤트 소스 구성



이벤트 소스 매핑을 생성하여 Amazon SQS 대기열을 Lambda 함수에 연결합니다. 이벤트 소스 매핑은 Amazon SQS 대기열을 읽고 새 메시지가 추가되면 Lambda 함수를 간접적으로 호출합니다.

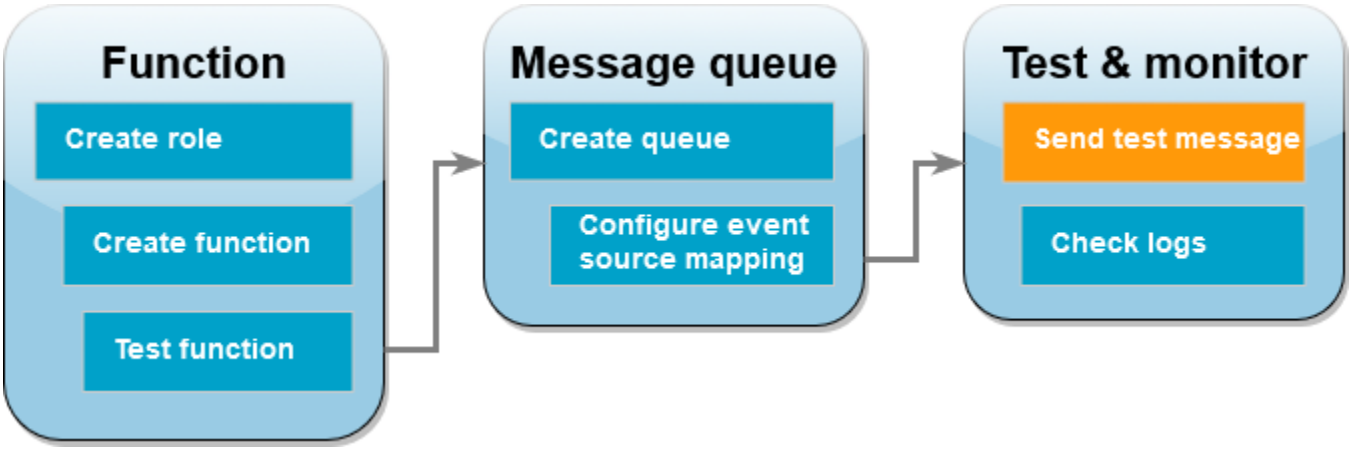
Amazon SQS 대기열과 Lambda 함수 간에 매핑을 생성하려면 AWS CLI create-event-source-mapping 명령을 사용합니다. 예제

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

이벤트 소스 매핑 목록을 가져오려면 list-event-source-mappings 명령을 사용합니다. 예제

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

테스트 메시지 보내기

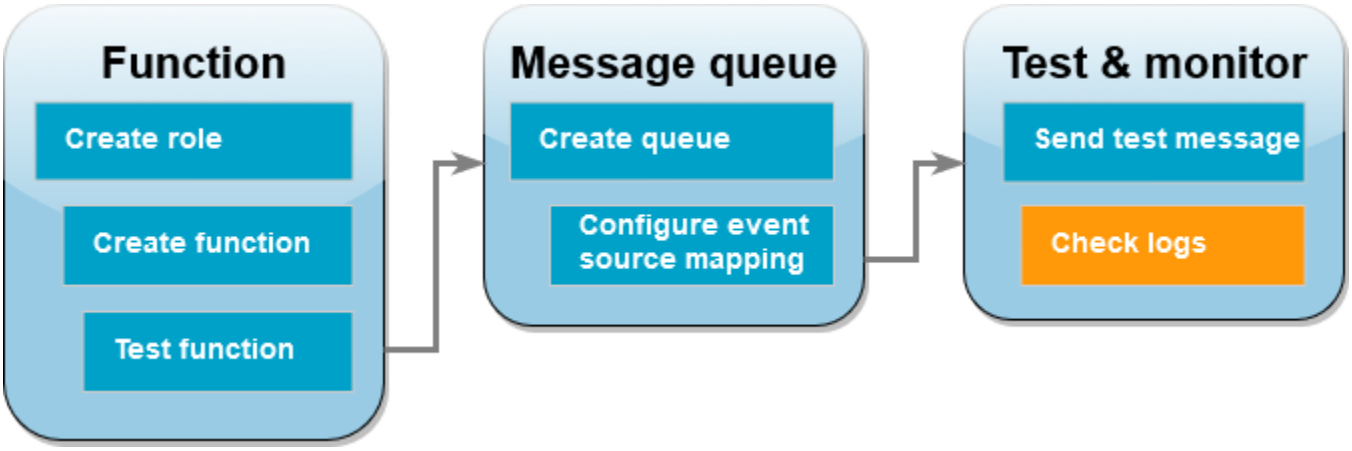


Amazon SQS 메시지를 Lambda 함수로 보내려면

1. Amazon SQS 콘솔을 엽니다.
2. 이전에 생성한 대기열을 선택합니다.
3. [메시지 전송 및 수신(Send and receive messages)]을 선택합니다.
4. 메시지 본문에 테스트 메시지(예: “이것은 테스트 메시지입니다.”)를 입력합니다.
5. 메시지 전송을 선택합니다.

Lambda는 업데이트를 위해 대기열을 폴링합니다. 새 메시지가 있는 경우 Lambda는 대기열에서 이 새 이벤트 데이터를 사용하여 함수를 호출합니다. 함수 핸들러가 예외 없이 반환되면 Lambda는 메시지가 성공적으로 처리된 것으로 간주하고 대기열의 새 메시지 읽기를 시작합니다. 메시지를 성공적으로 처리하면 Lambda는 대기열에서 메시지를 자동으로 삭제합니다. 핸들러가 예외를 발생시키면 Lambda는 메시지 배치가 성공적으로 처리되지 않은 것으로 간주하고 Lambda는 동일한 메시지 배치로 함수를 호출합니다.

CloudWatch Logs 확인



함수가 메시지를 처리했는지 확인하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. ProcessSQSRecord 함수를 선택합니다.
3. 모니터(Monitor)를 선택합니다.
4. CloudWatch 로그 보기를 선택합니다.
5. CloudWatch 콘솔에서 함수의 로그 스트림을 선택합니다.
6. `INFO` 로그를 찾습니다. 여기에 Lambda 함수가 메시지 본문을 기록합니다. Amazon SQS 대기열을 통해 보낸 메시지가 표시됩니다. 예제

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

Amazon SQS 대기열을 삭제하려면

1. AWS Management Console에 로그인한 후 https://console.aws.amazon.com/sqs/에서 Amazon SQS 콘솔을 엽니다.
2. 생성한 대기열을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 **confirm**을 입력합니다.
5. 삭제를 선택합니다.

자습서: 교차 계정 Amazon SQS 대기열을 이벤트 소스로 사용

이 자습서에서는 다른 AWS 계정의 Amazon Simple Queue Service(Amazon SQS) 대기열에서 메시지를 사용하는 Lambda 함수를 생성합니다. 이 자습서에는 2개의 AWS 계정, 즉 Lambda 함수가 포함된 계정을 나타내는 계정 A와 Amazon SQS 대기열이 포함된 계정을 나타내는 계정 B가 사용됩니다.

필수 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 생성(계정 A)

계정 A에서 필요한 AWS 리소스에 액세스하는 함수 권한을 부여하는 실행 역할을 만듭니다.

실행 역할을 만들려면

1. AWS Identity and Access Management(IAM) 콘솔에서 역할(Roles) 페이지를 엽니다.

2. 역할 생성을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.
   - 신뢰할 수 있는 엔터티 – AWS Lambda
   - 권한 – AWSLambdaSQSQueueExecutionRole
   - 역할 이름 – **cross-account-lambda-sqs-role**

AWSLambdaSQSQueueExecutionRole 정책은 함수가 Amazon SQS에서 항목을 읽고 Amazon CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다.

함수 생성(계정 A)

계정 A에서 Amazon SQS 메시지를 처리하는 Lambda 함수를 생성합니다. 다음 Node.js 18 코드 예는 각 메시지를 CloudWatch Logs의 로그에 기록합니다.

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

함수를 만들려면

> **Note**
>
> 다음 단계에 따라 Node.js 18에서 함수를 생성합니다. 다른 언어의 경우 단계는 비슷하지만 일부 세부 사항은 다릅니다.

1. 예제 코드를 `index.mjs`라는 파일로 저장합니다.
2. 배포 패키지를 만듭니다.

   ```
   zip function.zip index.mjs
   ```

3. `create-function` AWS Command Line Interface(AWS CLI) 명령을 사용하여 함수를 만듭니다.

```
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
```

함수 테스트(계정 A)

계정 A에서 invoke AWS CLI 명령 및 샘플 Amazon SQS 이벤트를 사용하여 Lambda 함수를 수동으로 테스트합니다.

핸들러가 예외 없이 정상적으로 반환되면 Lambda는 메시지가 성공적으로 처리된 것으로 간주하고 대기열의 새 메시지 읽기를 시작합니다. 메시지를 성공적으로 처리하면 Lambda는 대기열에서 메시지를 자동으로 삭제합니다. 핸들러가 예외를 발생시키면 Lambda는 메시지 배치가 성공적으로 처리되지 않은 것으로 간주하고 Lambda는 동일한 메시지 배치로 함수를 호출합니다.

1. 다음 JSON을 input.txt라는 파일로 저장합니다.

```
{
    "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"
        }
    ]
}
```

앞의 JSON은 Amazon SQS가 "body"에 대기열의 실제 메시지를 포함하는 Lambda 함수로 보낼 수 있는 이벤트를 시뮬레이션합니다.

2. 다음 invoke AWS CLI 명령을 실행합니다.

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

3. `outputfile.txt` 파일의 출력을 확인합니다.

Amazon SQS 대기열 생성(계정 B)

계정 B에서 계정 A의 Lambda 함수가 이벤트 소스로 사용할 수 있는 Amazon SQS 대기열을 생성합니다.

대기열을 생성하려면

1. Amazon SQS 콘솔을 엽니다.
2. Create queue(대기열 생성)를 선택합니다.
3. 다음 속성을 사용하여 대기열을 만듭니다.

   - Type – Standard
   - Name – LambdaCrossAccountQueue
   - Configuration - 기본 설정을 유지합니다.
   - Access policy – Advanced를 선택합니다. 다음 JSON 정책을 붙여넣습니다.

```
{
   "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"
    }
  ]
}
```

이 정책은 이 Amazon SQS 대기열에서 발생하는 메시지를 사용할 수 있는 계정 A 권한에 Lambda 실행 역할을 부여합니다.

4. 대기열을 만든 후 대기열의 Amazon 리소스 이름(ARN)을 기록합니다. 다음 단계에서 대기열을 Lambda 함수와 연결할 때 필요합니다.

이벤트 소스 구성(계정 A)

다음 `create-event-source-mapping` AWS CLI 명령을 실행하여, 계정 A에서 계정 B의 Amazon SQS 대기열 간에 이벤트 소스 매핑과 Lambda 함수를 생성합니다.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-
size 10 \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

이벤트 소스 매핑 목록을 가져오려면 다음 명령을 실행합니다.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:<AccountB_ID>:LambdaCrossAccountQueue
```

설정 테스트

이제 다음과 같이 설정을 테스트할 수 있습니다.

1. 계정 B에서 Amazon SQS 콘솔을 엽니다.
2. 앞서 생성한 LambdaCrossAccountQueue를 선택합니다.
3. [메시지 전송 및 수신(Send and receive messages)]을 선택합니다.
4. 메시지 본문 아래에 테스트 메시지를 입력합니다.
5. 메시지 전송을 선택합니다.

계정 A의 Lambda 함수가 이 메시지를 수신해야 합니다. Lambda는 계속해서 업데이트를 위해 대기열을 폴링합니다. 새 메시지가 있는 경우 Lambda는 대기열에서 이 새 이벤트 데이터를 사용하여 함수를

호출합니다. 함수는 Amazon CloudWatch에서 로그를 실행하고 생성합니다. CloudWatch 콘솔에서 로그를 확인할 수 있습니다.

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

계정 A에서 실행 역할 및 Lambda 함수를 정리합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**를 입력하고 Delete(삭제)를 선택합니다.

계정 B에서 Amazon SQS 대기열을 정리합니다.

Amazon SQS 대기열을 삭제하려면

1. AWS Management Console에 로그인한 후 https://console.aws.amazon.com/sqs/에서 Amazon SQS 콘솔을 엽니다.
2. 생성한 대기열을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 **`confirm`**을 입력합니다.
5. Delete(삭제)를 선택합니다.

Lambda로 Amazon DocumentDB 이벤트 처리

Amazon DocumentDB 클러스터를 이벤트 소스로 구성하면 Lambda 함수를 사용하여 Amazon DocumentDB(MongoDB 호환) 변경 스트림의 이벤트를 처리할 수 있습니다. 그런 다음 Amazon DocumentDB 클러스터에서 데이터가 변경될 때마다 Lambda 함수를 호출하여 이벤트 기반 워크로드를 자동화할 수 있습니다.

> **Note**
>
> Lambda는 Amazon DocumentDB 버전 4.0 및 5.0만 지원합니다. Lambda는 버전 3.6을 지원하지 않습니다.
> 또한 이벤트 소스 매핑의 경우 Lambda는 인스턴스 기반 클러스터와 리전 클러스터만 지원합니다. Lambda는 탄력적 클러스터 또는 글로벌 클러스터를 지원하지 않습니다. Lambda를 클라이언트로 사용하여 Amazon DocumentDB에 연결할 때는 이 제한이 적용되지 않습니다. Lambda는 모든 클러스터 유형에 연결하여 CRUD 작업을 수행할 수 있습니다.

Lambda는 Amazon DocumentDB 변경 스트림의 이벤트를 도착 순서대로 순차적으로 처리합니다. 따라서 함수는 DocumentDB에서 한 번에 하나의 동시 호출만 처리할 수 있습니다. 함수를 모니터링하기 위해 함수의 동시성 지표를 추적할 수 있습니다.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

주제

- Amazon DocumentDB 이벤트 소스 모니터링
- 자습서: Amazon DocumentDB Streams와 함께 AWS Lambda 사용

예시 Amazon DocumentDB 이벤트

```
{
    "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"
}
```

이 예제의 이벤트와 해당 셰이프에 대한 자세한 내용은 MongoDB Documentation 웹 사이트의 Change Events를 참조하세요.

사전 조건 및 권한

Lambda 함수에 Amazon DocumentDB를 이벤트 소스로 사용하기 전에 다음 사전 요구 사항을 확인하세요. 다음을 수행해야 합니다.

- 함수와 동일한 AWS 계정 및 AWS 리전에 기존 Amazon DocumentDB 클러스터가 있어야 합니다. 기존 클러스터가 없다면 Amazon DocumentDB 개발자 안내서의 Get Started with Amazon DocumentDB에 나와 있는 단계에 따라 하나 생성하면 됩니다. 또는 자습서: Amazon DocumentDB Streams와 함께 AWS Lambda 사용 안내서의 첫 번째 단계 집합은 필요한 모든 사전 조건이 있는 DocumentDB 클러스터를 생성하는 과정을 안내합니다.
- Lambda가 Amazon DocumentDB 클러스터와 연결된 Amazon Virtual Private Cloud(Amazon VPC) 리소스에 액세스할 수 있도록 허용해야 합니다. 자세한 내용은 네트워크 구성 단원을 참조하십시오.
- Amazon DocumentDB 클러스터에서 TLS를 활성화합니다. 이것이 기본 설정입니다. TLS를 사용하지 않도록 설정하면 Lambda가 클러스터와 통신할 수 없습니다.
- Amazon DocumentDB 클러스터에서 변경 스트림을 활성화해야 합니다. 자세한 내용은 Amazon DocumentDB 개발자 안내서의 Using Change Streams with Amazon DocumentDB를 참조하세요.
- Amazon DocumentDB 클러스터에 액세스할 수 있도록 Lambda에 보안 인증을 제공해야 합니다. 이벤트 소스를 설정할 때 클러스터에 액세스하는 데 필요한 인증 세부 정보(사용자 이름 및 암호)가 포함된 AWS Secrets Manager 키를 제공합니다. 설치 중 이 키를 제공하려면 다음 중 하나를 수행합니다.
  - 설정에 Lambda 콘솔을 사용하는 경우 Secrets Manager 키 필드에 키를 제공합니다.
  - 설정에 AWS Command Line Interface(AWS CLI)를 사용하는 경우 `source-access-configurations` 옵션에 이 키를 제공합니다. `create-event-source-mapping` 명령 또는 `update-event-source-mapping` 명령과 함께 이 옵션을 포함할 수 있습니다. 예:

```
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 스트림과 관련된 리소스를 관리할 수 있는 권한을 Lambda에 부여해야 합니다. 함수의 실행 역할에 다음 권한을 수동으로 추가합니다.

- rds:DescribeDBClusters
- rds:DescribeDBClusterParameters
- rds:DescribeDBSubnetGroups
- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups
- kms:Decrypt
- secretsmanager:GetSecretValue

- Lambda로 전송하는 Amazon DocumentDB 변경 스트림 이벤트의 크기를 6MB 미만으로 유지해야 합니다. Lambda는 최대 6MB의 페이로드 크기를 지원합니다. 변경 스트림이 Lambda에 6MB보다 큰 이벤트를 전송하려고 하면 Lambda는 메시지를 삭제하고 `OversizedRecordCount` 지표를 내보냅니다. Lambda는 최대한 모든 지표를 전송합니다.

> **Note**
>
> Lambda 함수의 최대 제한 시간은 일반적으로 15분이지만 Amazon MSK, 자체 관리형 Apache Kafka, Amazon DocumentDB, ActiveMQ 및 RabbitMQ용 Amazon MQ에 대한 이벤트 소스 매핑은 최대 제한 시간이 14분인 함수만 지원합니다. 이 제약 조건에 따라 이벤트 소스 매핑에서 함수 오류 및 재시도를 적절히 처리할 수 있습니다.

네트워크 구성

Lambda가 Amazon DocumentDB 클러스터를 이벤트 소스로 사용하려면 클러스터가 있는 Amazon VPC에 액세스해야 합니다. Lambda가 VPC에 액세스할 수 있도록 AWS PrivateLink VPC 엔드포인트를 배포하는 것이 좋습니다. Lambda용 VPC 엔드포인트를 배포하고, 클러스터가 인증을 사용하는 경우 Secrets Manager용 VPC 엔드포인트도 배포합니다.

또는 Amazon DocumentDB 클러스터와 연결된 VPC에 퍼블릭 서브넷당 하나의 NAT 게이트웨이가 포함되는지 확인합니다. 자세한 내용은 the section called “VPC 함수에 대한 인터넷 액세스” 단원을 참조하십시오.

또한 VPC 엔드포인트를 사용하는 경우 프라이빗 DNS 이름을 활성화하도록 구성해야 합니다.

Amazon DocumentDB 클러스터에 대한 이벤트 소스 매핑을 생성하는 경우 Lambda는 클러스터 VPC의 서브넷 및 보안 그룹에 대해 ENI(탄력적 네트워크 인터페이스)가 이미 존재하는지 확인합니다. Lambda가 기존 ENI를 찾으면 이를 재사용하려고 시도합니다. 그렇지 않으면 Lambda가 이벤트 소스에 연결하고 함수를 호출하기 위해 새 ENI를 생성합니다.

> **Note**
>
> Lambda 함수는 항상 Lambda 서비스가 소유한 VPC 내에서 실행됩니다. 이러한 VPC는 서비스에 의해 자동으로 유지 관리되며 고객에게는 표시되지 않습니다. 또한 함수를 Amazon VPC에 연결할 수도 있습니다. 어느 경우든 함수의 VPC 구성은 이벤트 소스 매핑에 영향을 미치지 않습니다. 이벤트 소스의 VPC 구성에 따라 Lambda가 이벤트 소스에 연결되는 방식이 결정됩니다.

VPC 보안 그룹 규칙

최소한 다음 규칙을 사용하여 클러스터가 포함된 Amazon VPC의 보안 그룹을 구성합니다.

- 인바운드 규칙 - 이벤트 소스에 대해 지정된 보안 그룹에 대해 Amazon DocumentDB 클러스터 포트의 모든 트래픽을 허용하세요. Amazon DocumentDB는 기본적으로 포트 27017을 사용합니다.
- 아웃바운드 규칙 - 모든 대상에 대해 포트 443의 모든 트래픽을 허용합니다. Amazon DocumentDB 클러스터 포트에서 모든 트래픽을 허용하세요. Amazon DocumentDB는 기본적으로 포트 27017을 사용합니다.
- NAT 게이트웨이 대신 VPC 엔드포인트를 사용하는 경우 VPC 엔드포인트와 연결된 보안 그룹은 이벤트 소스의 보안 그룹에서 포트 443의 모든 인바운드 트래픽을 허용해야 합니다.

VPC 엔드포인트 작업

VPC 엔드포인트를 사용하는 경우 함수를 호출하는 API 직접 호출은 ENI를 사용하여 이러한 엔드포인트를 통해 라우팅됩니다. Lambda 서비스 보안 주체는 해당 ENI를 사용하는 모든 함수에서 `lambda:InvokeFunction`을 호출해야 합니다.

기본적으로 VPC 엔드포인트에는 개방적인 IAM 정책이 있습니다. 모범 사례는 특정 보안 주체만 해당 엔드포인트를 사용하여 필요한 작업을 수행할 수 있도록 이러한 정책을 제한하는 것입니다. 이벤트 소스 매핑이 Lambda 함수를 호출할 수 있도록 하려면 VPC 엔드포인트 정책에서 Lambda 서비스 원칙이 `lambda:InvokeFunction`을 호출할 수 있도록 허용해야 합니다. 조직 내에서 발생하는 API 직접 호출만 허용하도록 VPC 엔드포인트 정책을 제한하면 이벤트 소스 매핑이 제대로 작동하지 않습니다.

다음 예제 VPC 엔드포인트 정책은 Lambda 엔드포인트에 필요한 액세스 권한을 부여하는 방법을 설명합니다.

Example VPC 엔드포인트 정책 - Lambda 엔드포인트

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Amazon DocumentDB 클러스터가 인증을 사용하는 경우 Secrets Manager 엔드포인트에 대한 VPC 엔드포인트 정책을 제한할 수도 있습니다. Secrets Manager API를 호출하기 위해 Lambda는 Lambda 서비스 보안 주체가 아닌 함수 역할을 사용합니다. 다음 예제는 Secrets Manager 엔드포인트 정책을 보여줍니다.

Example VPC 엔드포인트 정책 – Secrets Manager 엔드포인트

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "customer_function_execution_role_arn"
                ]
            },
            "Resource": "customer_secret_arn"
        }
    ]
}
```

Amazon DocumentDB 이벤트 소스 매핑 생성(콘솔)

Lambda 함수가 Amazon DocumentDB 클러스터의 변경 스트림에서 읽을 수 있도록 이벤트 소스 매핑을을 생성합니다. 이 섹션에서는 Lambda 콘솔에서 이를 수행하는 방법을 설명합니다. AWS SDK 및 AWS CLI 지침은 the section called “Amazon DocumentDB 이벤트 소스 매핑 생성(SDK 또는 CLI)” 섹션을 참조하세요.

Amazon DocumentDB 이벤트 소스 매핑 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.
4. 트리거 구성 아래의 드롭다운 목록에서 DocumentDB를 선택합니다.
5. 필요한 옵션을 구성한 다음 추가를 선택합니다.

Lambda는 Amazon DocumentDB 이벤트 소스에 대해 다음과 같은 옵션을 지원합니다.

- DocumentDB 클러스터 - Amazon DocumentDB 클러스터를 선택합니다.
- 트리거 활성화 - 즉시 트리거를 활성화할지 여부를 선택합니다. 이 확인란을 선택하면 이벤트 소스 매핑을 생성할 때 함수가 지정된 Amazon DocumentDB 변경 스트림으로부터 즉시 트래픽을 수신하기 시작합니다. 테스트할 이벤트 소스 매핑을 비활성화된 상태로 생성하려면 확인란 선택을 취소하는 것이 좋습니다. 생성 후에는 언제든지 이벤트 소스 매핑을 활성화할 수 있습니다.
- 데이터베이스 이름 — 클러스터 내에서 사용할 데이터베이스의 이름을 입력합니다.
- (선택 사항) 컬렉션 이름 - 사용할 데이터베이스 내의 컬렉션 이름을 입력합니다. 컬렉션을 지정하지 않으면 Lambda는 데이터베이스의 각 컬렉션에서 모든 이벤트를 수신합니다.
- 배치 크기 — 단일 배치에서 검색할 최대 메시지 수를 설정합니다. 최대 10,000. 기본 크기는 100입니다.
- 시작 위치 — 스트림에서 레코드 읽기를 시작할 위치를 선택합니다.
  - 최신 — 스트림에 추가된 새 레코드만 처리합니다. 함수는 Lambda가 이벤트 소스 생성을 완료한 후에만 레코드 처리를 시작합니다. 즉, 이벤트 소스가 성공적으로 생성될 때까지 일부 레코드가 삭제될 수 있습니다.
  - 수평 트리밍 – 스트림의 모든 레코드를 처리합니다. Lambda는 클러스터의 로그 보존 기간을 사용하여 이벤트 읽기를 시작할 위치를 결정합니다. 구체적으로 Lambda는 `current_time - log_retention_duration`에서 읽기를 시작합니다. Lambda가 모든 이벤트를 제대로 읽으려면 이 타임스탬프 전에 변경 스트림이 이미 활성화되어 있어야 합니다.

  - 타임스탬프 – 특정 시간에 시작하는 레코드를 시작합니다. Lambda가 모든 이벤트를 제대로 읽으려면 지정된 타임스탬프 전에 변경 스트림이 이미 활성화되어 있어야 합니다.
- 인증 — 클러스터에서 브로커에 액세스하기 위한 인증 방법을 선택합니다.
  - BASIC_AUTH — 기본 인증을 사용하는 경우 클러스터에 액세스하기 위한 자격 증명이 포함된 Secrets Manager 키를 제공해야 합니다.
- Secrets Manager 키 - Amazon DocumentDB 클러스터에 액세스하는 데 필요한 인증 세부 정보(사용자 이름 및 암호)가 포함된 Secrets Manager 키를 선택합니다.
- (선택 사항) 배치 기간 – 함수를 호출하기 전에 레코드를 수집할 최대 시간(초)을 최대 300초로 설정합니다.
- (선택 사항) 전체 문서 구성 - 문서 업데이트 작업의 경우 스트림으로 전송할 내용을 선택합니다. 기본값은 `Default`이며, 이는 각 변경 스트림 이벤트에 대해 Amazon DocumentDB가 변경 내용을 설명하는 델타만 전송한다는 것을 의미합니다. 이 필드에 대한 자세한 내용은 MongoDB Javadoc API 설명서의 FullDocument를 참조하세요.
  - 기본값 — Lambda는 변경 사항을 설명하는 문서의 일부만 전송합니다.
  - UpdateLookup — Lambda는 전체 문서의 복사본과 함께 변경 사항을 설명하는 델타를 전송합니다.

Amazon DocumentDB 이벤트 소스 매핑 생성(SDK 또는 CLI)

AWS SDK를 사용하여 Amazon DocumentDB 이벤트 소스 매핑을 생성하거나 관리하려면 다음 API 작업을 사용합니다.

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

AWS CLI를 사용하여 이벤트 소스 매핑을 생성하려면 `create-event-source-mapping` 명령을 사용합니다. 다음 예제에서는 이 명령을 사용하여 `my-function`이라는 함수를 Amazon DocumentDB 변경 스트림에 매핑합니다. 이벤트 소스는 Amazon 리소스 이름(ARN)으로 지정되고, 배치 크기는 500이며 Unix 시간의 타임스탬프부터 시작합니다. 또한 이 명령은 Lambda가 Amazon DocumentDB에 연결하는 데 사용하는 Secrets Manager 키를 지정합니다. 데이터베이스와 읽을 컬렉션을 지정하는 `document-db-event-source-config` 파라미터도 포함되어 있습니다.

```
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"}' \
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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"
}
```

생성 후 update-event-source-mapping 명령을 사용하여 Amazon DocumentDB 이벤트 소스에 대한 설정을 업데이트합니다. 다음 예제는 배치 크기를 1,000으로 업데이트하고 배치 기간을 10초로 업데이트합니다. 이 명령을 실행하려면 `list-event-source-mapping` 명령 또는 Lambda 콘솔을 사용하여 검색할 수 있는 이벤트 소스 매핑의 UUID가 필요합니다.

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

다음과 유사한 출력이 표시되어야 합니다.

```
{
    "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"
}
```

Lambda는 설정을 비동기식으로 업데이트하므로 프로세스가 완료된 후에야 출력에 이러한 변경 사항이 표시되지 않을 수 있습니다. 이벤트 소스 매핑의 현재 설정을 보려면 get-event-source-mapping 명령을 사용합니다.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

다음과 유사한 출력이 표시되어야 합니다.

```
{
    "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 이벤트 소스 매핑을 삭제하려면 delete-event-source-mapping 명령을 사용합니다.

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

폴링 및 스트리밍 시작 위치

이벤트 소스 매핑 생성 및 업데이트 중 스트림 폴링은 최종적으로 일관됩니다.

- 이벤트 소스 매핑 생성 중 스트림에서 이벤트 폴링을 시작하는 데 몇 분 정도 걸릴 수 있습니다.
- 이벤트 소스 매핑 업데이트 중 스트림에서 이벤트 폴링을 중지했다가 다시 시작하는 데 몇 분 정도 걸릴 수 있습니다.

이 동작은 스트림의 시작 위치로 LATEST를 지정하면 이벤트 소스 매핑이 생성 또는 업데이트 중에 이벤트를 놓칠 수 있음을 의미합니다. 누락된 이벤트가 없도록 하기 위해서는 스트림 시작 위치를 TRIM_HORIZON 또는 AT_TIMESTAMP로 지정하세요.

Amazon DocumentDB 이벤트 소스 모니터링

Amazon DocumentDB 이벤트 소스를 모니터링하는 데 도움이 되도록 Lambda는 함수가 레코드 배치 처리를 완료할 때 IteratorAge 지표를 내보냅니다. 반복기 수명은 가장 최근 이벤트의 타임스탬프와 현재 타임스탬프 간의 차이입니다. 기본적으로 IteratorAge 지표는 배치에서 마지막으로 처리된 레코드가 얼마나 오래되었는지를 나타냅니다. 함수가 현재 새 이벤트를 처리하고 있다면 반복기 수명을 사용하여 레코드가 추가된 후 함수에서 레코드를 처리할 때까지 지연 시간을 추정할 수 있습니다. IteratorAge의 증가 추세는 함수에 문제가 있음을 나타낼 수 있습니다. 자세한 내용은 Lambda 함수 지표 작업 단원을 참조하십시오.

Amazon DocumentDB 변경 스트림은 이벤트 간의 큰 시간 간격을 처리하는 데 최적화되지 않았습니다. Amazon DocumentDB 이벤트 소스에서 오랜 기간 동안 이벤트가 수신되지 않으면 Lambda가 이벤트 소스 매핑을 비활성화할 수 있습니다. 이 기간은 클러스터의 크기와 기타 워크로드에 따라 몇 주에서 몇 개월까지 다양할 수 있습니다.

Lambda는 최대 6MB의 페이로드를 지원합니다. 하지만 Amazon DocumentDB 변경 스트림 이벤트의 크기는 최대 16MB일 수 있습니다. 변경 스트림이 Lambda에 6MB보다 큰 변경 스트림 이벤트를 전송

하려고 하면 Lambda는 메시지를 삭제하고 `OversizedRecordCount` 지표를 내보냅니다. Lambda는 최대한 모든 지표를 전송합니다.

자습서: Amazon DocumentDB Streams와 함께 AWS Lambda 사용

이 자습서에서는 Amazon DocumentDB(MongoDB 호환) 변경 스트림의 이벤트를 사용하는 기본 Lambda 함수를 생성합니다. 이 자습서를 완료하는 과정에서 다음 단계를 거치게 됩니다.

- Amazon DocumentDB 클러스터를 설정하고, 연결하고, 변경 스트림을 활성화합니다.
- Lambda 함수를 생성하고 Amazon DocumentDB 클러스터를 함수의 이벤트 소스로 구성합니다.
- Amazon DocumentDB 데이터베이스에 항목을 삽입하여 엔드 투 엔드 설정을 테스트합니다.

주제

- 필수 조건
- AWS Cloud9 환경 생성
- EC2 보안 그룹 생성
- DocumentDB 클러스터 생성
- Secrets Manager에서 보안 암호 생성
- mongo 셸 설치
- DocumentDB 클러스터에 연결
- 변경 스트림 활성화
- 인터페이스 VPC 엔드포인트 생성
- 실행 역할 생성
- Lambda 함수 생성
- Lambda 이벤트 소스 매핑 생성
- 함수 테스트 - 수동 호출
- 함수 테스트 - 레코드 삽입
- 함수 테스트 - 레코드 업데이트
- 함수 테스트 - 레코드 삭제
- 리소스 정리

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

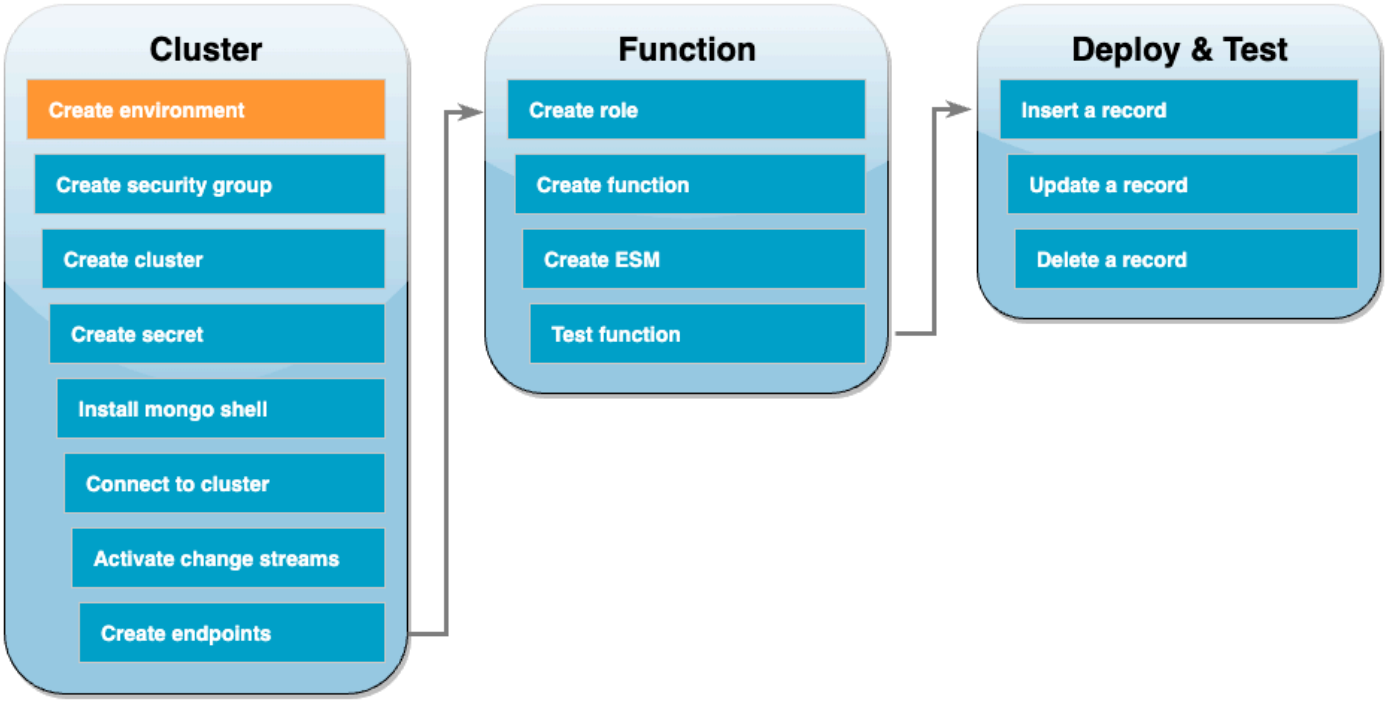
AWS Command Line Interface 설치

아직 AWS Command Line Interface를 설치하지 않은 경우 AWS CLI의 최신 버전 설치 또는 업데이트에서 설명하는 단계에 따라 설치하세요.

이 자습서에서는 명령을 실행할 셸 또는 명령줄 터미널이 필요합니다. Linux 및 macOS에서는 선호하는 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다.

AWS Cloud9 환경 생성



Lambda 함수를 생성하기 전에 Amazon DocumentDB 클러스터를 생성하고 구성해야 합니다. 이 자습서에서 클러스터를 설정하는 단계는 Amazon DocumentDB 시작하기의 절차를 기반으로 합니다.

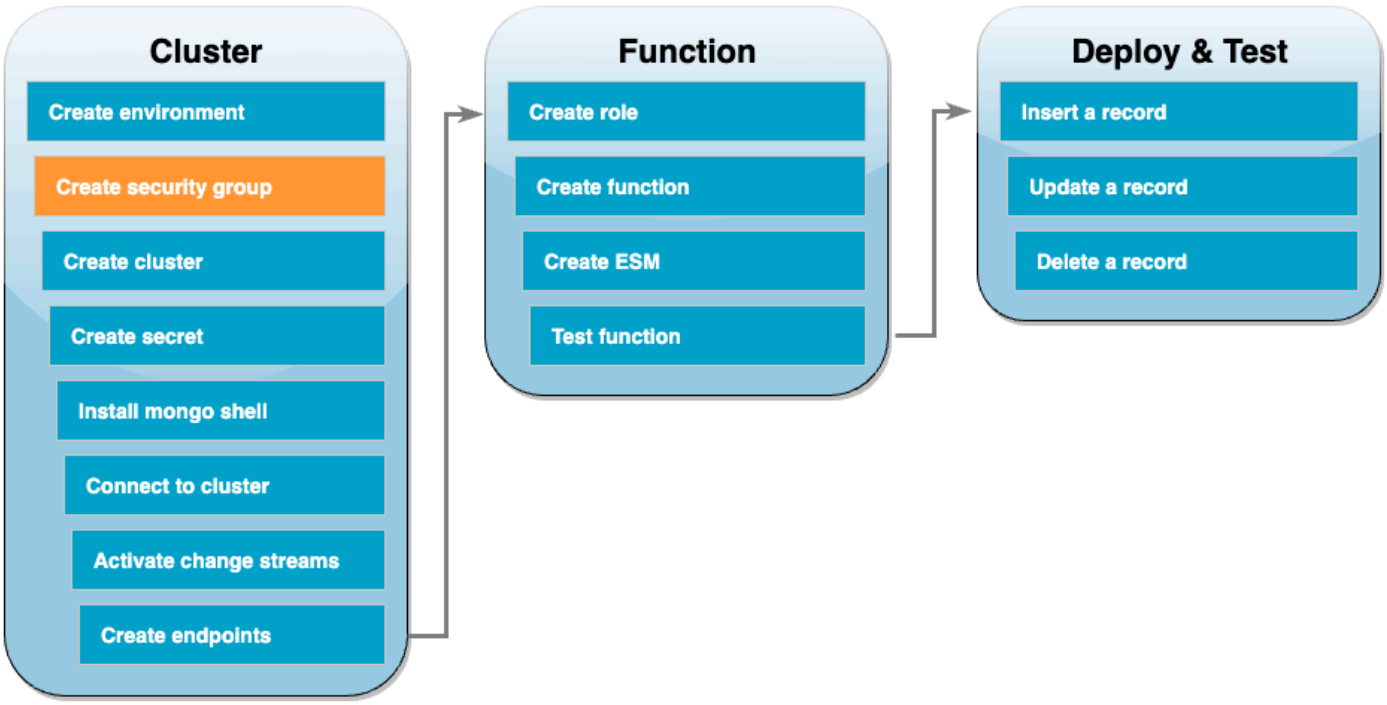
> **Note**
>
> Amazon DocumentDB 클러스터를 이미 설정한 경우 변경 스트림을 활성화하고 필요한 인터페이스 VPC 엔드포인트를 생성해야 합니다. 그런 다음 함수 생성 단계로 바로 건너뛸 수 있습니다.

먼저 AWS Cloud9 환경을 생성합니다. 본 자습서에서는 이 환경을 사용하여 DocumentDB 클러스터에 연결하고 쿼리합니다.

AWS Cloud9 환경을 만들려면

1. Cloud9 콘솔을 열고 환경 생성을 선택합니다.
2. 다음 구성으로 환경을 생성합니다.
   - 세부 정보에서
     - 명칭 – `DocumentDBCloud9Environment`
     - 환경 유형 - 새 EC2 인스턴스
   - 새 EC2 인스턴스에서
     - 인스턴스 유형 - `t2.micro`(1GiB RAM + vCPU 1개)
     - 플랫폼 – Amazon Linux 2
     - 제한 시간 - 30분
   - 네트워크 설정에서
     - 연결 - AWS Systems Manager(SSM)
     - VPC 설정 드롭다운을 확장합니다.
     - Amazon Virtual Private Cloud(VPC) - 기본 VPC를 선택합니다.
     - 서브넷 - 기본 설정 없음
   - 다른 기본 설정을 모두 유지합니다.
3. 생성(Create)을 선택합니다. 새 AWS Cloud9 환경을 프로비저닝하는 데 몇 분이 걸릴 수 있습니다.

EC2 보안 그룹 생성



그런 다음 DocumentDB 클러스터와 Cloud9 환경 간의 트래픽을 허용하는 규칙을 사용하여 EC2 보안 그룹을 생성합니다.

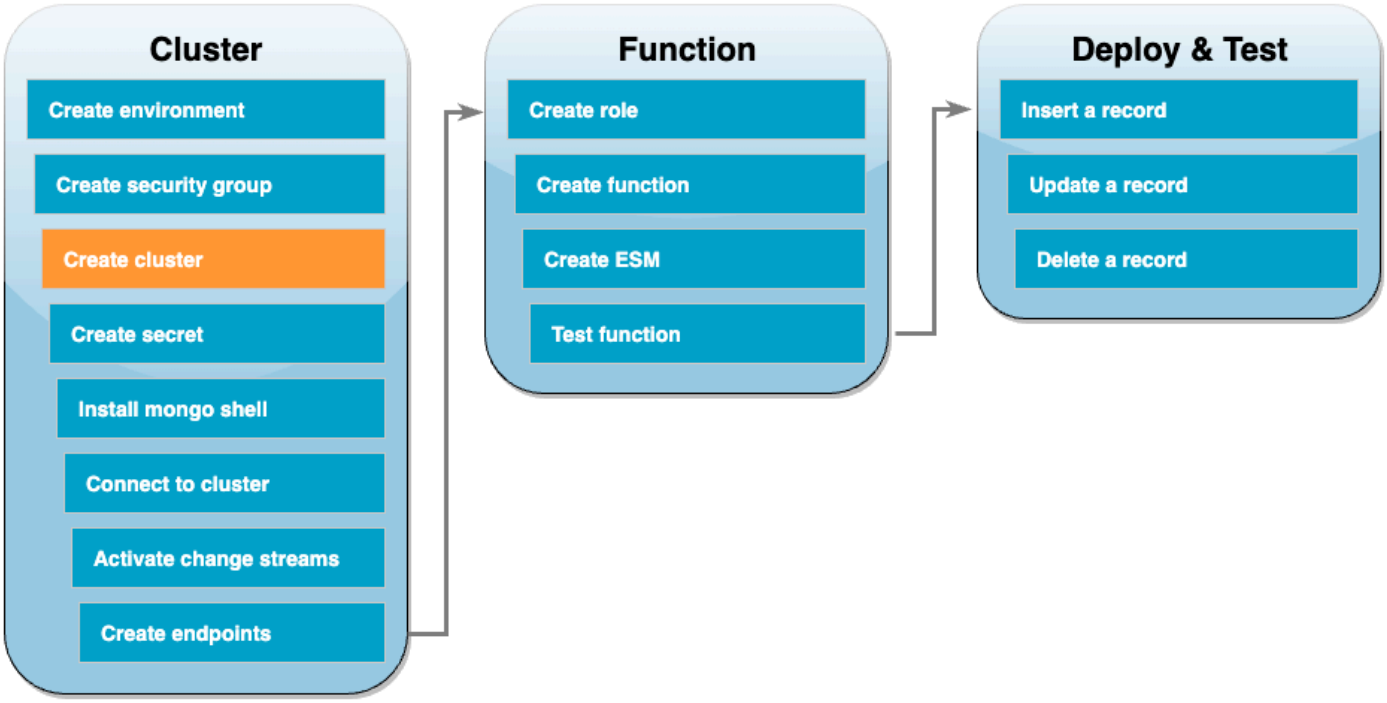
EC2 보안 그룹 생성

1. EC2 콘솔을 엽니다. 네트워크 및 보안에서 보안 그룹을 선택합니다.
2. 보안 그룹 생성을 선택합니다.
3. 다음 구성으로 보안 그룹을 생성합니다.
   - 기본 세부 정보에서
     - 보안 그룹 이름 – `DocDBTutorial`
     - 설명 - Cloud9와 DocumentDB 간의 트래픽을 위한 보안 그룹입니다.
     - VPC – 기본 VPC를 선택합니다.
   - 인바운드 규칙에서 규칙 추가를 선택합니다. 다음 구성으로 규칙을 생성합니다.
     - 유형 - 사용자 지정 TCP
     - 포트 범위 – 27,017
     - 소스 - 사용자 지정

- 소스 옆의 검색 상자에서 이전 단계에서 생성한 AWS Cloud9 환경의 보안 그룹을 선택합니다. 사용할 수 있는 보안 그룹 목록을 보려면 검색 상자에 `cloud9`를 입력합니다. 이름이 `aws-cloud9-<environment_name>`인 보안 그룹을 선택합니다.
- 다른 기본 설정을 모두 유지합니다.

4. 보안 그룹 생성을 선택합니다.

DocumentDB 클러스터 생성



이 단계에서는 이전 단계의 보안 그룹을 사용하여 DocumentDB 클러스터를 생성합니다.

DocumentDB 클러스터 생성

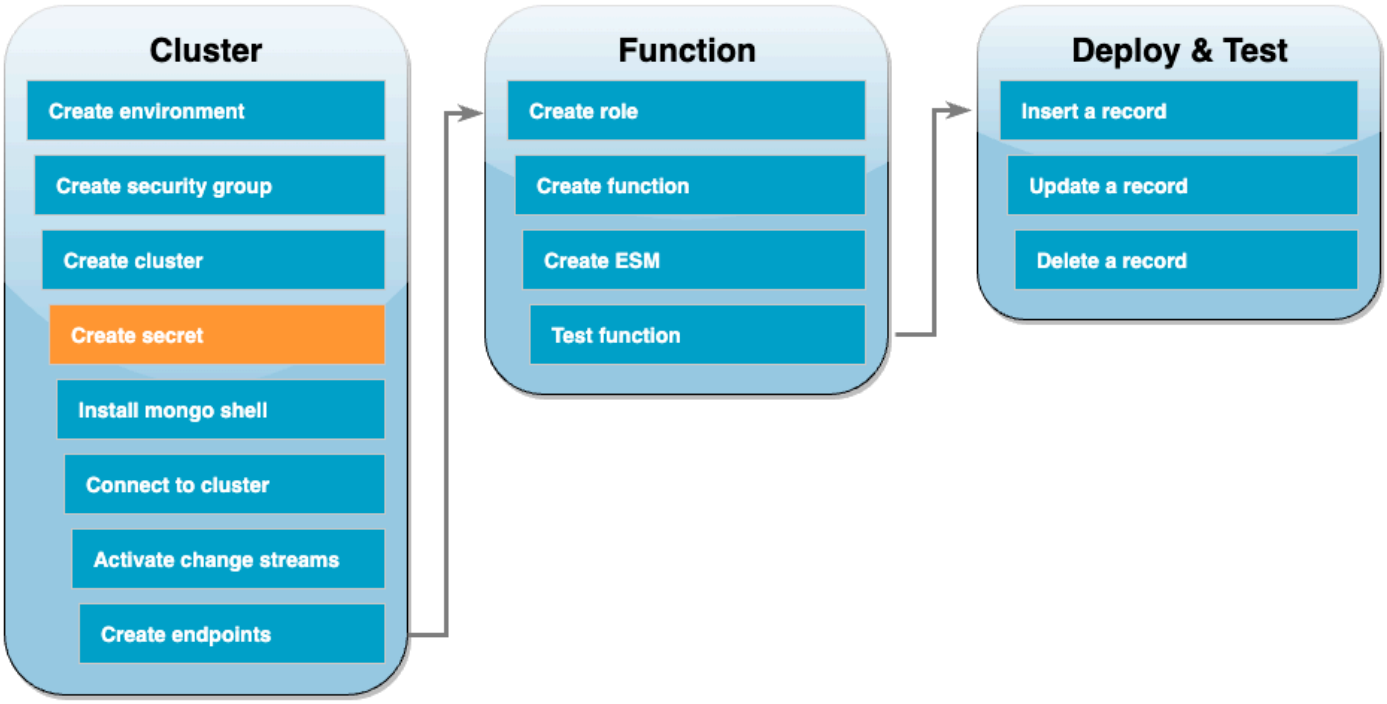
1. DocumentDB 콘솔을 엽니다. 클러스터에서 생성을 선택합니다.
2. 다음 구성을 사용하여 클러스터를 생성합니다.

   - 클러스터 유형에서 인스턴스 기반 클러스터를 선택합니다.
   - 구성에서
     - 엔진 버전 - 5.0.0
     - 인스턴스 클래스 - db.t3.medium(무료 평가판 사용 가능)
     - 인스턴스 개수 – 1.

- 인증에서
  - 클러스터에 연결하는 데 필요한 사용자 이름과 암호(이전 단계에서 보안 암호를 생성하는 데 사용한 것과 동일한 자격 증명)를 입력합니다. 암호 확인에서 암호를 확인합니다.
- 고급 설정 표시를 켭니다.
- 네트워크 설정에서
  - Virtual Private Cloud(VPC) - 기본 VPC를 선택합니다.
  - 서브넷 그룹 - 기본값
  - VPC 보안 그룹 - `default (VPC)` 외에 이전 단계에서 생성한 `DocDBTutorial (VPC)` 보안 그룹을 선택합니다.
- 다른 기본 설정을 모두 유지합니다.

3. 클러스터 생성을 선택합니다. DocumentDB 클러스터를 프로비저닝하는 데 몇 분이 걸릴 수 있습니다.

Secrets Manager에서 보안 암호 생성



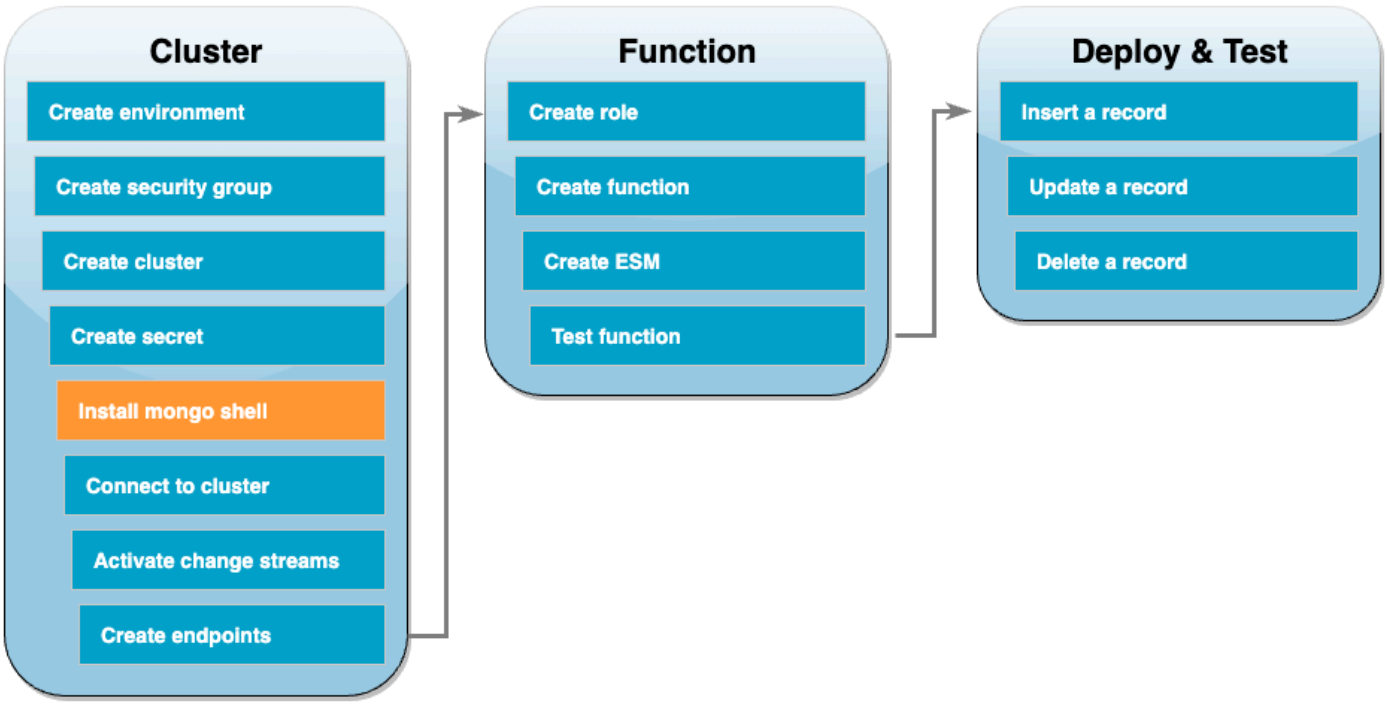
DocumentDB 클러스터에 수동으로 액세스하려면 사용자 이름과 암호 자격 증명을 제공해야 합니다. Lambda가 클러스터에 액세스하려면 이벤트 소스 매핑을 설정할 때 이와 동일한 액세스 자격 증명이 포함된 Secrets Manager 보안 암호를 제공해야 합니다. 이 단계에서 이 보안 암호를 생성합니다.

Secrets Manager에서 보안 암호 생성

1. Secrets Manager 콘솔을 열고 새 보안 암호 저장을 선택합니다.
2. 보안 암호 유형 선택)에서 다음 옵션을 선택합니다.
   - 기본 세부 정보에서
     - 보안 암호 유형 - Amazon DocumentDB 데이터베이스의 자격 증명
     - 자격 증명에 DocumentDB 클러스터에 액세스하는 데 사용할 사용자 이름과 암호를 입력합니다.
     - 데이터베이스 - DocumentDB 클러스터를 선택합니다.
     - Next(다음)를 선택합니다.
3. 보안 암호 구성에서 다음 옵션을 선택합니다.
   - 보안 암호 이름 – `DocumentDBSecret`
   - Next(다음)를 선택합니다.
4. Next(다음)를 선택합니다.
5. 저장(Store)을 선택합니다.
6. 콘솔을 새로 고쳐 `DocumentDBSecret` 보안 암호가 성공적으로 저장되었는지 확인합니다.

보안 암호의 Secret ARN을 적어 둡니다. 이는 이후 단계에서 필요합니다.

mongo 셸 설치



이 단계에서는 Cloud9 환경에 mongo 쉘을 설치합니다. mongo 쉘은 DocumentDB 클러스터에 연결하고 쿼리하는 데 사용하는 명령줄 유틸리티입니다.

Cloud9 환경에 몽고 쉘 설치

1. Cloud9 콘솔을 엽니다. 이전에 생성한 `DocumentDBCloud9Environment` 환경 옆에 있는 Cloud9 IDE 열 아래의 열기 링크를 클릭합니다.
2. 터미널 창에서 다음 명령을 사용하여 MongoDB 리포지토리 파일을 생성합니다.

```
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
```

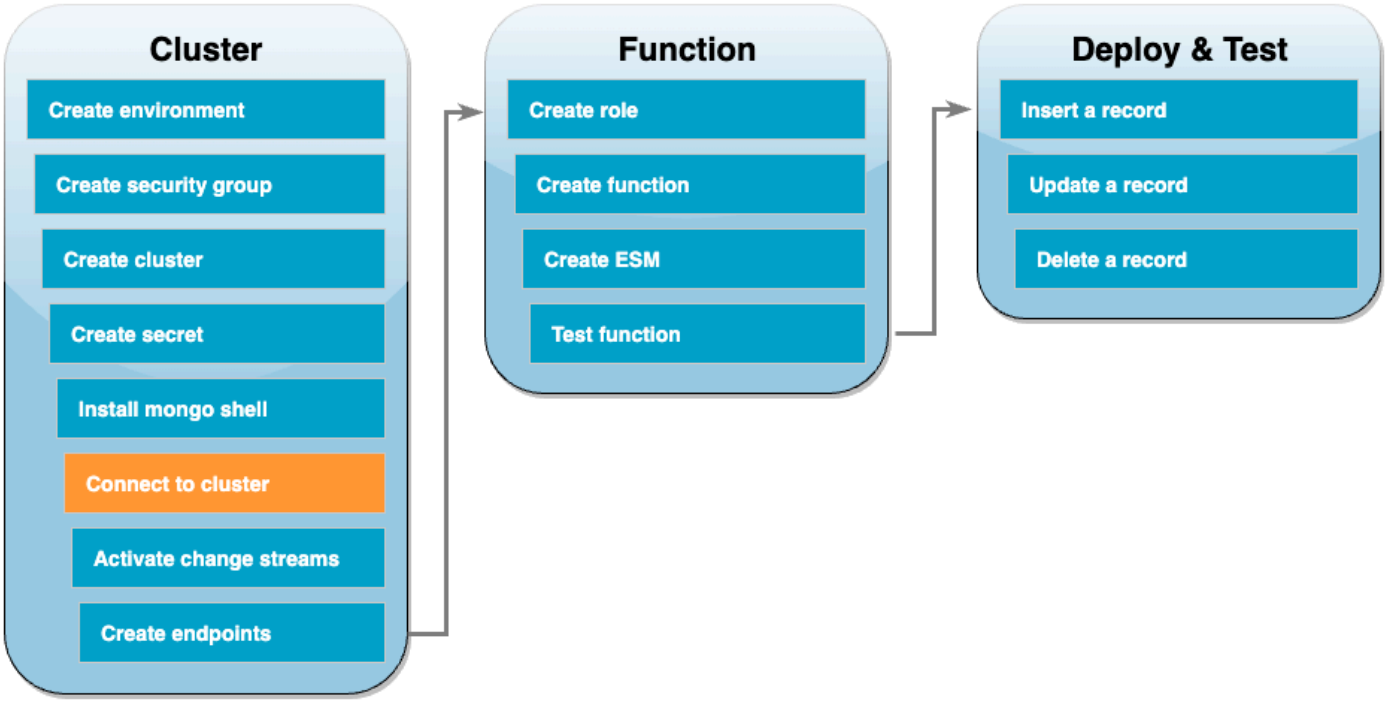
3. 그리고 다음 명령을 사용하여 mongo 쉘을 설치합니다.

```
sudo yum install -y mongodb-org-shell
```

4. 전송 중 데이터를 암호화하려면 Amazon DocumentDB의 퍼블릭 키를 다운로드합니다. 다음 명령은 `global-bundle.pem`이라는 파일을 다운로드합니다.

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

DocumentDB 클러스터에 연결



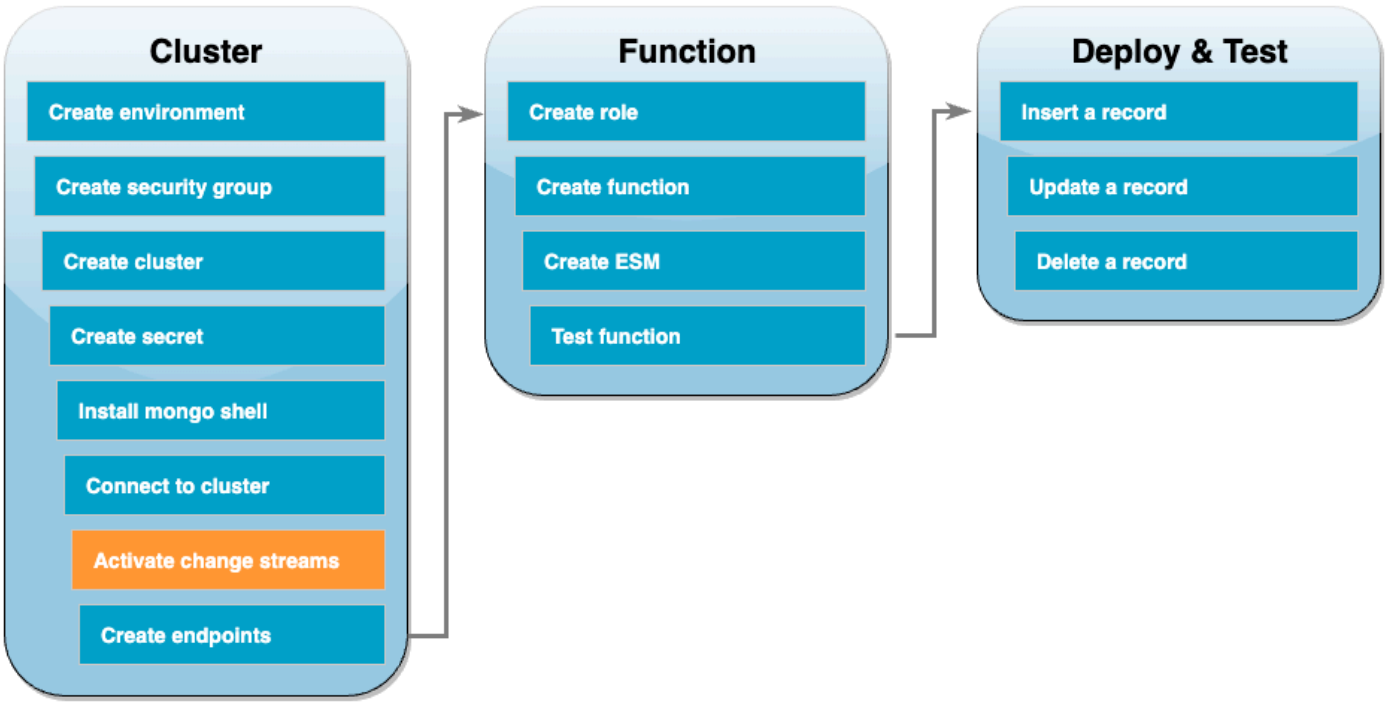
이제 mongo 쉘을 사용하여 DocumentDB 클러스터에 연결할 준비가 되었습니다.

DocumentDB 클러스터에 연결

1. DocumentDB 콘솔을 엽니다. 클러스터에서 클러스터 식별자를 선택하여 클러스터를 선택합니다.
2. 연결 & 보안) 탭의 mongo 셸을 사용하여 이 클러스터에 연결에서 복사를 선택합니다.
3. Cloud9 환경에서 터미널에 이 명령을 붙여넣습니다. `<insertYourPassword>`를 올바른 암호로 바꿉니다.

이 명령을 입력한 후 명령 프롬프트가 `rs0:PRIMARY>`가 되면 Amazon DocumentDB 클러스터에 연결된 것입니다.

변경 스트림 활성화



이 자습서에서는 DocumentDB 클러스터에 있는 docdbdemo 데이터베이스의 products 컬렉션에 대한 변경 사항을 추적합니다. 변경 스트림을 활성화하면 됩니다. 먼저 docdbdemo 데이터베이스를 생성하고 레코드를 삽입하여 테스트합니다.

클러스터 내에 새 데이터베이스 생성

1. Cloud9 환경에서 여전히 DocumentDB 클러스터에 연결되어 있는지 확인합니다.
2. 터미널 창에서 다음 명령을 사용하여 docdbdemo라는 새 데이터베이스를 생성합니다.

```
use docdbdemo
```

3. 그리고 다음 명령을 사용하여 docdbdemo에 레코드를 삽입합니다.

```
db.products.insert({"hello":"world"})
```

   다음과 유사한 출력 화면이 표시되어야 합니다.

```
WriteResult({ "nInserted" : 1 })
```

4. 다음 명령을 사용하여 모든 데이터베이스를 나열합니다.

```
show dbs
```

출력에 docdbdemo 데이터베이스가 포함되어 있는지 확인합니다.

```
docdbdemo  0.000GB
```

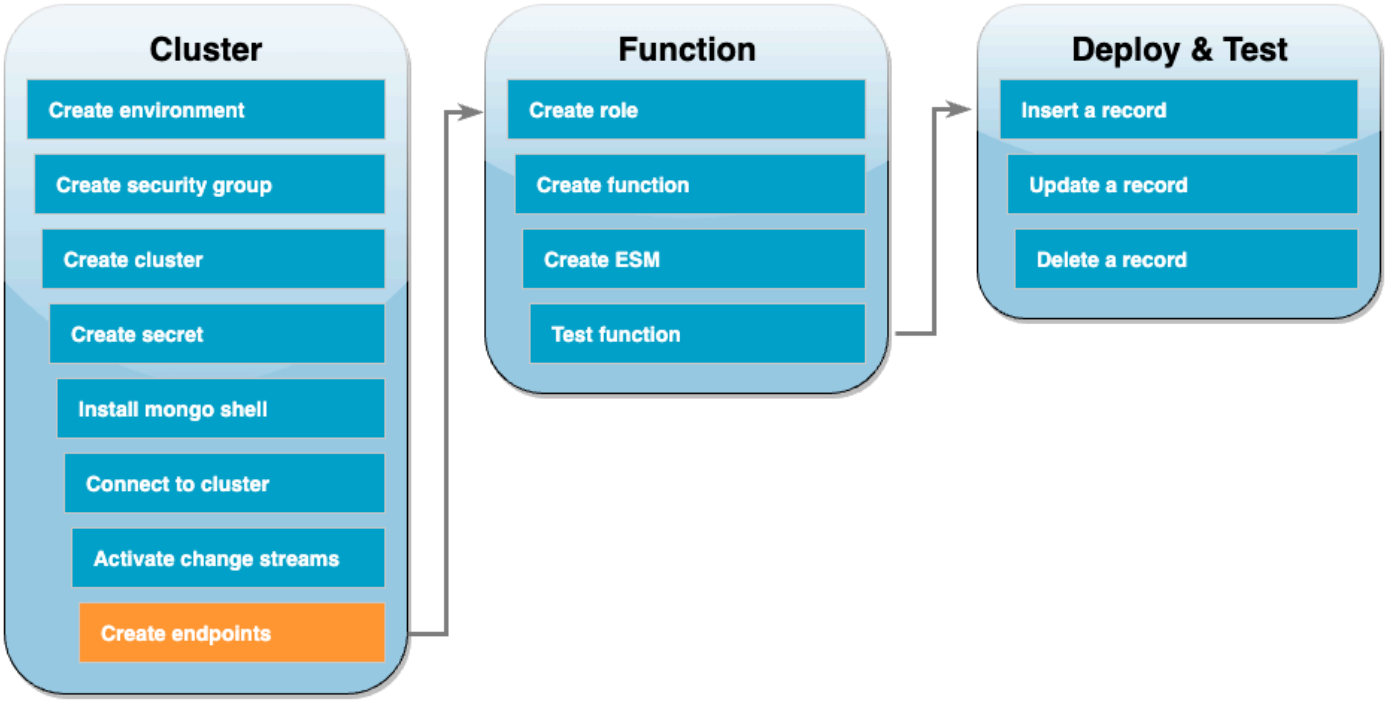
그리고 다음 명령을 사용하여 docdbdemo 데이터베이스의 products 컬렉션에서 변경 스트림을 활성화합니다.

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

인터페이스 VPC 엔드포인트 생성



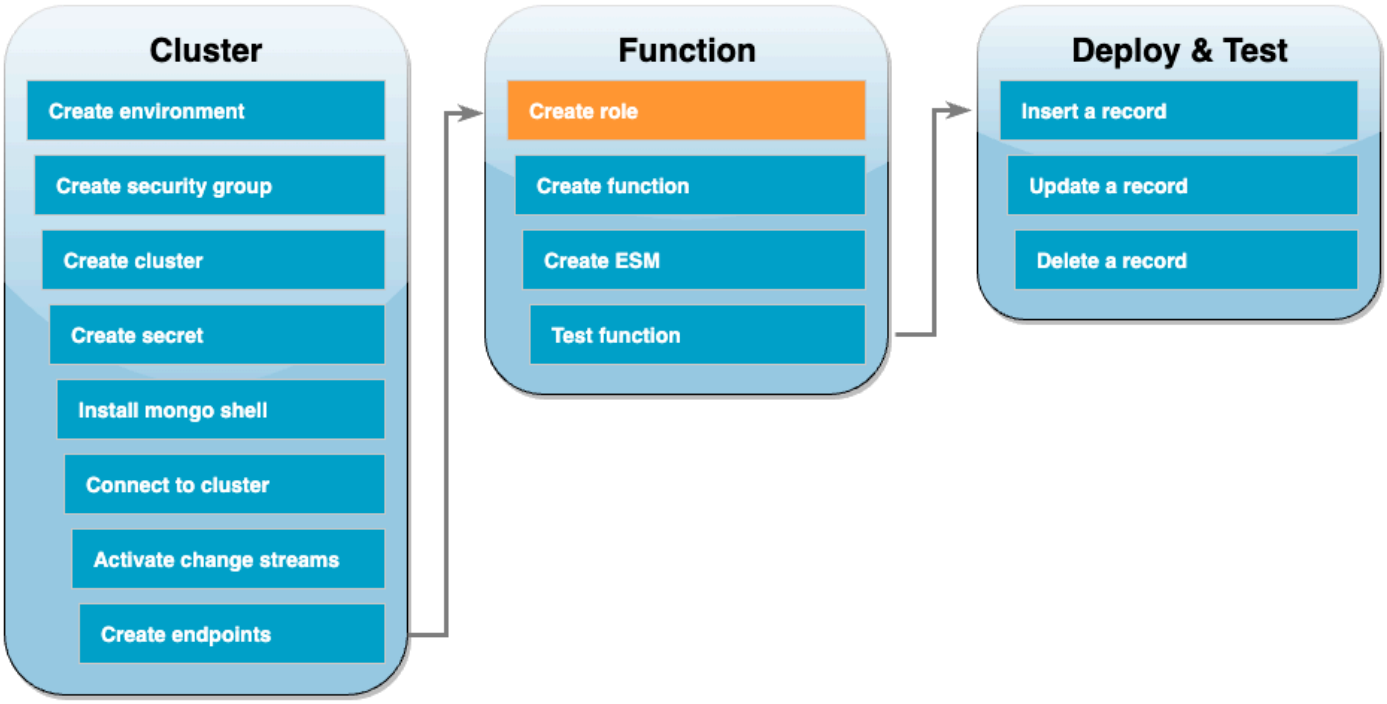
다음으로 인터페이스 VPC 엔드포인트를 생성하여 Lambda와 Secrets Manager(나중에 클러스터 액세스 자격 증명을 저장하는 데 사용됨)가 기본 VPC에 연결할 수 있는지 확인합니다.

인터페이스 VPC 엔드포인트 생성

1. VPC 콘솔을 엽니다. 왼쪽 메뉴의 Virtual Private Cloud에서 엔드포인트를 선택합니다.
2. Create endpoint(엔드포인트 생성)을 선택합니다. 다음 구성으로 엔드포인트를 생성합니다.
   - 이름 태그에 `lambda-default-vpc`를 입력합니다.
   - 서비스 범주에서 AWS 서비스를 선택합니다.
   - 서비스 검색 상자에 `lambda`를 입력합니다. `com.amazonaws.<region>.lambda` 형식의 서비스를 선택합니다.
   - VPC에서 기본 VPC를 선택합니다.
   - 서브넷에서 각 가용 영역 옆의 확인란을 선택합니다. 각 가용 영역에 대한 올바른 서브넷 ID를 선택합니다.
   - IP 주소 유형에서 IPv4를 선택합니다.
   - 보안 그룹에서 기본 VPC 보안 그룹(그룹 이름 `default`)과 이전에 생성한 보안 그룹(그룹 이름 `DocDBTutorial`)을 선택합니다.
   - 다른 기본 설정을 모두 유지합니다.
   - Create endpoint(엔드포인트 생성)을 선택합니다.
3. 다시 엔드포인트 생성을 선택합니다. 다음 구성으로 엔드포인트를 생성합니다.
   - 이름 태그에 `secretsmanager-default-vpc`를 입력합니다.
   - 서비스 범주에서 AWS 서비스를 선택합니다.
   - 서비스 검색 상자에 `secretsmanager`를 입력합니다. `com.amazonaws.<region>.secretsmanager` 형식의 서비스를 선택합니다.
   - VPC에서 기본 VPC를 선택합니다.
   - 서브넷에서 각 가용 영역 옆의 확인란을 선택합니다. 각 가용 영역에 대한 올바른 서브넷 ID를 선택합니다.
   - IP 주소 유형에서 IPv4를 선택합니다.
   - 보안 그룹에서 기본 VPC 보안 그룹(그룹 이름 `default`)과 이전에 생성한 보안 그룹(그룹 이름 `DocDBTutorial`)을 선택합니다.
   - 다른 기본 설정을 모두 유지합니다.
   - Create endpoint(엔드포인트 생성)을 선택합니다.

이것으로 이 자습서의 클러스터 설정 부분이 완료되었습니다.

실행 역할 생성



다음 단계에서는 Lambda 함수를 생성합니다. 먼저 함수에 클러스터에 액세스할 수 있는 권한을 제공하는 실행 역할을 생성해야 합니다. 이 작업을 수행하려면 먼저 IAM 정책을 생성한 다음 이 정책을 IAM 역할에 연결합니다.

IAM 정책 생성

1. IAM 콘솔에서 정책 페이지를 열고 정책 생성을 선택합니다.
2. JSON 탭을 선택합니다. 다음 정책에서 명령문의 마지막 줄에 있는 Secrets Manager 리소스 ARN을 이전의 보안 암호 ARN으로 바꾸고 정책을 편집기에 복사합니다.

```
{
    "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. 다음: 태그를 선택하고 다음: 검토를 선택합니다.
4. 이름에서 `AWSDocumentDBLambdaPolicy`을 입력합니다.
5. Create policy(정책 생성)를 선택합니다.

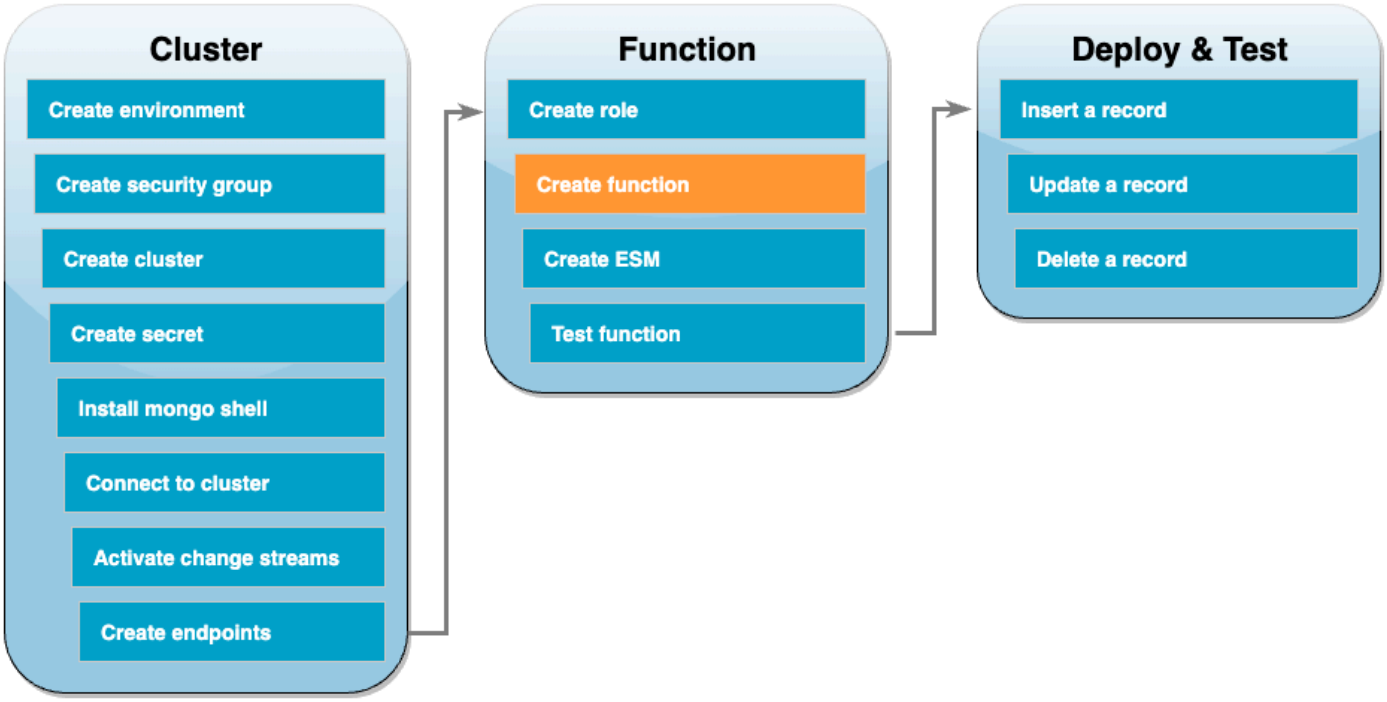
IAM 역할을 생성하려면

1. IAM 콘솔에서 역할 페이지를 열고 역할 생성을 선택합니다.
2. 신뢰할 수 있는 엔터티 선택에서 다음 옵션을 선택합니다.
   - 신뢰할 수 있는 엔터티 유형 – AWS 서비스
   - 사용 사례 – Lambda

- Next(다음)를 선택합니다.

3. 권한 추가에서 방금 생성한 `AWSDocumentDBLambdaPolicy` 정책과 `AWSLambdaBasicExecutionRole`을 선택하여 함수에 Amazon CloudWatch Logs에 쓸 수 있는 권한을 부여합니다.
4. Next(다음)를 선택합니다.
5. [역할 이름(Role name)]에 `AWSDocumentDBLambdaExecutionRole`을 입력합니다.
6. 역할 생성을 선택합니다.

Lambda 함수 생성



다음은 DocumentDB 이벤트 입력을 수신하고 이벤트에 포함된 메시지를 처리하는 예제 코드입니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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 함수를 생성하려면

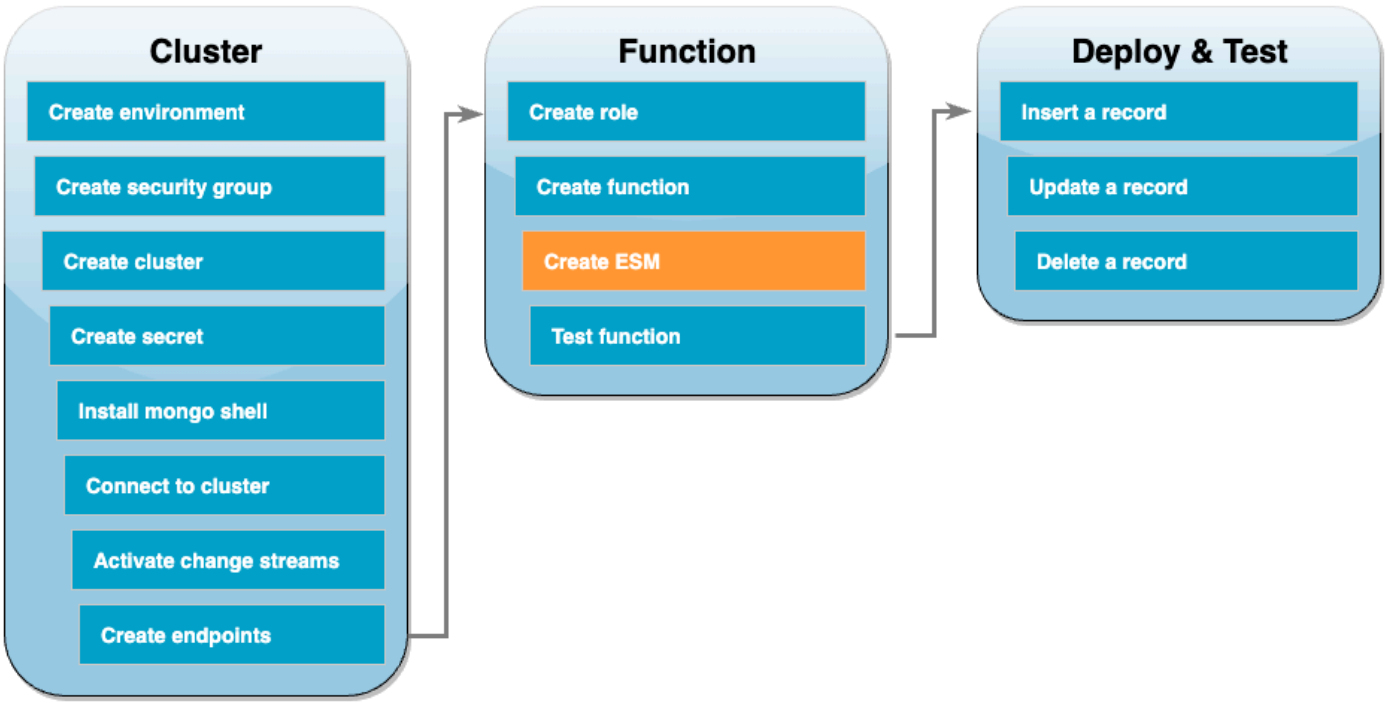
1. 샘플 코드를 `index.js` 파일에 복사합니다.
2. 다음 명령으로 배포 패키지를 생성합니다.

```
zip function.zip index.js
```

3. 다음 CLI 명령을 사용하여 함수를 생성합니다. `us-east-1`을 리전으로 바꾸고 `123456789012`를 계정 ID로 바꿉니다.

```
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 이벤트 소스 매핑 생성



DocumentDB 변경 스트림을 Lambda 함수와 연결하는 이벤트 소스 매핑을 생성합니다. 이 이벤트 소스 매핑을 생성하면 AWS Lambda가 스트림 폴링을 즉시 시작합니다.

이벤트 소스 매핑 생성

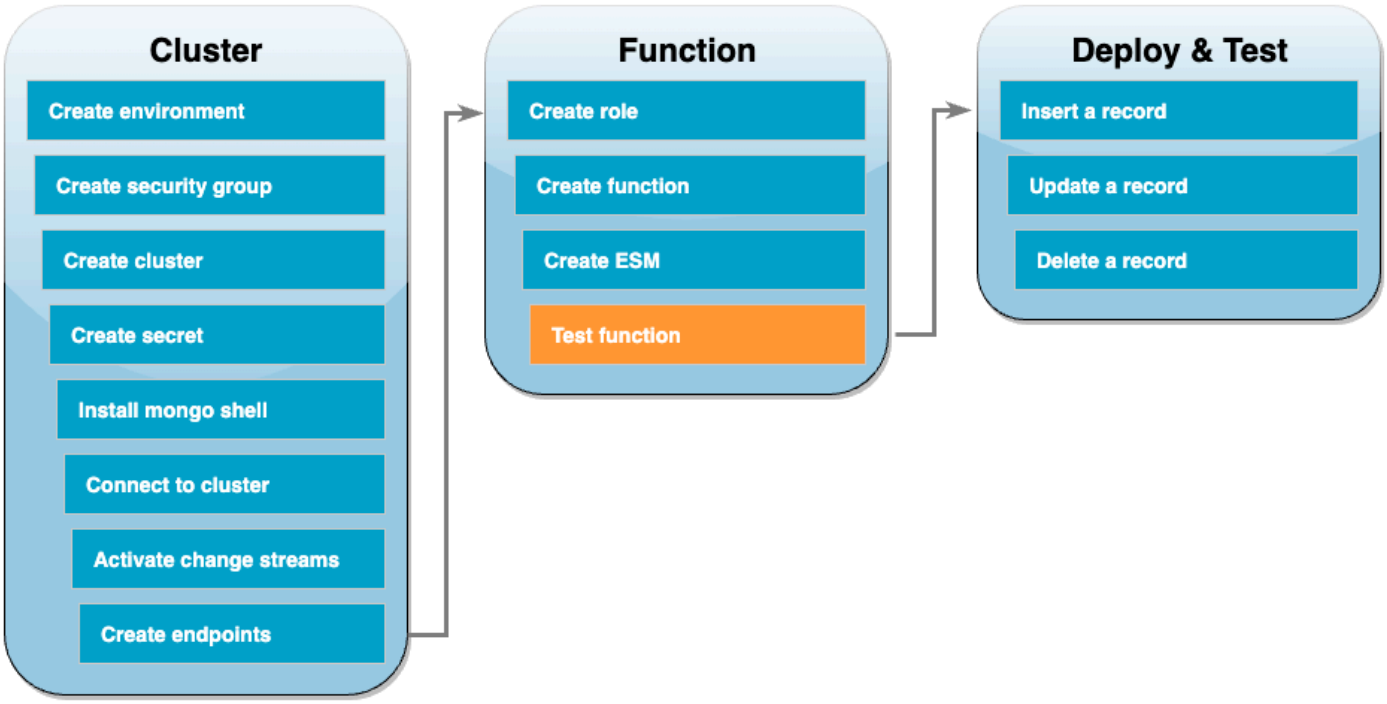
1. Lambda 콘솔에서 함수 페이지를 엽니다.
2. 앞에서 생성한 `ProcessDocumentDBRecords` 함수를 선택합니다.
3. 구성 탭을 선택한 다음 왼쪽 메뉴에서 트리거를 선택합니다.
4. 트리거 추가를 선택합니다.
5. 트리거 구성에서 소스로 DocumentDB를 선택합니다.
6. 다음 구성으로 이벤트 소스 매핑을 생성합니다.

   - DocumentDB 클러스터 - 이전에 생성한 클러스터를 선택합니다.
   - 데이터베이스 이름 – `docdbdemo`
   - 모음 이름 - 제품
   - 배치 크기 – 1
   - 시작 위치 - 최신
   - 인증 – BASIC_AUTH

- Secrets Manager 키 - 방금 생성한 `DocumentDBSecret`을 선택합니다.
- 배치 기간 – 1
- 전체 문서 구성 - UpdateLookup

7. 추가를 선택합니다. 이벤트 소스 매핑을 생성하는 데 몇 분 정도 걸릴 수 있습니다.

함수 테스트 - 수동 호출



함수와 이벤트 소스 매핑을 올바르게 생성했는지 테스트하려면 `invoke` 명령을 사용하여 함수를 호출합니다. 이렇게 하려면 먼저 다음 이벤트 JSON을 `input.txt`라는 파일에 복사합니다.

```
{
  "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"
}
```

그리고 다음 명령을 사용하여 이 이벤트와 함께 함수를 호출합니다.

```
aws lambda invoke --function-name ProcessDocumentDBRecords \
    --cli-binary-format raw-in-base64-out \
    --region us-east-1 \
    --payload file://input.txt out.txt
```

다음과 같은 응답이 표시됩니다.

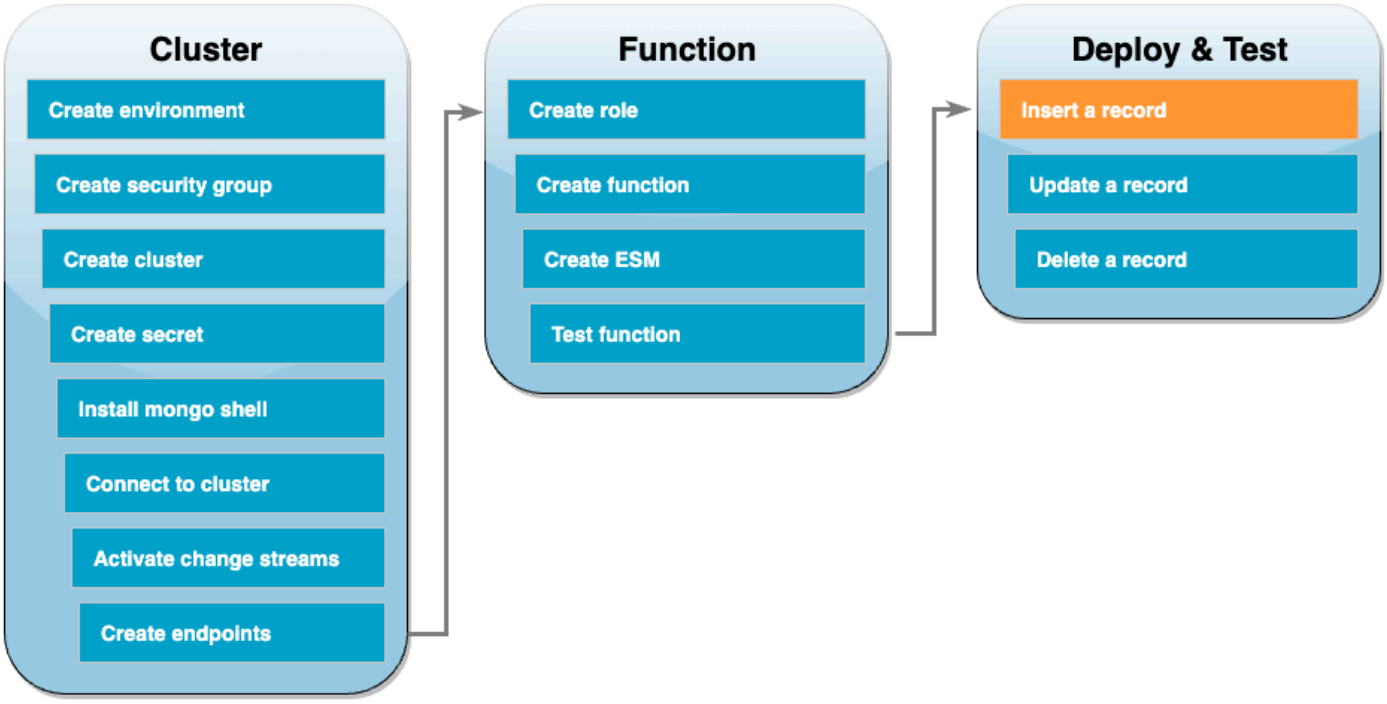
```
{
   "StatusCode": 200,
   "ExecutedVersion": "$LATEST"
}
```

CloudWatch Logs를 확인하여 함수가 이벤트를 성공적으로 처리했는지 확인할 수 있습니다.

CloudWatch Logs를 통해 수동 호출 확인

1. Lambda 콘솔에서 함수 페이지를 엽니다.
2. 모니터링 탭을 선택하고 CloudWatch 로그 보기를 선택합니다. 그러면 CloudWatch 콘솔에서 함수와 연결된 특정 로그 그룹으로 이동합니다.
3. 최신 로그 스트림을 선택합니다. 로그 메시지 내에 JSON 이벤트가 표시됩니다.

함수 테스트 - 레코드 삽입



DocumentDB 데이터베이스와 직접 상호 작용하여 엔드-투-엔드 설정을 테스트합니다. 다음 단계에서는 레코드를 삽입하고 업데이트한 다음 삭제합니다.

레코드 삽입

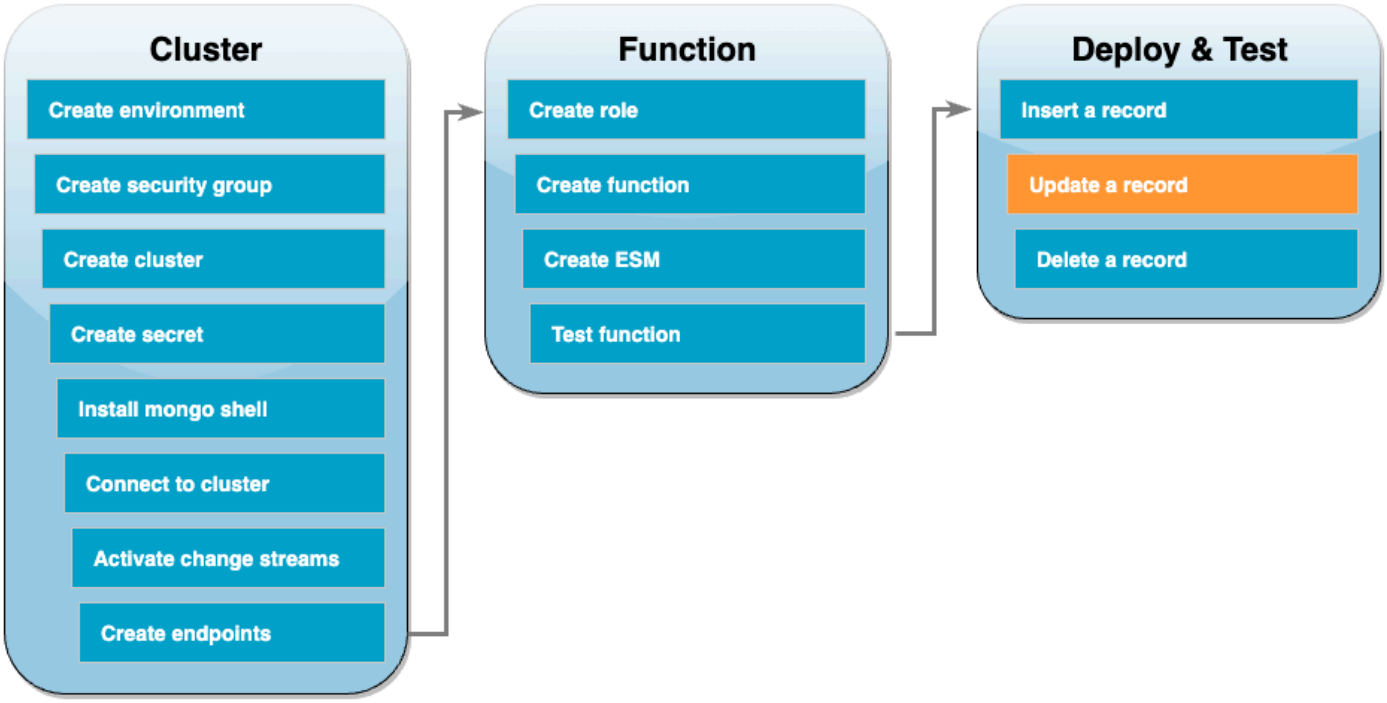
1. Cloud9 환경에서 DocumentDB 클러스터에 다시 연결합니다.
2. 이 명령을 사용하여 현재 `docdbdemo` 데이터베이스를 사용하고 있는지 확인합니다.

```
use docdbdemo
```

3. `docdbdemo` 데이터베이스의 `products` 컬렉션에 레코드 삽입:

```
db.products.insert({"name":"Pencil", "price": 1.00})
```

함수 테스트 - 레코드 업데이트

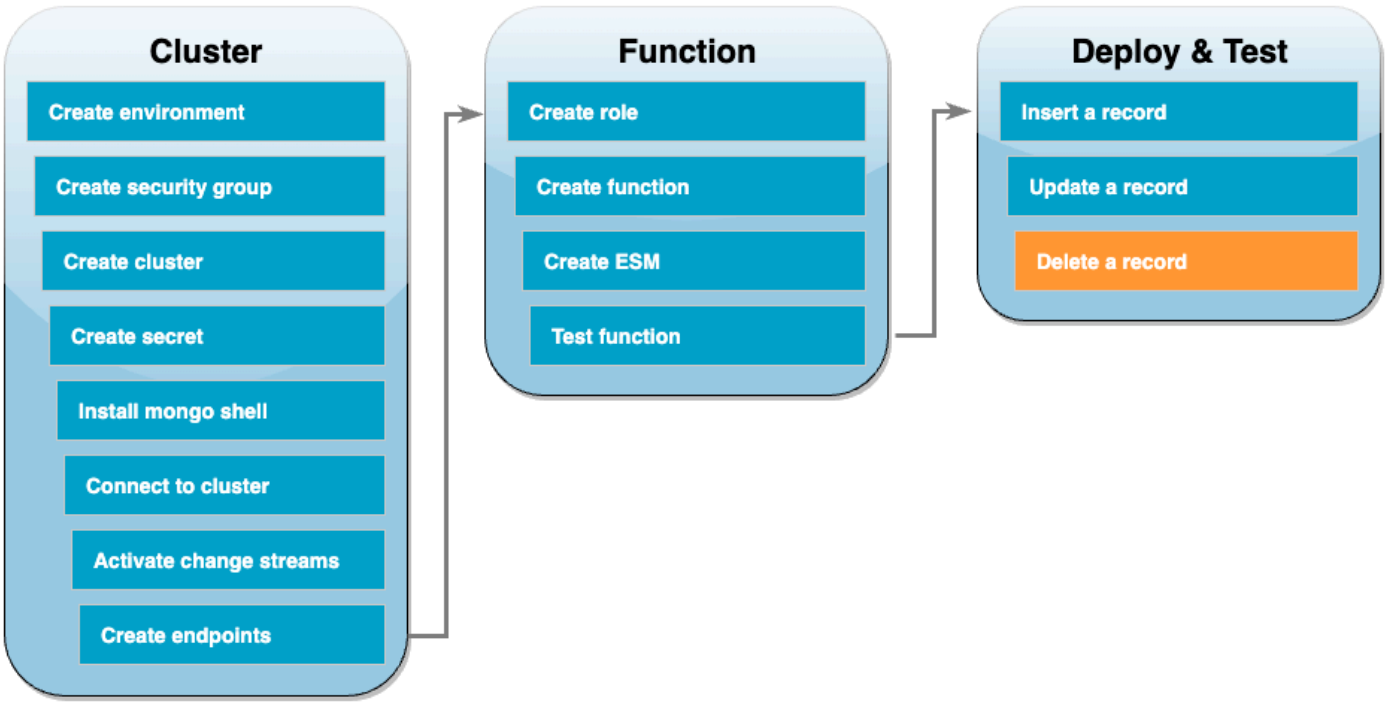


그리고 다음 명령을 사용하여 방금 삽입한 레코드를 업데이트합니다.

```
db.products.update(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

CloudWatch Logs를 확인하여 함수가 이 이벤트를 성공적으로 처리했는지 확인합니다.

함수 테스트 - 레코드 삭제



그리고 다음 명령을 사용하여 방금 업데이트한 레코드를 삭제합니다.

```
db.products.remove( { "name": "Pencil" } )
```

CloudWatch Logs를 확인하여 함수가 이 이벤트를 성공적으로 처리했는지 확인합니다.

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

VPC 엔드포인트 삭제

1. VPC 콘솔을 엽니다. 왼쪽 메뉴의 Virtual Private Cloud에서 엔드포인트를 선택합니다.
2. 생성한 엔드포인트를 선택합니다.
3. 작업(Actions), VPC 엔드포인트 삭제(Delete VPC endpoints)를 차례로 선택합니다.
4. 텍스트 입력 필드에 **`delete`**을 입력합니다.
5. Delete(삭제)를 선택합니다.

Amazon DocumentDB 클러스터 삭제

1. DocumentDB 콘솔을 엽니다.
2. 이 자습서용으로 생성한 DocumentDB 클러스터를 선택하고 삭제 방지를 비활성화합니다.
3. 기본 클러스터 페이지에서 DocumentDB 클러스터를 다시 선택합니다.
4. 작업, 삭제를 선택합니다.
5. 최종 클러스터 스냅샷 생성에서 아니요를 선택합니다.
6. 텍스트 입력 필드에 **`delete`**을 입력합니다.
7. Delete(삭제)를 선택합니다.

Secrets Manager에서 보안 암호 삭제

1. Secrets Manager 콘솔을 엽니다.
2. 이 자습서용으로 생성한 보안 암호를 선택합니다.
3. 작업, 보안 암호 삭제를 선택합니다.
4. 삭제 예약(Schedule deletion)을 선택합니다.

Amazon EC2 보안 그룹 삭제

1. EC2 콘솔을 엽니다. 네트워크 및 보안에서 보안 그룹을 선택합니다.
2. 이 자습서용으로 생성한 보안 그룹을 선택합니다.
3. 작업, 보안 그룹 삭제를 선택합니다.
4. Delete(삭제)를 선택합니다.

Cloud9 환경 삭제

1. Cloud9 콘솔을 엽니다.
2. 이 자습서용으로 생성한 환경을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**을 입력합니다.
5. Delete(삭제)를 선택합니다.

Lambda 이벤트 필터링

이벤트 필터링을 사용하여 Lambda가 함수로 전송하는 스트림 또는 대기열의 레코드를 제어할 수 있습니다. 예를 들어 함수에서 특정 데이터 파라미터가 포함된 Amazon SQS 메시지만 처리하도록 필터를 추가할 수 있습니다. 이벤트 필터링은 이벤트 소스 매핑과 함께 작동합니다. 다음 AWS 서비스에 대한 이벤트 소스 매핑에 필터를 추가할 수 있습니다.

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka(Amazon MSK)
- 자체 관리형 Apache Kafka
- Amazon Simple Queue Service(Amazon SQS)

Lambda는 Amazon DocumentDB에 대한 이벤트 필터링을 지원하지 않습니다.

기본적으로 단일 이벤트 소스 매핑에 대해 최대 5개의 필터를 정의할 수 있습니다. 필터는 논리적으로 OR로 결합됩니다. 이벤트 소스의 레코드가 하나 이상의 필터를 충족하는 경우 Lambda는 함수로 전송하는 다음 이벤트에 해당 레코드를 포함합니다. 충족되는 필터가 없으면 Lambda는 레코드를 삭제합니다.

> **Note**
>
> 이벤트 소스에 대해 5개가 넘는 필터를 정의해야 하는 경우 각 이벤트 소스에 대해 최대 필터 10개까지 할당량 증가를 요청할 수 있습니다. 현재 할당량이 허용하는 것보다 많은 필터를 추가하려고 하면 이벤트 소스를 생성하려고 할 때 Lambda에서 오류를 반환합니다.

주제

이벤트 필터링 기본 사항

필터 기준(`FilterCriteria`) 객체는 필터(`Filters`) 목록으로 구성된 구조입니다. 각 필터는 이벤트 필터링 패턴(`Pattern`)을 정의하는 구조입니다. 패턴은 JSON 필터 규칙의 문자열 표현입니다. `FilterCriteria` 객체의 구조는 다음과 같습니다.

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 Pattern 값은 다음과 같습니다.

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

필터 패턴에는 메타데이터 속성, 데이터 속성 또는 둘 다 포함될 수 있습니다. 사용 가능한 메타데이터 파라미터와 데이터 파라미터의 형식은 이벤트 소스 역할을 하는 AWS 서비스에 따라 다릅니다. 예를 들어, 이벤트 소스 매핑이 Amazon SQS 대기열에서 다음 레코드를 수신한다고 가정해 보겠습니다.

```
{
    "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"
}
```

- 메타데이터 속성은 레코드를 생성한 이벤트에 대한 정보가 들어 있는 필드입니다. 예제 Amazon SQS 레코드에서 메타데이터 속성에는 messageID, eventSourceArn, awsRegion 등의 필드가 포함됩니다.
- 데이터 속성은 스트림 또는 대기열의 데이터를 포함하는 레코드의 필드입니다. Amazon SQS 이벤트 예제에서 데이터 필드의 키는 body이고, 데이터 속성은 필드 City State 및 Temperature입니다.

이벤트 소스 유형마다 데이터 필드에 서로 다른 키 값을 사용합니다. 데이터 속성을 필터링하려면 필터의 패턴에 올바른 키를 사용해야 합니다. 데이터 필터링 키 목록과 지원되는 각 AWS 서비스의 필터 패턴 예제는 서로 다른 AWS 서비스에서 필터 사용 섹션을 참조하세요.

이벤트 필터링은 멀티 레벨 JSON 필터링을 처리할 수 있습니다. 예를 들어 DynamoDB 스트림의 다음 레코드 조각을 고려하세요.

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

정렬 키 `Number`의 값이 4567인 레코드만 처리하려고 한다고 가정해 보겠습니다. 이러한 경우 `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

필터 기준을 충족하지 않는 레코드 처리

필터를 충족하지 않는 레코드의 처리 방식은 이벤트 소스에 따라 다릅니다.

- Amazon SQS의 경우 메시지가 필터 기준을 충족하지 못하는 경우 Lambda는 자동으로 대기열에서 메시지를 제거합니다. 이러한 메시지를 Amazon SQS에서 수동으로 삭제할 필요가 없습니다.
- Kinesis와 DynamoDB의 경우 필터 기준에서 레코드를 처리하면 스트림 반복기가 이 레코드를 너머 진행됩니다. 레코드가 필터 조건을 충족하지 않으면 이벤트 소스에서 수동으로 레코드를 삭제할 필요가 없습니다. 보존 기간이 지나면 Kinesis 및 DynamoDB는 이러한 이전 레코드를 자동으로 삭제합니다. 레코드를 더 빨리 삭제하려면 데이터 보존 기간 변경을 참조하세요.
- Amazon MSK, 자체 관리형 Apache Kafka 및 Amazon MQ 메시지의 경우 Lambda는 필터에 포함된 모든 필드와 일치하지 않는 메시지를 삭제합니다. 자체 관리형 Apache Kafka의 경우 Lambda는 함수를 성공적으로 호출한 후 일치하는 메시지와 일치하지 않는 메시지에 대한 오프셋을 커밋합니다. Amazon MQ의 경우 Lambda는 함수를 성공적으로 호출한 후 일치하는 메시지를 확인하고, 일치하지 않는 메시지를 필터링할 때 확인합니다.

필터 규칙 구문

필터 규칙의 경우 Lambda는 Amazon EventBridge 규칙을 지원하며 EventBridge와 동일한 구문을 사용합니다. 자세한 내용은 Amazon EventBridge 사용 설명서의 Amazon EventBridge 이벤트 패턴을 참조하세요.

다음은 Lambda 이벤트 필터링에 사용할 수 있는 모든 비교 연산자에 대한 요약입니다.

비교 연산자	예	규칙 구문
Null	UserID가 null임	"UserID": [null]
비어 있음	LastName이 비어 있음	"LastName": [""]
같음	이름이 “Alice”임	"Name": ["Alice"]
같음(대/소문자 무시)	이름이 “Alice”임	"Name": [{ "equals-ignore-case": "alice" }]
및	위치가 “New York”이고 요일이 “Monday”임	"Location": ["New York"], "Day": ["Monday"]
또는	PaymentType이 “Credit” 또는 “Debit”임	"PaymentType": ["Credit", "Debit"]

비교 연산자	예	규칙 구문
또는(여러 필드)	위치가 'New York'이고 요일이 'Monday'임	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
아님	날씨는 “Raining”이 아님	"Weather": [{ "anything-but": ["Raining"] }]
숫자(같음)	가격은 100임	"Price": [{ "numeric": ["=", 100] }]
숫자(범위)	가격이 10을 초과하고 20보다 작거나 같음	"Price": [{ "numeric": [">", 10, "<=", 20] }]
존재함	Productname이 있음	"ProductName": [{ "exists": true }]
존재하지 않음	Productname이 없음	"ProductName": [{ "exists": false }]
로 시작함	리전이 미국에 있음	"Region": [{"prefix": "us-" }]
다음으로 끝남	FileName은 .png 확장명으로 끝납니다.	"FileName": [{ "suffix": ".png" }]

> **Note**
>
> EventBridge와 마찬가지로 문자열의 경우 Lambda는 대소문자 변환이나 기타 문자열 정규화 없이 정확한 문자별 일치를 사용합니다. 숫자의 경우 Lambda는 문자열 표현도 사용합니다. 예를 들어 300, 300.0 및 3.0e2는 동일한 것으로 간주되지 않습니다.

Exists 연산자가 이벤트 소스 JSON의 리프 노드에서만 작동한다는 점에 유의하세요. 중간 노드와 일치하지 않습니다. 예를 들어 다음 JSON에서는 `"address"`가 중간 노드이므로 필터 패턴(`{ "person": { "address": [ { "exists": true } ] } }`")이 일치하는 항목을 찾지 못합니다.

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

이벤트 소스 매핑에 필터 기준 연결(콘솔)

다음 단계에 따라 Lambda 콘솔을 사용하여 필터 기준을 사용하여 새 이벤트 소스 매핑을 생성합니다.

필터 기준을 사용하여 새 이벤트 소스 매핑을 생성하려면(콘솔)

1. Lambda 콘솔의 함수(Functions) 페이지를 엽니다.
2. 이벤트 소스 매핑을 생성할 함수의 이름을 선택합니다.
3. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.
4. 트리거 구성(Trigger configuration)에서 이벤트 필터링을 지원하는 트리거 유형을 선택합니다. 지원되는 서비스 목록은 이 페이지의 시작 부분에 있는 목록을 참조하세요.
5. 추가 설정을 편집니다.
6. 필터 기준(Filter criteria)에서 추가(Add)를 선택한 다음, 필터를 정의하고 입력합니다. 예를 들면 다음과 같이 입력할 수 있습니다.

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   이는 필드 `Metadata`가 1 또는 2인 레코드만 처리하도록 Lambda에 지시합니다. 계속해서 추가를 선택하여 허용되는 최대 개수까지 필터를 더 추가할 수 있습니다.
7. 필터 추가를 마쳤으면 저장을 선택합니다.

콘솔을 사용하여 필터 기준을 입력할 때 필터 패턴만 입력하면 되며 `Pattern` 키나 이스케이프 따옴표를 제공할 필요가 없습니다. 앞의 지침의 6단계에서 `{ "Metadata" : [ 1, 2 ] }`는 다음 `FilterCriteria`에 해당합니다.

```
{
```

```
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

콘솔에서 이벤트 소스 매핑을 생성한 후 트리거 세부 정보에서 형식이 지정된 `FilterCriteria`를 확인할 수 있습니다. 콘솔을 사용하여 이벤트 필터를 생성하는 추가 예는 서로 다른 AWS 서비스에서 필터 사용 섹션을 참조하세요.

이벤트 소스 매핑에 필터 기준 연결(AWS CLI)

이벤트 소스 매핑에 다음 `FilterCriteria`가 포함되기를 원한다고 가정해 보겠습니다.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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 ]}"}]}'
```

이 create-event-source-mapping 명령은 지정된 `FilterCriteria`로 함수 `my-function`에 대한 새 Amazon SQS 이벤트 소스 매핑을 생성합니다.

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

이벤트 소스 매핑을 업데이트하려면 해당 UUID가 필요합니다. list-event-source-mappings 호출에서 UUID를 가져올 수 있습니다. Lambda는 또한 create-event-source-mapping 응답에서 UUID를 반환합니다.

이벤트 소스에서 필터 기준을 제거하려면 빈 `FilterCriteria` 개체로 다음 update-event-source-mapping 명령을 실행할 수 있습니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

AWS CLI를 사용하여 이벤트 필터를 생성하는 추가 예는 서로 다른 AWS 서비스에서 필터 사용 섹션을 참조하세요.

이벤트 소스 매핑에 필터 기준 연결(AWS SAM)

AWS SAM에서 다음 필터 기준을 사용하도록 이벤트 소스를 구성한다고 가정해 보겠습니다.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

이러한 필터 기준을 이벤트 소스 매핑에 추가하려면 다음 코드 조각을 이벤트 소스의 YAML 템플릿에 삽입합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

이벤트 소스 매핑을 위한 AWS SAM 템플릿을 생성하고 구성하는 방법에 대한 자세한 내용은 AWS SAM 개발자 가이드에서 EventSource 섹션을 참조하세요. AWS SAM 템플릿을 사용하여 이벤트 필터를 생성하는 추가 예는 서로 다른 AWS 서비스에서 필터 사용 섹션을 참조하세요.

서로 다른 AWS 서비스에서 필터 사용

이벤트 소스 유형마다 데이터 필드에 서로 다른 키 값을 사용합니다. 데이터 속성을 필터링하려면 필터의 패턴에 올바른 키를 사용해야 합니다. 다음 표에는 지원되는 각 AWS 서비스에 대한 필터링 키가 나와 있습니다.

AWS 서비스	키 필터링
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
자체 관리형 Apache Kafka	`value`
Amazon SQS	`body`

다음 섹션에서는 다양한 유형의 이벤트 소스에 대한 필터 패턴의 예제를 제공합니다. 또한 지원되는 수신 데이터 형식의 정의와 지원되는 각 서비스에 대한 필터 패턴 본문 형식을 제공합니다.

DynamoDB로 필터링

프라이머리 키 `CustomerName`과 `AccountManager` 및 `PaymentTerms` 속성이 있는 DynamoDB 테이블이 있다고 가정해 보겠습니다. 다음은 DynamoDB 테이블 스트림의 예제 레코드를 보여줍니다.

```
{
      "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 테이블의 키 및 속성 값을 기준으로 필터링하려면 레코드의 dynamodb 키를 사용하세요. 다음 섹션에는 다양한 필터 유형에 대한 예제가 나와 있습니다.

테이블 키로 필터링

프라이머리 키 `CustomerName`이 'AnyCompany Industries'인 레코드만 함수에서 처리하도록 하려는 경우를 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
                "S": [ "AnyCompany Industries" ]
                }
            }
        }
```

```
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" ] } } } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
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 SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

테이블 속성으로 필터링

DynamoDB에서는 `NewImage` 및 `OldImage` 키를 사용하여 속성 값을 필터링할 수도 있습니다. 최신 테이블 이미지의 `AccountManager` 속성이 'Pat Candella' 또는 'Shirley Rodriguez'인 레코드를 필터링하려고 한다고 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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
\" ] } } } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
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 SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

부울 표현식으로 필터링

부울 AND 표현식을 사용하여 필터를 생성할 수도 있습니다. 이러한 표현식에는 테이블의 키 및 속성 파라미터가 모두 포함될 수 있습니다. `AccountManager`의 `NewImage` 값이 'Pat Candella'이고 `OldImage` 값이 'Terry Whitlock'인 레코드를 필터링하려고 한다고 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
```

```
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" ] } } } }
 "}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
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 SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB 이벤트 필터링은 숫자 연산자(숫자 등호 및 숫자 범위)의 사용을 지원하지 않습니다. 테이블의 항목이 숫자로 저장되더라도 이러한 파라미터는 JSON 레코드 객체의 문자열로 변환됩니다.

DynamoDB에서 Exists 연산자 사용

DynamoDB의 JSON 이벤트 객체가 구조화되는 방식으로 인해 Exists 연산자를 사용하려면 특별한 주의가 필요합니다. Exists 연산자는 이벤트 JSON의 리프 노드에서만 작동하므로 필터 패턴이 Exists를 사용하여 중간 노드를 테스트하는 경우에는 작동하지 않습니다. 다음 DynamoDB 테이블 항목을 고려하세요.

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

"Organizations"가 포함된 이벤트를 테스트하는 다음과 같은 필터 패턴을 생성할 수 있습니다.

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

그러나 이 필터 패턴은 "Organizations"가 리프 노드가 아니므로 일치하는 항목을 반환하지 않습니다. 다음 예제에서는 Exists 연산자를 올바르게 사용하여 원하는 필터 패턴을 구성하는 방법을 보여줍니다.

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

DynamoDB 필터링을 위한 JSON 형식

DynamoDB 소스에서 이벤트를 올바르게 필터링하려면 데이터 필드와 데이터 필드(dynamodb)의 필터 기준이 모두 유효한 JSON 형식이어야 합니다. 두 필드 중 하나가 유효한 JSON 형식이 아닌 경우 Lambda는 해당 메시지를 삭제하거나 예외를 발생시킵니다. 다음 표에는 특정 동작이 요약되어 있습니다.

수신 데이터 형식	데이터 속성에 대한 필터 패턴 형식	결과적 작업
유효한 JSON	유효한 JSON	Lambda는 필터 기준에 따라 필터링합니다.
유효한 JSON	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.

수신 데이터 형식	데이터 속성에 대한 필터 패턴 형식	결과적 작업
유효한 JSON	JSON 외	이벤트 소스 매핑 생성 또는 업데이트 시 Lambda에서 예외가 발생합니다. 데이터 속성에 대한 필터 패턴은 유효한 JSON 형식이어야 합니다.
JSON 외	유효한 JSON	Lambda는 해당 레코드를 삭제합니다.
JSON 외	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
JSON 외	JSON 외	이벤트 소스 매핑 생성 또는 업데이트 시 Lambda에서 예외가 발생합니다. 데이터 속성에 대한 필터 패턴은 유효한 JSON 형식이어야 합니다.

Kinesis로 필터링

생산자가 JSON 형식의 데이터를 Kinesis 데이터 스트림에 넣고 있다고 가정해 보겠습니다. 예제 레코드는 다음과 같으며, JSON 데이터는 `data` 필드에서 Base64로 인코딩된 문자열로 변환됩니다.

```
{
    "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"
}
```

생산자가 스트림에 넣는 데이터가 유효한 JSON이면 이벤트 필터링을 사용하여 `data` 키로 레코드를 필터링할 수 있습니다. 생산자가 다음 JSON 형식으로 레코드를 Kinesis 스트림에 넣고 있다고 가정해 보겠습니다.

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

주문 유형이 'buy'인 레코드만 필터링하려면 `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" ] } } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

Kinesis 소스에서 이벤트를 올바르게 필터링하려면 데이터 필드와 데이터 필드의 필터 기준이 모두 유효한 JSON 형식이어야 합니다. 두 필드 중 하나가 유효한 JSON 형식이 아닌 경우 Lambda는 해당 메시지를 삭제하거나 예외를 발생시킵니다. 다음 표에는 특정 동작이 요약되어 있습니다.

수신 데이터 형식	데이터 속성에 대한 필터 패턴 형식	결과적 작업
유효한 JSON	유효한 JSON	Lambda는 필터 기준에 따라 필터링합니다.
유효한 JSON	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	JSON 외	이벤트 소스 매핑 생성 또는 업데이트 시 Lambda에서 예외가 발생합니다. 데이터 속성에 대한 필터 패턴은 유효한 JSON 형식이어야 합니다.
JSON 외	유효한 JSON	Lambda는 해당 레코드를 삭제합니다.
JSON 외	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
JSON 외	JSON 외	이벤트 소스 매핑 생성 또는 업데이트 시 Lambda에서 예외가 발생합니다. 데이터 속성에 대한 필터 패턴은 유효한 JSON 형식이어야 합니다.

Kinesis 집계 레코드 필터링

Kinesis를 사용하면 여러 레코드를 단일 Kinesis Data Streams 레코드로 집계하여 데이터 처리량을 늘릴 수 있습니다. Lambda는 Kinesis 향상된 팬아웃을 사용할 때 집계 레코드에만 필터 기준을 적용할 수 있습니다. 표준 Kinesis를 사용하여 집계 레코드를 필터링하는 것은 지원되지 않습니다. 향상된 팬아웃을 사용하는 경우 Lambda 함수의 트리거 역할을 하도록 Kinesis 전용 처리량 소비자를 구성합니다. 그런 다음 Lambda는 집계 레코드를 필터링하고 필터 기준을 충족하는 레코드만 전달합니다.

Kinesis 레코드 집계에 대해 자세히 알아보려면 Kinesis Producer Library(KPL) 주요 개념 페이지의 집계 섹션을 참조하세요. Kinesis 향상된 팬아웃과 함께 Lambda를 사용하는 방법에 대해 자세히 알아보려면 AWS 컴퓨팅 블로그의 Amazon Kinesis Data Streams 향상된 팬아웃 및 AWS Lambda로 실시간 스트림 처리 성능 향상을 참조하세요.

Amazon MQ로 필터링

Amazon MQ 메시지 대기열에 유효한 JSON 형식 또는 일반 문자열의 메시지가 포함되어 있다고 가정해 보겠습니다. 예제 레코드는 다음과 같으며, 데이터는 `data` 필드에서 Base64로 인코딩된 문자열로 변환됩니다.

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 및 Rabbit MQ 브로커 모두에 이벤트 필터링을 사용하여 `data` 키로 레코드를 필터링할 수 있습니다. Amazon MQ 대기열에 다음 JSON 형식의 메시지가 포함되어 있다고 가정해 보겠습니다.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

`timeout` 필드가 0보다 큰 레코드만 필터링하려는 경우 `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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 ] } ] } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

Amazon MQ를 사용하면 메시지가 일반 문자열인 레코드를 필터링할 수도 있습니다. 메시지가 'Result:'로 시작하는 레코드만 처리하려고 한다고 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "data": [
```

```
        {
        "prefix": "Result: "
        }
    ]
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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: \" } ] }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
```

```
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ 메시지는 UTF-8 인코딩 문자열이어야 하며, 일반 문자열이거나 JSON 형식이어야 합니다. 이는 Lambda가 필터 기준을 적용하기 전에 Amazon MQ 바이트 배열을 UTF-8로 디코딩하기 때문입니다. 메시지가 UTF-16 또는 ASCII와 같은 다른 인코딩을 사용하거나 메시지 형식이 `FilterCriteria` 형식과 일치하지 않는 경우 Lambda는 메타데이터 필터만 처리합니다. 다음 표에는 특정 동작이 요약되어 있습니다.

수신 메시지 형식	메시지 속성에 대한 필터 패턴 형식	결과적 작업
일반 문자열	일반 문자열	Lambda는 필터 기준에 따라 필터링합니다.
일반 문자열	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
일반 문자열	유효한 JSON	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	일반 문자열	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	유효한 JSON	Lambda는 필터 기준에 따라 필터링합니다.
UTF-8이 아닌 인코딩 문자열	JSON, 일반 문자열 또는 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.

Amazon MSK 및 자체 관리형 Apache Kafka로 필터링

생산자가 Amazon MSK 또는 자체 관리형 Apache Kafka 클러스터의 주제에 유효한 JSON 형식 또는 일반 문자열로 메시지를 작성한다고 가정해 보겠습니다. 예제 레코드는 다음과 같으며, 메시지는 `value` 필드에서 Base64로 인코딩된 문자열로 변환됩니다.

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Apache Kafka 생산자가 다음 JSON 형식으로 주제에 메시지를 작성하고 있다고 가정해 보겠습니다.

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

`value` 키를 사용하여 레코드를 필터링할 수 있습니다. `device_ID`가 문자 AB로 시작하는 레코드만 필터링하려고 한다고 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" } ] } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
```

```
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

Amazon MSK 및 자체 관리형 Apache Kafka를 사용하면 메시지가 일반 문자열인 레코드를 필터링할 수도 있습니다. 문자열이 'error'인 메시지를 무시하려고 한다고 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" ] } ] }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Amazon MSK 및 자체 관리형 Apache Kafka 메시지는 UTF-8 인코딩 문자열이어야 하며, 일반 문자열이거나 JSON 형식이어야 합니다. 이는 Lambda가 필터 기준을 적용하기 전에 Amazon MSK 바이트 배열을 UTF-8로 디코딩하기 때문입니다. 메시지가 UTF-16 또는 ASCII와 같은 다른 인코딩을 사용하거나 메시지 형식이 `FilterCriteria` 형식과 일치하지 않는 경우 Lambda는 메타데이터 필터만 처리합니다. 다음 표에는 특정 동작이 요약되어 있습니다.

수신 메시지 형식	메시지 속성에 대한 필터 패턴 형식	결과적 작업
일반 문자열	일반 문자열	Lambda는 필터 기준에 따라 필터링합니다.
일반 문자열	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.

수신 메시지 형식	메시지 속성에 대한 필터 패턴 형식	결과적 작업
일반 문자열	유효한 JSON	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	일반 문자열	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	유효한 JSON	Lambda는 필터 기준에 따라 필터링합니다.
UTF-8이 아닌 인코딩 문자열	JSON, 일반 문자열 또는 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.

Amazon SQS로 필터링

Amazon SQS 대기열에 다음 JSON 형식의 메시지가 포함되어 있다고 가정해 보겠습니다.

```
{
    "RecordNumber": 0000,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

이 대기열에 대한 예제 레코드는 다음과 같습니다.

```
{
    "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 메시지의 콘텐츠를 기반으로 필터링하려면 Amazon SQS 메시지 레코드의 body 키를 사용합니다. Amazon SQS 메시지의 RequestCode가 'BBBB'인 레코드만 처리하려고 한다고 가정해 보겠습니다. FilterCriteria 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 Pattern 값은 다음과 같습니다.

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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\" ] } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

함수가 `RecordNumber`가 9,999보다 큰 레코드만 처리하도록 하려는 경우를 가정해 보겠습니다. `FilterCriteria` 객체는 다음과 같습니다.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

명확성을 더하기 위해 일반 JSON으로 확장된 필터의 `Pattern` 값은 다음과 같습니다.

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

콘솔, AWS CLI 또는 AWS SAM 템플릿을 사용하여 필터를 추가할 수 있습니다.

Console

콘솔을 사용하여 이 필터를 추가하려면 이벤트 소스 매핑에 필터 기준 연결(콘솔)의 지침을 따르고 필터 기준에 대해 다음 문자열을 입력합니다.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

AWS Command Line Interface(AWS CLI)를 사용하여 이러한 필터 기준으로 새 이벤트 소스 매핑을 생성하려면 다음 명령을 실행합니다.

```
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 ] } ] } }"}]}'
```

이러한 필터 기준을 기존 이벤트 소스 매핑에 추가하려면 다음 명령을 실행합니다.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

AWS SAM을 사용하여 이 필터를 추가하려면 이벤트 소스의 YAML 템플릿에 다음 코드 조각을 추가합니다.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

Amazon SQS의 경우 메시지 본문은 임의의 문자열이 될 수 있습니다. 그러나 `FilterCriteria`에서 유효한 JSON 형식의 body를 기대하는 경우 문제가 될 수 있습니다. 반대 시나리오도 마찬가지입니다. 수신 메시지 본문이 JSON 형식이지만 필터 기준이 body를 일반 문자열로 예상하는 경우 의도하지 않은 동작이 발생할 수 있습니다.

이 문제를 방지하려면 `FilterCriteria`에서 본문의 형식이 대기열에서 수신하는 메시지의 body의 예상 형식과 일치하는지 확인합니다. 메시지를 필터링하기 전에 Lambda는 수신 메시지 본문의 형식과 body의 필터 패턴의 형식을 자동으로 평가합니다. 일치하지 않으면 Lambda는 메시지를 삭제합니다. 다음 표에는 이 평가가 요약되어 있습니다.

수신 메시지 **body** 형식	필터 패턴 **body** 형식	결과적 작업
일반 문자열	일반 문자열	Lambda는 필터 기준에 따라 필터링합니다.
일반 문자열	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
일반 문자열	유효한 JSON	Lambda가 메시지를 삭제합니다.
유효한 JSON	일반 문자열	Lambda가 메시지를 삭제합니다.
유효한 JSON	데이터 속성에 대한 필터 패턴 없음	Lambda는 필터 기준에 따라 (다른 메타데이터 속성에만 해당) 필터링합니다.
유효한 JSON	유효한 JSON	Lambda는 필터 기준에 따라 필터링합니다.

콘솔에서 Lambda 함수 테스트

테스트 이벤트로 함수를 호출하여 콘솔에서 Lambda 함수를 테스트할 수 있습니다. 테스트 이벤트는 함수에 대한 JSON 입력입니다. 함수에 입력이 필요하지 않은 경우 이벤트는 빈 문서(`{}`)가 될 수 있습니다.

콘솔에서 테스트를 실행하면 Lambda는 테스트 이벤트와 함께 동시에 함수를 호출합니다. 함수 런타임은 JSON을 객체로 변환하고 처리를 위해 코드의 핸들러 메서드로 전달합니다.

> **테스트 이벤트 생성**
>
> 콘솔에서 테스트하려면 먼저 프라이빗 또는 공유 가능한 테스트 이벤트를 생성해야 합니다.

테스트 이벤트로 함수 호출

함수 테스트

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 테스트하려는 함수의 이름을 선택합니다.
3. 테스트(Test) 탭을 선택합니다.
4. 테스트 이벤트에서 새 이벤트 생성 또는 저장된 이벤트 편집을 선택한 다음 사용하려는 저장된 이벤트를 선택합니다.
5. 선택 사항 - 이벤트 JSON용 템플릿을 선택합니다.
6. 테스트(Test)를 선택합니다.
7. 테스트 결과를 확인하려면 실행 결과(Execution result)에서 세부 정보(Details)를 확장합니다.

테스트 이벤트를 저장하지 않고 함수를 호출하려면 저장하기 전에 테스트(Test)를 선택합니다. 이렇게 하면 세션 기간 동안에만 Lambda가 보존하는 저장되지 않은 테스트 이벤트가 생성됩니다.

코드(Code) 탭에서 저장 및 미저장 상태의 테스트 이벤트에 액세스할 수도 있습니다. 이 탭에서 테스트(Test)를 선택한 다음 테스트 이벤트를 선택하면 됩니다.

프라이빗 테스트 이벤트 생성

프라이빗 테스트 이벤트는 이벤트 작성자만 사용할 수 있으며, 사용하기 위해 추가 권한이 필요하지 않습니다. 함수당 최대 10개의 프라이빗 테스트 이벤트를 생성하고 저장할 수 있습니다.

프라이빗 테스트 이벤트 생성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 테스트하려는 함수의 이름을 선택합니다.
3. 테스트(Test) 탭을 선택합니다.
4. 테스트 이벤트(Test event)에서 다음을 수행합니다.
   a. 템플릿(Template)을 선택합니다.
   b. 테스트의 이름(Name)을 입력합니다.
   c. 텍스트 입력 상자에 JSON 테스트 이벤트를 입력합니다.
   d. 이벤트 공유 설정(Event sharing settings)에서 프라이빗(Private)을 선택합니다.
5. 변경 사항 저장을 선택합니다.

코드(Code) 탭에서 새 테스트 이벤트를 생성할 수도 있습니다. 이 탭에서 테스트(Test), 테스트 이벤트 구성(Configure test event)을 차례로 선택하면 됩니다.

공유 가능한 테스트 이벤트 생성

공유 가능한 테스트 이벤트는 동일한 AWS 계정의 다른 사용자와 공유할 수 있는 테스트 이벤트입니다. 다른 사용자의 공유 가능한 테스트 이벤트를 편집하고 해당 이벤트로 자신의 함수를 호출할 수 있습니다.

Lambda는 공유 가능한 테스트 이벤트를 이름이 지정된 EventBridge Amazon CloudWatch (Events) 스키마 레지스트리에 스키마로 저장합니다. `lambda-testevent-schemas` Lambda는 이 레지스트리를 사용하여 사용자가 생성한 공유 가능한 테스트 이벤트를 저장하고 호출하므로, 이 레지스트리를 편집하거나 `lambda-testevent-schemas`라는 이름을 사용하여 레지스트리를 생성하지 않는 것이 좋습니다.

공유 가능한 테스트 이벤트를 보고, 공유하고, 편집하려면 다음 EventBridge (CloudWatch Events) 스키마 레지스트리 API 작업 모두에 대한 권한이 있어야 합니다.

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`

- schemas.DescribeSchema
- schemas.GetDiscoveredSchema
- schemas.ListSchemaVersions
- schemas.UpdateSchema

공유 가능한 테스트 이벤트에 대한 편집 내용을 저장하면 해당 이벤트를 덮어씁니다.

공유 가능한 테스트 이벤트를 만들거나, 편집하거나, 볼 수 없는 경우 계정에 이러한 작업에 필요한 권한이 있는지 확인합니다. 필요한 권한이 있지만 공유 가능한 테스트 이벤트에 여전히 액세스할 수 없는 경우 EventBridge (CloudWatch 이벤트) 레지스트리에 대한 액세스를 제한할 수 있는 리소스 기반 정책이 있는지 확인하세요.

공유 가능한 테스트 이벤트 생성

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 테스트하려는 함수의 이름을 선택합니다.
3. 테스트(Test) 탭을 선택합니다.
4. 테스트 이벤트(Test event)에서 다음을 수행합니다.
   a. 템플릿(Template)을 선택합니다.
   b. 테스트의 이름(Name)을 입력합니다.
   c. 텍스트 입력 상자에 JSON 테스트 이벤트를 입력합니다.
   d. 이벤트 공유 설정(Event sharing settings)에서 공유 가능(Shareable)을 선택합니다.
5. 변경 사항 저장을 선택합니다.

> AWS Serverless Application Model에서 공유 가능한 테스트 이벤트를 사용합니다.
>
> AWS SAM을 사용하여 공유 가능한 테스트 이벤트를 호출할 수 있습니다. AWS Serverless Application Model 개발자 안내서의 `sam remote test-event` 참조

공유 가능한 테스트 이벤트 스키마 삭제

공유 가능한 테스트 이벤트를 삭제하면 Lambda는 `lambda-testevent-schemas` 레지스트리에서 해당 이벤트를 제거합니다. 레지스트리에서 공유 가능한 마지막 테스트 이벤트를 제거하면 Lambda는 해당 레지스트리를 삭제합니다.

사용자가 함수를 삭제할 경우 Lambda는 연결된 공유 가능한 테스트 이벤트 스키마를 삭제하지 않습니다. EventBridge (CloudWatch 이벤트) 콘솔에서 이러한 리소스를 수동으로 정리해야 합니다.

Lambda 함수 상태

함수를 호출할 준비가 된 시기를 나타내기 위해 Lambda는 모든 함수에 대한 함수 구성에 상태 필드를 포함합니다. `State`는 함수를 성공적으로 호출할 수 있는지 여부를 포함하여 함수의 현재 상태에 대한 정보를 제공합니다. 함수 상태는 함수 호출의 동작 또는 함수가 코드를 실행하는 방법을 변경하지 않습니다. 함수 상태는 다음을 포함합니다.

- `Pending` - Lambda가 함수를 만든 후 상태를 보류 중으로 설정합니다. 보류 상태 동안 Lambda는 VPC 또는 EFS 리소스와 같은 함수에 대한 리소스를 생성하거나 구성하려고 시도합니다. Lambda는 보류 상태에서 함수를 호출하지 않습니다. 해당 함수에서 작동하는 호출 또는 기타 API 작업은 실패합니다.
- `Active` - Lambda가 리소스 구성 및 프로비저닝을 완료한 후에 함수가 활성 상태로 전환됩니다. 함수는 활성 상태에서만 성공적으로 호출될 수 있습니다.
- `Failed` - 리소스 구성 또는 프로비저닝에 오류가 발생했음을 나타냅니다.
- `Inactive` - Lambda가 함수에 대해 구성된 외부 리소스를 회수할 수 있을 만큼 충분히 유휴 상태이면 해당 함수는 비활성 상태가 됩니다. 비활성 상태인 함수를 호출하려고 하면 호출이 실패하고 함수 리소스가 다시 만들어질 때까지 Lambda가 해당 함수를 보류 상태로 설정합니다. Lambda가 리소스를 다시 생성하지 못하면 함수가 비활성 상태로 돌아갑니다. 함수가 비활성 상태에서 멈춘 경우 추가 문제 해결을 위해 함수의 `StatusCode` 및 `StatusCodeReason` 속성을 참조하십시오. 함수를 활성 상태로 복원하려면 오류를 해결하고 함수를 다시 배포해야 할 수 있습니다.

SDK 기반 자동화 워크플로를 사용하거나 Lambda의 서비스 API를 직접 호출하는 경우, 호출 전에 함수의 상태를 확인하여 활성 상태인지 확인합니다. 이 작업은 Lambda API 작업 GetFunction을 사용하거나 AWS SDK for Java 2.0을 사용해 웨이터를 구성하여 수행할 수 있습니다.

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

다음 결과가 표시됩니다.

```
[
 "Active",
 "Successful"
]
```

함수 생성이 대기 중인 동안에는 다음 작업이 실패합니다.

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

업데이트 중 함수 상태

Lambda는 `LastUpdateStatus` 속성의 업데이트를 진행 중인 함수에 대한 추가 컨텍스트를 제공하며, 여기엔 다음 상태가 포함됩니다.

- `InProgress` - 기존 함수에서 업데이트가 진행 중입니다. 함수 업데이트가 진행되는 동안 호출은 함수의 이전 코드 및 구성으로 이동합니다.
- `Successful` - 업데이트가 완료되었습니다. Lambda가 업데이트를 완료하면 추가 업데이트가 있을 때까지 설정된 상태로 유지됩니다.
- `Failed` - 함수를 업데이트하지 못했습니다. Lambda가 업데이트를 중단하고 함수의 이전 코드와 구성은 사용 가능한 상태로 유지됩니다.

Example

다음은 업데이트 중인 함수에 대한 `get-function-configuration`의 결과입니다.

```
{
    "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",
    ...
```

```
}
```

FunctionConfiguration에는 두 개의 다른 속성인 `LastUpdateStatusReason` 및 `LastUpdateStatusReasonCode`가 있습니다. 이는 업데이트 관련 문제를 해결하는 데 도움이 됩니다.

비동기 업데이트가 진행 중인 동안에는 다음 작업이 실패합니다.

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion
- TagResource

Lambda의 재시도 동작 이해

함수를 직접 호출하는 경우 함수 코드와 관련된 오류 처리 전략을 결정합니다. Lambda는 사용자를 대신하여 이러한 유형의 오류를 자동으로 재시도하지 않습니다. 재시도할 때에는 함수를 수동으로 다시 호출하고 실패한 이벤트를 대기열로 보내 디버깅하거나 오류를 무시할 수 있습니다. 함수의 코드는 완전히 또는 부분적으로 실행되었거나 전혀 실행되지 않았을 수 있습니다. 재시도하는 경우 함수의 코드가 중복 트랜잭션 또는 기타 원치 않는 부작용을 일으키지 않고도 동일한 이벤트를 여러 번 처리할 수 있게 해야 합니다.

함수를 직접 호출하는 경우 호출자의 재시도 동작과 요청이 수행되는 과정에서 만나게 되는 모든 서비스에 대해 잘 알고 있어야 합니다. 여기에는 다음 시나리오가 포함됩니다.

- 비동기식 호출 – Lambda는 함수 오류를 두 번 재시도합니다. 함수가 모든 수신 요청을 처리할 만큼 용량이 충분하지 않은 경우 이벤트는 함수로 전송될 때까지 몇 시간 또는 며칠 동안 대기열에서 대기할 수 있습니다. 성공적으로 처리되지 않은 이벤트를 캡처하도록 함수에 대해 배달 못한 편지 대기열을 구성할 수 있습니다. 자세한 내용은 비동기식 호출 섹션을 참조하세요.
- 이벤트 소스 매핑 – 스트림에서 읽기를 수행하는 이벤트 소스 매핑은 항목의 전체 배치를 재시도합니다. 반복되는 오류는 오류가 해결되거나 항목이 만료될 때까지 영향을 받은 샤드의 처리를 차단합니다. 중단된 샤드를 탐지하려면 반복기 수명 지표를 모니터링하면 됩니다.

  대기열에서 읽는 이벤트 소스 매핑의 경우 소스 대기열에서 가시성 제한 시간 및 리드라이브 정책을 구성하여 실패한 이벤트에 대한 재시도와 대상 간의 시간 길이를 결정합니다. 자세한 내용은 Lambda가 스트림 및 대기열 기반 이벤트 소스의 레코드를 처리하는 방법 및 다른 AWS 서비스의 이벤트로 Lambda 간접 호출의 서비스별 주제를 참조하세요.
- AWS 서비스 – AWS 서비스는 함수를 동기식 또는 비동기식으로 호출할 수 있습니다. 동기식 호출의 경우 서비스는 재시도 여부를 결정합니다. 예를 들어, Lambda 함수가 `TemporaryFailure` 응답 코드를 반환하면 Amazon S3 배치 작업이 작업을 다시 시도합니다. 업스트림 사용자 또는 클라이언트의 요청을 프록시하는 서비스는 재시도 전략을 갖고 있거나 오류 응답을 요청자에게 다시 릴레이할 수 있습니다. 예를 들어, API Gateway는 오류 응답을 요청자에게 항상 다시 릴레이합니다.

  비동기식 호출의 경우 동작은 함수를 동기식으로 호출할 때와 동일합니다. 자세한 내용은 다른 AWS 서비스의 이벤트로 Lambda 간접 호출의 서비스별 주제와 호출하는 서비스에 관한 설명서를 참조하세요.
- 기타 계정 및 클라이언트 – 다른 계정에 대한 액세스 권한을 부여할 때 리소스 기반 정책을 사용해 함수를 호출하도록 구성할 수 있는 서비스 또는 리소스를 제한할 수 있습니다. 함수가 오버로드되는 것을 방지하기 위해 Amazon API Gateway를 사용하여 함수 앞에 API 계층을 배치할 것을 고려하는 것이 좋습니다.

Lambda 애플리케이션에서 오류를 더 원활히 처리할 수 있도록 Lambda는 Amazon CloudWatch, AWS X-Ray와 같은 서비스와 통합됩니다. 로그, 지표, 경보 및 추적의 조합을 사용해 애플리케이션을 지원하는 함수 코드, API 또는 기타 리소스에서 문제를 신속하게 감지하고 식별할 수 있습니다. 자세한 내용은 Lambda 함수 모니터링 및 문제 해결 단원을 참조하십시오.

Lambda 재귀 루프 감지를 사용하여 무한 루프 방지

함수를 간접적으로 호출하는 동일한 서비스 또는 리소스로 출력하도록 Lambda 함수를 구성하면 무한 재귀 루프를 생성할 수 있습니다. 예를 들어 Lambda 함수는 Amazon Simple Queue Service(Amazon SQS) 대기열에 메시지를 작성한 다음 동일한 함수를 간접적으로 호출할 수 있습니다. 이 간접 호출로 인해 함수는 대기열에 다른 메시지를 작성하고 다시 함수를 호출합니다.

의도하지 않은 재귀 루프로 인해 예상치 못한 요금이 AWS 계정에 청구될 수 있습니다. 루프로 인해 Lambda가 규모를 조정하고 계정의 사용 가능한 모든 동시성을 사용할 수도 있습니다. 의도하지 않은 루프의 영향을 줄이기 위해 Lambda는 특정 유형의 재귀 루프가 발생한 직후 이를 감지할 수 있습니다. Lambda가 재귀 루프를 감지하면 함수 간접 호출을 중지하고 알려줍니다.

의도적으로 재귀 패턴 사용을 설계하는 경우 Lambda의 재귀 루프 감지를 해제하도록 요청할 수 있습니다. 이 변경을 요청하려면 AWS Support에 문의하세요.

> **Important**
>
> 의도적으로 Lambda 함수 사용을 설계하여 함수를 간접적으로 호출하는 동일한 AWS 리소스에 데이터를 다시 작성하는 경우 주의를 기울이고 적절한 가드레일을 구현하여 예기치 않은 요금이 AWS 계정에 청구되지 않도록 하세요. 재귀 간접 호출 패턴 사용에 대한 모범 사례에 대해 자세히 알아보려면 Serverless Land의 Recursive patterns that cause run-away Lambda functions를 참조하세요.

Sections

재귀 루프 감지에 대한 이해

Lambda의 재귀 루프 감지는 이벤트 추적을 통해 작동합니다. Lambda는 특정 이벤트가 발생할 때 함수 코드를 실행하는 이벤트 기반 컴퓨팅 서비스입니다. 예를 들어 항목이 Amazon SQS 대기열 또는 Amazon Simple Notification Service(SNS) 주제에 추가되는 경우를 들 수 있습니다. Lambda는 이벤트를 시스템 상태 변경에 대한 정보가 포함된 JSON 객체로 함수에 전달합니다. 이벤트로 인해 함수가 실행되는 경우를 간접 호출이라고 합니다.

재귀 루프를 감지하기 위해 Lambda는 AWS X-Ray 추적 헤더를 사용합니다. 재귀 루프 감지를 지원하는 AWS 서비스가 이벤트를 Lambda로 보내면 해당 이벤트에 자동으로 메타데이터 주석이 추가됩니다. Lambda 함수가 지원되는 AWS SDK 버전을 사용하여 이러한 이벤트 중 하나를 지원되는 다른 AWS 서비스에 쓸 때 이 메타데이터가 업데이트됩니다. 업데이트된 메타데이터에는 이벤트가 함수를 간접적으로 호출한 횟수가 포함됩니다.

> **Note**
>
> 이 기능을 작동하기 위해 X-Ray 활성 추적을 활성화할 필요는 없습니다. 재귀 루프 감지는 기본적으로 모든 AWS 고객에 대해 켜져 있습니다. 이 기능은 무료로 사용할 수 있습니다.

요청 체인은 동일한 트리거 이벤트로 인해 발생하는 일련의 Lambda 간접 호출입니다. 예를 들어 Amazon SQS 대기열이 Lambda 함수를 간접적으로 호출한다고 가정해 보겠습니다. 그러면 Lambda 함수가 처리된 이벤트를 동일한 Amazon SQS 대기열로 다시 전송하여 함수를 다시 간접적으로 호출합니다. 이 예제에서 함수의 각 간접 호출은 동일한 요청 체인에 속합니다.

함수가 동일한 요청 체인에서 16회 넘게 간접적으로 호출되면 Lambda는 해당 요청 체인에서 다음 함수의 간접 호출을 자동으로 중지하고 사용자에게 알립니다. 함수가 여러 트리거로 구성된 경우 다른 트리거에서의 간접 호출은 영향을 받지 않습니다.

> **Note**
>
> 소스 대기열의 리드라이브 정책에 대한 `maxReceiveCount` 설정이 16보다 높은 경우, Lambda 재귀 보호는 재귀 루프가 감지되고 종료된 후 Amazon SQS가 메시지를 재시도하는 것을 막지 않습니다. Lambda는 재귀 루프를 감지하고 후속 간접 호출을 삭제하면 `RecursiveInvocationException`을(를) 이벤트 소스 매핑으로 반환합니다. 이로 인해 메시지의 `receiveCount` 값이 증가합니다. Amazon SQS가 `maxReceiveCount`를 초과했다고 판단하여 구성된 DLQ(Dead Letter Queue)로 메시지를 보낼 때까지 Lambda는 메시지를 계속 재시도하고 함수 호출을 계속 차단합니다.

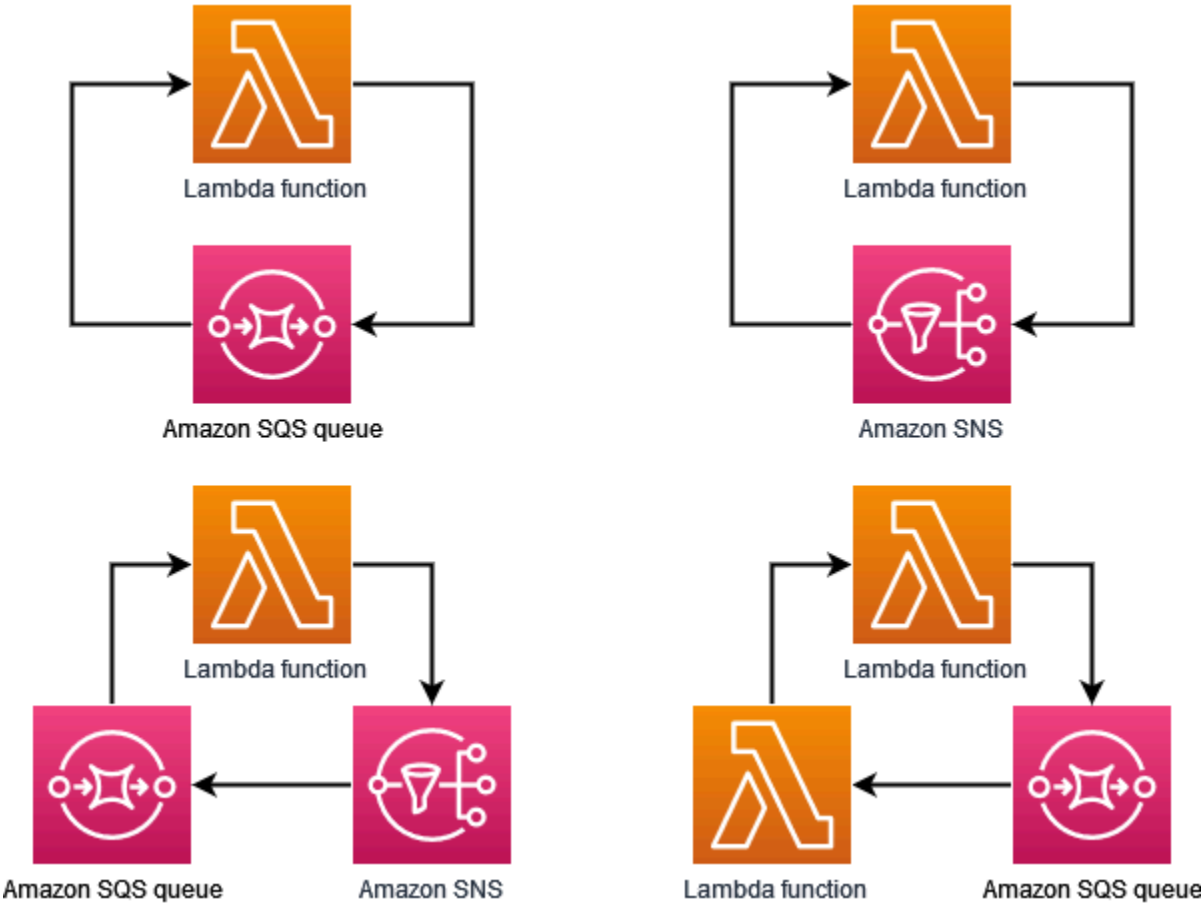
함수에 대해 실패 시 대상 또는 DLQ(Dead Letter Queue)가 구성되어 있는 경우에도, Lambda는 중지된 간접 호출에서 이벤트를 대상 또는 DLQ로 전송합니다. 함수에 대한 대상 또는 DLQ(Dead Letter Queue)를 구성하는 경우 함수가 이벤트 트리거 또는 이벤트 소스 매핑으로도 사용하는 Amazon SNS 주제 또는 Amazon SQS 대기열을 사용하지 마십시오. 함수를 간접적으로 호출하는 동일한 리소스에 이벤트를 보내면 또 다른 재귀 루프를 만들 수 있습니다.

지원되는 AWS 서비스 및 SDK

Lambda는 지원되는 특정 AWS 서비스를 포함하는 재귀 루프만 감지할 수 있습니다. 재귀 루프를 감지하려면 함수도 지원되는 AWS SDK 중 하나를 사용해야 합니다.

지원되는 AWS 서비스

Lambda는 현재 함수, Amazon SQS, Amazon SNS 간의 재귀 루프를 감지합니다. Lambda는 또한 동기식 또는 비동기식으로 서로를 간접적으로 호출할 수 있는 Lambda 함수로만 구성된 루프를 감지합니다. 다음 다이어그램은 Lambda가 감지할 수 있는 루프의 몇 가지 예를 보여줍니다.



Amazon DynamoDB 또는 Amazon Simple Storage Service(Amazon S3)와 같은 다른 AWS 서비스가 루프의 일부인 경우 현재 Lambda는 이를 감지하고 중지할 수 없습니다.

Lambda는 현재 Amazon SQS 및 Amazon SNS와 관련된 재귀 루프만 감지하기 때문에 다른 AWS 서비스와 관련된 루프로 인해 Lambda 함수가 의도치 않게 사용될 수 있습니다.

AWS 계정에 예기치 않은 요금이 청구되지 않도록 하려면 비정상적인 사용 패턴을 알리도록 Amazon CloudWatch 경보를 구성하는 것이 좋습니다. 예를 들어 Lambda 함수 동시성 또는 간접 호출의 급증에

대해 알리도록 CloudWatch를 구성할 수 있습니다. 계정 지출이 지정한 임계값을 초과할 때 알림을 받도록 결제 경보를 구성할 수도 있습니다. 또는 AWS Cost Anomaly Detection를 사용하여 비정상적인 청구 패턴에 대해 경고할 수 있습니다.

지원되는 AWS SDK

Lambda가 재귀 루프를 감지하려면 함수가 다음 이상의 SDK 버전 중 하나를 사용해야 합니다.

런타임	필요한 최소 AWS SDK 버전
Node.js	2.1147.0(SDK 버전 2) 3.105.0(SDK 버전 3)
Python	1.24.46(boto3) 1.27.46(botocore)
Java 8 및 Java 11	1.12.200(SDK 버전 1) 2.17.135(SDK 버전 2)
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0
Ruby	3.134.0
PHP	3.232.0

Python 및 Node.js와 같은 일부 Lambda 런타임에는 AWS SDK 버전이 포함되어 있습니다. 함수의 런타임에 포함된 SDK 버전이 필요한 최소 버전보다 낮은 경우 지원되는 SDK 버전을 함수의 배포 패키지에 추가할 수 있습니다. Lambda 계층을 사용하여 지원되는 SDK 버전을 함수에 추가할 수도 있습니다. 각 Lambda 런타임에 포함된 SDK 목록은 Lambda 런타임 단원을 참조하세요.

Lambda Go 런타임에는 Lambda 재귀 감지가 지원되지 않습니다.

재귀 루프 알림

Lambda가 재귀 루프를 중지하면 AWS Health Dashboard와 이메일을 통해 알림을 받습니다. 또한 CloudWatch 지표를 사용하여 Lambda가 중지한 재귀 간접 호출 수를 모니터링할 수 있습니다.

AWS Health Dashboard 알림

Lambda가 재귀 간접 호출을 중지하면 AWS Health Dashboard는 계정 상태 페이지의 미결 및 최근 문제 아래에 알림을 표시합니다. Lambda가 재귀 간접 호출을 중지한 후 이 알림이 표시되기까지 최대 3시간이 걸릴 수 있습니다. AWS Health Dashboard에서 계정 이벤트 보기에 대한 자세한 내용은 AWS Health 사용 설명서에서 Getting started with your AWS Health Dashboard – Your account health를 참조하세요.

이메일 알림

Lambda가 함수의 재귀 간접 호출을 처음 중지하면 이메일 알림이 전송됩니다. Lambda는 AWS 계정의 각 함수와 관련된 이메일을 24시간마다 최대 한 번 보냅니다. Lambda가 이메일 알림을 보낸 후 Lambda가 함수의 추가 재귀 간접 호출을 중지하더라도 그다음 24시간 동안 해당 함수에 대한 이메일이 더 이상 전송되지 않습니다. Lambda가 재귀 간접 호출을 중지한 후 이 이메일 알림을 받을 때까지 최대 3시간이 걸릴 수 있습니다.

Lambda는 AWS 계정의 기본 계정 연락처 및 대체 작업 연락처에 재귀 루프 이메일 알림을 보냅니다. 계정의 이메일 주소 보기 또는 업데이트에 대한 자세한 내용은 AWS 참조 가이드에서 Updating contact information을 참조하세요.

Amazon CloudWatch 지표

CloudWatch 지표 `RecursiveInvocationsDropped`는 단일 요청 체인에서 함수가 16회 넘게 간접적으로 호출되어 Lambda가 중지한 함수 간접 호출 수를 기록합니다. Lambda는 재귀 간접 호출을 중지하는 즉시 이 지표를 내보냅니다. 이 지표를 보려면 CloudWatch 콘솔에서 지표 보기 지침을 따르고 지표 `RecursiveInvocationsDropped`를 선택하세요.

재귀 루프 감지 알림에 응답

동일한 트리거 이벤트에 의해 함수가 16회 넘게 간접적으로 호출되면 Lambda는 해당 이벤트에 대한 다음 함수 간접 호출을 중지하여 재귀 루프를 중지합니다. Lambda가 중지한 재귀 루프의 재발을 방지하려면 다음을 수행합니다.

- 함수의 사용 가능한 동시성을 0으로 줄이면 향후 모든 간접 호출이 제한됩니다.

- 함수를 간접적으로 호출하는 트리거 또는 이벤트 소스 매핑을 제거하거나 비활성화합니다.
- 함수를 간접적으로 호출하는 AWS 리소스에 이벤트를 다시 기록하는 코드 결함을 식별하고 수정합니다. 일반적인 결함은 변수를 사용하여 함수의 이벤트 소스와 대상을 정의할 때 발생합니다. 두 변수에 동일한 값을 사용하고 있지 않은지 확인하세요.

또한 Lambda 함수의 이벤트 소스가 Amazon SQS 대기열인 경우 소스 대기열에서 DLQ(Dead Letter Queue)를 구성하는 것이 좋습니다.

> **Note**
>
> Lambda 함수가 아닌 소스 대기열에서 배달 못한 편지 대기열을 구성해야 합니다. 함수에서 구성하는 배달 못한 편지 대기열은 이벤트 소스 대기열이 아닌 함수의 비동기식 호출 대기열에 사용됩니다.

이벤트 소스가 Amazon SNS 주제인 경우 함수에 대해 실패 시 대상을 추가하는 것이 좋습니다.

함수의 사용 가능한 동시성을 0으로 줄이려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 제한을 선택합니다.
4. 함수 제한 대화 상자에서 확인을 선택합니다.

함수의 트리거 또는 이벤트 소스 매핑을 제거하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성 탭을 선택한 다음 트리거를 선택합니다.
4. 트리거에서 삭제하려는 트리거 또는 이벤트 소스 매핑을 선택한 다음 삭제를 선택합니다.
5. 트리거 삭제 대화 상자에서 삭제를 선택합니다.

함수에 대한 이벤트 소스 매핑을 비활성화하려면(AWS CLI)

1. 비활성화하려는 이벤트 소스 매핑의 UUID를 찾으려면 AWS Command Line Interface(AWS CLI) list-event-source-mappings 명령을 실행합니다.

```
aws lambda list-event-source-mappings
```

2. 이벤트 소스 매핑을 비활성화하려면 다음 AWS CLI update-event-source-mapping 명령을 실행합니다.

```
aws lambda update-event-source-mapping --function-name MyFunction \
--uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
```

Lambda 함수 URL

함수 URL은 Lambda 함수를 위한 전용 HTTP(S) 엔드포인트입니다. Lambda 콘솔 또는 Lambda API를 통해 함수 URL을 생성하고 구성할 수 있습니다. 함수 URL을 생성하면 Lambda가 자동으로 고유한 URL 엔드포인트를 생성합니다. 함수 URL을 생성하면 URL 엔드포인트가 변경되지 않습니다. 함수 URL 엔드포인트는 다음 형식을 취합니다.

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 함수 URL은 아시아 태평양 (하이데라바드) (), 아시아 태평양 (멜버른) (ap-south-2), 캐나다 서부 (캘거리ap-southeast-4) (), 유럽 (스페인) (ca-west-1), 유럽 (취리히) (eu-south-2), 이스라엘 (텔아비브) (eu-central-2), 중동 (UAE) (il-central-1) 지역에서 지원되지 않습니다. me-central-1

함수 URL은 IPv4 및 IPv6을 지원하는 이중 스택을 지원합니다. 함수에 대한 함수 URL을 구성한 후 웹 브라우저, curl, Postman 또는 모든 HTTP 클라이언트를 통해 HTTP(S) 엔드포인트에서 함수를 호출할 수 있습니다.

> **Note**
>
> 퍼블릭 인터넷을 통해서만 함수 URL에 액세스할 수 있습니다. Lambda 함수는 AWS PrivateLink를 지원하지만 함수 URL은 지원하지 않습니다.

Lambda 함수 URL은 보안 및 액세스 제어를 위해 리소스 기반 정책을 사용합니다. 함수 URL은 교차 오리진 리소스 공유(CORS) 구성 옵션도 지원합니다.

함수 URL을 함수 별칭이나 $LATEST 게시되지 않은 함수 버전에 적용할 수 있습니다. 다른 함수 버전에는 함수 URL을 추가할 수 없습니다.

주제

- Lambda 함수 URL 생성 및 관리
- Lambda 함수 URL에 대한 액세스 제어
- Lambda 함수 URL 호출

- Lambda 함수 URL 모니터링
- 자습서: 함수 URL을 사용하여 Lambda 함수 생성

Lambda 함수 URL 생성 및 관리

함수 URL은 Lambda 함수를 위한 전용 HTTP(S) 엔드포인트입니다. Lambda 콘솔 또는 Lambda API를 통해 함수 URL을 생성하고 구성할 수 있습니다. 함수 URL을 생성하면 Lambda가 자동으로 고유한 URL 엔드포인트를 생성합니다. 함수 URL을 생성하면 URL 엔드포인트가 변경되지 않습니다. 함수 URL 엔드포인트는 다음 형식을 취합니다.

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 아시아 태평양(하이데라바드)(ap-south-2), 아시아 태평양(멜버른)(ap-southeast-4), 캐나다 서부(캘거리)(ca-west-1), 유럽(스페인)(eu-south-2), 유럽(취리히)(eu-central-2), 이스라엘(텔아비브)(il-central-1) 및 중동(아랍에미리트)(me-central-1) 리전에서는 함수 URL이 지원되지 않습니다.

다음 섹션에서는 Lambda 콘솔, AWS CLI 및 AWS CloudFormation 템플릿을 사용하여 함수 URL을 생성하고 관리하는 방법을 보여줍니다.

주제

함수 URL 생성(콘솔)

다음 단계에 따라 콘솔을 사용하여 함수 URL을 생성합니다.

기존 함수에 대한 함수 URL을 생성하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.

2. 함수 URL을 생성할 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 함수 URL(Function URL)을 선택합니다.
4. 함수 URL 생성(Create function URL)을 선택합니다.
5. 인증 유형(Auth type)에서 AWS_IAM 또는 NONE을 선택합니다. 함수 URL 인증에 대한 자세한 내용은 액세스 제어 섹션을 참조하세요.
6. (선택 사항) 교차 오리진 리소스 공유(CORS) 구성(Configure cross-origin resource sharing (CORS))을 선택한 다음 함수 URL에 대한 CORS 설정을 구성합니다. CORS에 대한 자세한 내용은 교차 오리진 리소스 공유(CORS) 섹션을 참조하세요.
7. 저장(Save)을 선택합니다.

이렇게 하면 $LATEST 게시되지 않은 함수 버전에 대한 함수 URL이 생성됩니다. 함수 URL이 콘솔의 함수 개요(Function overview) 섹션에 표시됩니다.

기존 별칭에 대한 함수 URL을 생성하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 URL을 생성할 별칭을 사용하는 함수의 이름을 선택합니다.
3. 별칭(Aliases) 탭을 선택한 다음, 함수 URL을 생성할 별칭의 이름을 선택합니다.
4. 구성(Configuration) 탭을 선택한 다음, 함수 URL(Function URL)을 선택합니다.
5. 함수 URL 생성(Create function URL)을 선택합니다.
6. 인증 유형(Auth type)에서 AWS_IAM 또는 NONE을 선택합니다. 함수 URL 인증에 대한 자세한 내용은 액세스 제어 섹션을 참조하세요.
7. (선택 사항) 교차 오리진 리소스 공유(CORS) 구성(Configure cross-origin resource sharing (CORS))을 선택한 다음 함수 URL에 대한 CORS 설정을 구성합니다. CORS에 대한 자세한 내용은 교차 오리진 리소스 공유(CORS) 섹션을 참조하세요.
8. Save(저장)를 선택합니다.

이렇게 하면 함수 별칭에 대한 함수 URL이 생성됩니다. 함수 URL이 콘솔의 별칭에 대한 함수 개요(Function overview) 섹션에 표시됩니다.

함수 URL을 사용하여 새 함수를 생성하려면(콘솔)

함수 URL을 사용하여 새 함수를 생성하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.

2. 함수 생성을 선택합니다.
3. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름(Function name)에 **my-function**과 같은 함수 이름을 입력합니다.
   b. 런타임(Runtime)에서 원하는 언어 런타임을 선택합니다(예: Node.js 18.x).
   c. 아키텍처(Architecture)에서 x86_64 또는 arm64를 선택합니다.
   d. 권한(Permissions)을 확장한 다음, 새 실행 역할을 생성할지 아니면 기존 역할을 사용할지 여부를 선택합니다.
4. 고급 설정(Advanced settings)을 확장한 다음 함수 URL(Function URL)을 선택합니다.
5. 인증 유형(Auth type)에서 AWS_IAM 또는 NONE을 선택합니다. 함수 URL 인증에 대한 자세한 내용은 액세스 제어 섹션을 참조하세요.
6. (선택 사항) 교차 오리진 리소스 공유(CORS) 구성(Configure cross-origin resource sharing (CORS))을 선택합니다. 함수 생성 중에 이 옵션을 선택하면 함수 URL이 기본적으로 모든 오리진의 요청을 허용합니다. 함수를 생성한 후 함수 URL에 대한 CORS 설정을 편집할 수 있습니다. CORS에 대한 자세한 내용은 교차 오리진 리소스 공유(CORS) 섹션을 참조하세요.
7. 함수 생성을 선택합니다.

이렇게 하면 $LATEST 게시되지 않은 함수 버전에 대한 함수 URL이 있는 새로운 함수가 생성됩니다. 함수 URL이 콘솔의 함수 개요(Function overview) 섹션에 표시됩니다.

함수 URL 생성(AWS CLI)

AWS Command Line Interface(AWS CLI)를 사용하여 기존 Lambda 함수에 대한 함수 URL을 생성하려면 다음 명령을 실행합니다.

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

이렇게 하면 함수 **my-function**에 대한 **prod** 한정자에 함수 URL이 추가됩니다. 이러한 구성 파라미터에 대한 자세한 내용은 API 참조의 CreateFunctionUrlConfig를 참조하세요.

> **Note**
>
> AWS CLI를 통해 함수 URL을 생성하려면 함수가 이미 존재해야 합니다.

CloudFormation 템플릿에 함수 URL 추가

AWS CloudFormation 템플릿에 `AWS::Lambda::Url` 리소스를 추가하려면 다음 구문을 사용합니다.

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
```

파라미터

- (필수) `AuthType` – 함수 URL에 대한 인증 유형을 정의합니다. 가능한 값은 `AWS_IAM` 또는 `NONE`입니다. 액세스 권한을 인증된 사용자로 제한하려면 `AWS_IAM`으로 설정합니다. IAM 인증을 우회하고 모든 사용자가 함수에 요청을 수행할 수 있도록 허용하려면 `NONE`으로 설정합니다.
- (선택 사항) `Cors` – 함수 URL에 대한 CORS 설정을 정의합니다. CloudFormation의 `AWS::Lambda::Url` 리소스에 `Cors`를 추가하려면 다음 구문을 사용합니다.

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
```

- (선택 사항) `Qualifier` – 별칭 이름입니다.
- (필수) `TargetFunctionArn` – Lambda 함수의 이름 또는 Amazon 리소스 이름(ARN)입니다. 유효한 이름 형식은 다음과 같습니다.
  - 함수 이름 - `my-function`
  - 함수 ARN - `arn:aws:lambda:us-west-2:123456789012:function:my-function`
  - 부분적 ARN - `123456789012:function:my-function`

교차 오리진 리소스 공유(CORS)

다양한 오리진이 함수 URL에 액세스하는 방법을 정의하려면 교차 오리진 리소스 공유(CORS)를 사용합니다. 다른 도메인에서 함수 URL을 호출하려는 경우 CORS를 구성하는 것이 좋습니다. Lambda는 함수 URL에 대해 다음 CORS 헤더를 지원합니다.

CORS 헤더	CORS 구성 속성	예제 값
Access-Control-Allow-Origin	AllowOrigins	* (모든 오리진 허용) 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 (기본값), 300

Lambda 콘솔 또는 AWS CLI를 사용하여 함수 URL에 대한 CORS를 구성하는 경우, Lambda는 함수 URL을 통해 모든 응답에 CORS 헤더를 자동으로 추가합니다. 또는 함수 응답에 CORS 헤더를 수동으로 추가할 수 있습니다. 충돌하는 헤더가 있는 경우 함수 URL에 구성된 CORS 헤더가 우선합니다.

함수 URL 제한

제한은 함수가 요청을 처리하는 속도를 제한합니다. 이는 함수가 다운스트림 리소스를 오버로드하지 못하게 하거나 갑작스러운 요청 급증을 처리하는 등 여러 상황에서 유용합니다.

예약된 동시성을 구성하여 Lambda 함수가 함수 URL을 통해 처리하는 요청 속도를 제한할 수 있습니다. 예약된 동시성은 함수에 대한 최대 동시 호출 수를 제한합니다. 함수의 초당 최대 요청 속도(RPS)는 구성된 예약된 동시성의 10배에 해당합니다. 예를 들어 예약된 동시성이 100인 함수를 구성하는 경우 최대 RPS는 1,000입니다.

함수 동시성이 예약된 동시성을 초과할 때마다 함수 URL은 HTTP 429 상태 코드를 반환합니다. 함수가 구성된 예약된 동시성을 기준으로 10배의 RPS 최대값을 초과하는 요청을 수신하면 HTTP 429 오

류도 수신됩니다. 예약된 동시성에 대한 자세한 내용은 함수에 대해 예약된 동시성 구성 섹션을 참조하세요.

함수 URL 비활성화

비상시에는 함수 URL에 대한 모든 트래픽을 거부할 수 있습니다. 함수 URL을 비활성화하려면 예약된 동시성을 0으로 설정합니다. 이렇게 하면 함수 URL에 대한 모든 요청을 제한하여 HTTP 429 상태 응답이 발생합니다. 함수 URL을 다시 활성화하려면 예약된 동시성 구성을 삭제하거나 구성을 0보다 큰 값으로 설정합니다.

함수 URL 삭제

함수 URL을 삭제하면 복구할 수 없습니다. 새 함수 URL을 생성하면 URL 주소가 달라집니다.

> **Note**
>
> `NONE` 인증 유형이 있는 함수 URL을 삭제하는 경우, Lambda는 연결된 리소스 기반 정책을 자동으로 삭제하지 않습니다. 이 정책을 삭제하려면 수동으로 삭제해야 합니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 함수 URL(Function URL)을 선택합니다.
4. Delete(삭제)를 선택합니다.
5. 필드에 삭제라는 단어를 입력하여 삭제를 확인합니다.
6. Delete(삭제)를 선택합니다.

> **Note**
>
> 함수 URL이 있는 함수를 삭제할 경우 Lambda는 함수 URL을 비동기적으로 삭제합니다. 동일한 계정에서 동일한 이름으로 새 함수를 즉시 생성하는 경우 원래 함수 URL이 삭제되는 대신 새 함수에 매핑될 수 있습니다.

Lambda 함수 URL에 대한 액세스 제어

AuthType 파라미터를 특정 함수에 연결된 리소스 기반 정책과 함께 사용하여 Lambda 함수 URL에 대한 액세스를 제어할 수 있습니다. 이 두 구성 요소의 구성에 따라 함수 URL에서 다른 관리 작업을 호출하거나 수행할 수 있는 사람이 결정됩니다.

AuthType 파라미터는 Lambda가 함수 URL에 대한 요청을 인증하거나 권한을 부여하는 방법을 결정합니다. 함수 URL을 구성할 때 다음 AuthType 옵션 중 하나를 지정해야 합니다.

- AWS_IAM – Lambda는 AWS Identity and Access Management(IAM)를 사용하여 IAM 보안 주체의 자격 증명 정책 및 함수의 리소스 기반 정책에 따라 요청을 인증하고 권한을 부여합니다. 인증된 사용자 및 역할만 함수 URL을 통해 함수를 호출하도록 하려면 이 옵션을 선택합니다.

- NONE – Lambda가 함수를 호출하기 전에 인증을 수행하지 않습니다. 그러나 함수의 리소스 기반 정책은 항상 유효하며 함수 URL이 요청을 수신하기 전에 퍼블릭 액세스 권한을 부여해야 합니다. 함수 URL에 인증되지 않은 퍼블릭 액세스를 허용하려면 이 옵션을 선택합니다.

AuthType과 함께 리소스 기반 정책을 사용하여 다른 AWS 계정가 함수를 호출하도록 권한을 부여할 수도 있습니다. 자세한 정보는 Lambda에서 리소스 기반 정책 작업 섹션을 참조하세요.

보안에 대한 추가 인사이트를 얻기 위해, AWS Identity and Access Management Access Analyzer을 사용하여 함수 URL에 대한 외부 액세스의 종합적인 분석을 확인할 수 있습니다. 또한 IAM Access Analyzer는 Lambda 함수에 대한 신규 또는 업데이트된 권한을 모니터링하여 퍼블릭 및 교차 계정 액세스를 부여하는 권한을 식별하는 데 도움이 됩니다. IAM Access Analyzer는 모든 AWS 고객이 무료로 사용할 수 있습니다. IAM Access Analyzer를 시작하려면 AWS IAM Access Analyzer 사용 섹션을 참조하세요.

이 페이지에는 두 인증 유형에 대한 리소스 기반 정책의 예와 AddPermission API 작업 또는 Lambda 콘솔을 사용하여 이러한 정책을 생성하는 방법도 포함되어 있습니다. 권한을 설정한 후 함수 URL을 호출하는 방법에 대한 자세한 내용은 Lambda 함수 URL 호출 섹션을 참조하세요.

주제

- AWS_IAM 인증 유형 사용
- NONE 인증 유형 사용
- 거버넌스 및 액세스 제어

AWS_IAM 인증 유형 사용

AWS_IAM 인증 유형을 선택하는 경우 Lambda 함수 URL을 호출해야 하는 사용자에게 lambda:InvokeFunctionUrl 권한이 있어야 합니다. 호출을 요청하는 사람에 따라 리소스 기반 정책을 사용하여 이 권한을 부여해야 할 수 있습니다.

요청을 하는 보안 주체가 함수 URL과 동일한 AWS 계정에 있는 경우, 보안 주체가 자격 증명 기반 정책에 lambda:InvokeFunctionUrl 권한이 있거나 또는 함수의 리소스 기반 정책에 부여된 권한이 있어야 합니다. 즉, 사용자가 이미 자격 증명 기반 정책에 lambda:InvokeFunctionUrl 권한이 있는 경우 리소스 기반 정책은 선택 사항입니다. 정책 평가는 계정 내에서 요청 허용 여부 결정에 설명된 규칙을 따릅니다.

요청을 하는 보안 주체가 다른 계정에 있는 경우 보안 주체는 lambda:InvokeFunctionUrl 권한을 부여하는 자격 증명 기반 정책과 호출하려는 함수에 대한 리소스 기반 정책에 부여된 권한이 모두 있어야 합니다. 이러한 교차 계정 사례에서 정책 평가는 교차 계정 요청의 허용 여부 결정에 설명된 규칙을 따릅니다.

교차 계정 상호 작용의 예로, 다음과 같은 리소스 기반 정책은 AWS 계정 444455556666의 example 역할이 함수 my-function와 연결된 함수 URL을 호출하도록 허용합니다.

Example 함수 URL 교차 계정 호출 정책

```
{
    "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"
                }
            }
        }
    ]
}
```

다음 단계에 따라 콘솔을 통해 이 정책 설명을 생성할 수 있습니다.

다른 계정에 URL 호출 권한을 부여하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. URL 호출 권한을 부여하려는 함수의 이름을 선택합니다.
3. 구성(Configuration) 탭을 선택한 다음, 권한(Permissions)을 선택합니다.
4. 리소스 기반 정책(Resource-based policy)에서 권한 추가(Add permissions)를 선택합니다.
5. 함수 URL(Function URL)을 선택합니다.
6. 인증 유형(Auth type)에서 AWS_IAM을 선택합니다.
7. (선택 사항) 문 ID(Statement ID)에서 정책 설명의 문 ID를 입력합니다.
8. 보안 주체에 권한을 부여하려는 사용자나 역할의 계정 ID 또는 Amazon 리소스 이름(ARN)을 입력합니다. 예: **444455556666**.
9. 저장(Save)을 선택합니다.

또는 다음 add-permission AWS Command Line Interface(AWS CLI) 명령을 사용하여 이 정책 설명을 생성할 수 있습니다.

```
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
```

앞의 예제에서 `lambda:FunctionUrlAuthType` 조건 키 값은 `AWS_IAM`입니다. 이 정책은 함수 URL의 인증 유형도 `AWS_IAM`인 경우에만 액세스를 허용합니다.

NONE 인증 유형 사용

> **Important**
>
> 함수 URL 인증 유형이 `NONE`이고 퍼블릭 액세스 권한을 부여하는 리소스 기반 정책이 있는 경우, 함수 URL이 있는 인증되지 않은 모든 사용자가 함수를 호출할 수 있습니다.

경우에 따라 함수 URL을 공개할 수 있습니다. 예를 들어 웹 브라우저에서 직접 제출된 요청을 처리하고자 할 수 있습니다. 함수 URL에 대한 퍼블릭 액세스를 허용하려면 `NONE` 인증 유형을 선택합니다.

NONE 인증 유형을 선택하는 경우, Lambda는 IAM을 사용하여 함수 URL에 대한 요청을 인증하지 않습니다. 그러나 사용자는 여전히 `lambda:InvokeFunctionUrl` 권한이 있어야 함수 URL을 성공적으로 호출할 수 있습니다. 다음 리소스 기반 정책을 사용하여 `lambda:InvokeFunctionUrl` 권한을 부여할 수 있습니다.

Example 인증되지 않은 모든 보안 주체에 대한 함수 URL 호출 정책

```
{
    "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**
>
> 콘솔 또는 AWS Serverless Application Model(AWS SAM)을 통해 인증 유형 NONE을 사용하는 함수 URL을 생성하는 경우 Lambda에서 선행 리소스 기반 정책 설명을 자동으로 생성합니다. 정책이 이미 존재하거나 애플리케이션을 생성하는 사용자 또는 역할에 적절한 권한이 없는 경우 Lambda에서 이를 생성하지 않습니다. AWS CLI, AWS CloudFormation, Lambda API를 직접 사용하는 경우 `lambda:InvokeFunctionUrl` 권한을 직접 추가해야 합니다. 이렇게 하면 함수가 공개됩니다.
>
> 또한 NONE 인증 유형이 있는 함수 URL을 삭제하는 경우, Lambda는 연결된 리소스 기반 정책을 자동으로 삭제하지 않습니다. 이 정책을 삭제하려면 수동으로 삭제해야 합니다.

이 문에서 `lambda:FunctionUrlAuthType` 조건 키 값은 NONE입니다. 이 정책 설명은 함수 URL의 인증 유형도 NONE인 경우에만 액세스를 허용합니다.

함수의 리소스 기반 정책이 `lambda:invokeFunctionUrl` 권한을 부여하지 않는 경우 함수 URL이 `NONE` 인증 유형을 사용하더라도 사용자가 함수 URL을 호출하면 403 금지됨 오류 코드가 표시됩니다.

거버넌스 및 액세스 제어

함수 URL 호출 권한 외에도 함수 URL을 구성하는 데 사용되는 작업에 대한 액세스를 제어할 수도 있습니다. Lambda는 함수 URL에 대해 다음과 같은 IAM 정책 작업을 지원합니다.

- `lambda:InvokeFunctionUrl` – 함수 URL을 사용하여 Lambda 함수를 호출합니다.
- `lambda:CreateFunctionUrlConfig` – 함수 URL을 생성하고 해당 `AuthType`을 설정합니다.
- `lambda:UpdateFunctionUrlConfig` – 함수 URL 구성 및 해당 `AuthType`을 업데이트합니다.
- `lambda:GetFunctionUrlConfig` – 함수 URL의 세부 정보를 봅니다.
- `lambda:ListFunctionUrlConfigs` – 함수 URL 구성을 나열합니다.
- `lambda:DeleteFunctionUrlConfig` – 함수 URL을 삭제합니다.

> **Note**
>
> Lambda 콘솔은 `lambda:InvokeFunctionUrl`에 대해서만 권한 추가를 지원합니다. 다른 모든 작업의 경우 Lambda API 또는 AWS CLI를 사용하여 권한을 추가해야 합니다.

다른 AWS 엔터티에 대한 함수 URL 액세스를 허용하거나 거부하려면 IAM 정책에 이러한 작업을 포함합니다. 예를 들어 다음 정책은 계정 123456789012의 함수 **my-function**에 대한 함수 URL을 업데이트할 수 있는 권한을 AWS 계정 444455556666의 `example` 역할에 부여합니다.

Example 교차 계정 함수 URL 정책

```
{
    "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"
        }
    ]
```

```
}
```

조건 키

함수 URL에 대한 세분화된 액세스 제어를 위해 조건 키를 사용합니다. Lambda는 함수 URL에 대해 하나의 추가 조건 키 FunctionUrlAuthType을 지원합니다. FunctionUrlAuthType 키는 함수 URL에서 사용하는 인증 유형을 설명하는 열거형 값을 정의합니다. 이때 값은 AWS_IAM 또는 NONE가 될 수 있습니다.

함수와 연결된 정책에서 이 조건 키를 사용할 수 있습니다. 예를 들어 함수 URL의 구성을 변경할 수 있는 사용자를 제한하려고 할 수 있습니다. URL 인증 유형이 NONE인 모든 함수에 대한 UpdateFunctionUrlConfig 요청을 거부하려면 다음 정책을 정의할 수 있습니다.

Example 명시적 거부를 사용한 함수 URL 정책

```
{
    "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"
                }
            }
        }
    ]
}
```

URL 인증 유형이 AWS_IAM인 함수에 대한 CreateFunctionUrlConfig 및 UpdateFunctionUrlConfig 요청을 할 수 있는 권한을 AWS 계정 444455556666의 example 역할을 부여하려면 다음 정책을 정의할 수 있습니다.

Example 명시적 허용을 사용한 함수 URL 정책

```
{
```

```
    "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"
                }
            }
        }
    ]
}
```

서비스 제어 정책(SCP)에서 이 조건 키를 사용할 수도 있습니다. SCP를 사용하여 AWS Organizations의 조직 전체에서 권한을 관리합니다. 예를 들어, 사용자가 AWS_IAM 이외의 인증 유형을 사용하는 함수 URL을 생성하거나 업데이트하는 것을 거부하려면 다음 서비스 제어 정책을 사용합니다.

Example 명시적 거부를 사용한 함수 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 함수 URL 호출

함수 URL은 Lambda 함수를 위한 전용 HTTP(S) 엔드포인트입니다. Lambda 콘솔 또는 Lambda API를 통해 함수 URL을 생성하고 구성할 수 있습니다. 함수 URL을 생성하면 Lambda가 자동으로 고유한 URL 엔드포인트를 생성합니다. 함수 URL을 생성하면 URL 엔드포인트가 변경되지 않습니다. 함수 URL 엔드포인트는 다음 형식을 취합니다.

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 함수 URL은 아시아 태평양 (하이데라바드) (), 아시아 태평양 (멜버른) (ap-south-2), 캐나다 서부 (캘거리ap-southeast-4) (), 유럽 (스페인) (ca-west-1), 유럽 (취리히) (eu-south-2), 이스라엘 (텔아비브) (eu-central-2), 중동 (UAE) (il-central-1) 지역에서 지원되지 않습니다. me-central-1

함수 URL은 IPv4 및 IPv6을 지원하는 이중 스택을 지원합니다. 함수 URL을 구성한 후 웹 브라우저, curl, Postman 또는 모든 HTTP(S) 클라이언트를 통해 HTTP 엔드포인트에서 함수를 호출할 수 있습니다. 함수 URL을 호출하려면 lambda:InvokeFunctionUrl 권한이 있어야 합니다. 자세한 정보는 액세스 제어 섹션을 참조하세요.

주제

함수 URL 호출 기본 사항

함수 URL에서 AWS_IAM 인증 유형을 사용하는 경우 AWS서명 버전 4(SigV4)를 사용하여 각 HTTP 요청에 서명해야 합니다. awscurl, Postman, AWS SigV4 Proxy와 같은 도구는 SigV4를 사용하여 요청에 서명할 수 있는 방법을 기본으로 제공합니다.

도구를 사용하여 함수 URL에 대한 HTTP 요청에 서명하지 않으면 SigV4를 사용하여 각 요청에 수동으로 서명해야 합니다. 함수 URL이 요청을 수신하면 Lambda는 SigV4 서명도 계산합니다. 서명이 일치하는 경우 Lambda가 요청을 처리합니다. SigV4를 사용하여 요청에 수동으로 서명하는 방법에 대한 지침은 Amazon Web Services 일반 참조 일반 참조 가이드의 서명 버전 4를 사용하여 AWS 요청에 서명을 참조하세요.

함수 URL에서 NONE 인증 유형을 사용하는 경우 SigV4를 사용하여 요청에 서명할 필요가 없습니다. 웹 브라우저, curl, Postman 또는 HTTP 클라이언트를 사용하여 함수를 호출할 수 있습니다.

함수에 대한 간단한 GET 요청을 테스트하려면 웹 브라우저를 사용합니다. 예를 들어 함수 URL이 `https://abcdefg.lambda-url.us-east-1.on.aws`이며 문자열 파라미터 `message`에서 사용되는 경우 요청 URL은 다음과 같을 수 있습니다.

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

POST 요청과 같은 다른 HTTP 요청을 테스트하려면 curl 등의 도구를 사용할 수 있습니다. 예를 들어 함수 URL에 대한 POST 요청에 일부 JSON 데이터를 포함하려면 다음 curl 명령을 사용할 수 있습니다.

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

요청 및 응답 페이로드

클라이언트가 함수 URL을 호출하면 Lambda는 요청을 함수에 전달하기 전에 이벤트 객체에 매핑합니다. 그러면 함수의 응답이 HTTP 응답에 매핑되고, Lambda는 해당 HTTP 응답을 함수 URL을 통해 클라이언트로 다시 전송합니다.

요청 및 응답 이벤트 형식은 Amazon API Gateway 페이로드 포맷 버전 2.0과 동일한 스키마를 따릅니다.

요청 페이로드 형식

요청 페이로드는 다음 구조를 취합니다.

```
{
  "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
}
```

파라미터	설명	예
version	이 이벤트에 대한 페이로드 포맷 버전입니다. Lambda 함수 URL은 현재 페이로드 포맷 버전 2.0을 지원합니다.	2.0
routeKey	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 자리표시자를 의미하는 $default로 설정합니다.	$default
rawPath	요청 경로입니다. 예를 들어, 요청 URL이 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo 인 경우 원시 경로 값은 /example/test/demo 입니다.	/example/test/demo
rawQueryString	요청의 쿼리 문자열 파라미터를 포함하는 원시 문자열입니다. 지원되는 문자에는 a-z, A-Z, 0-9, ., _, -, %, &, = 및 +가 포함됩니다.	"?parameter1=value1&parameter2=value2"
cookies	요청의 일부로 전송되는 모든 쿠키를 포함하는 배열입니다.	["Cookie_1=Value_1", "Cookie_2=Value_2"]
headers	요청 헤더 목록으로, 키-값 페어로 표시됩니다.	{"header1": "value1", "header2": "value2"}
queryStringParameters	요청의 쿼리 파라미터입니다. 예를 들어, 요청 URL이 https://{url-id}.lambda-url.{region}.on.aws/e	{"name": "Jane"}

파라미터	설명	예
	xample?name=Jane 인 경우 queryStringParameters 값은 키가 name이고 값이 Jane인 JSON 객체입니다.	
requestContext	요청에 대한 추가 정보를 포함하는 객체입니다(예: requestId , 요청 시간, AWS Identity and Access Management(IAM)를 통해 승인된 호출자의 ID).	
requestContext.accountId	함수 소유자의 AWS 계정 ID입니다.	"123456789012"
requestContext.apiId	함수 URL의 ID입니다.	"33anwqw8fj"
requestContext.authentication	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 null로 설정합니다.	null
requestContext.authorizer	함수 URL에서 AWS_IAM 인증 유형을 사용하는 경우 호출자 자격 증명에 관한 정보를 포함하는 객체입니다. 그렇지 않으면 Lambda가 이 파라미터를 null로 설정합니다.	
requestContext.authorizer.iam.accessKey	호출자 자격 증명의 액세스 키입니다.	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer.iam.accountId	호출자 자격 증명의 AWS 계정 ID입니다.	"111122223333"

파라미터	설명	예
requestContext.authorizer.iam.callerId	호출자의 ID(사용자 ID)입니다.	"AIDACKCEVSQ6C2EXAMPLE"
requestContext.authorizer.iam.cognitoIdentity	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 null로 설정하거나 JSON에서 제외합니다.	null
requestContext.authorizer.iam.principalOrgId	호출자 자격 증명과 연결된 보안 주체 조직 ID입니다.	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer.iam.userArn	호출자 자격 증명의 사용자 Amazon 리소스 이름(ARN)입니다.	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer.iam.userId	호출자 자격 증명의 사용자 ID입니다.	"AIDACOSFODNN7EXAMPLE2"
requestContext.domainName	함수 URL의 도메인 이름입니다.	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	함수 URL의 도메인 접두사입니다.	"<url-id>"
requestContext.http	HTTP 요청에 대한 세부 정보를 포함하는 객체입니다.	
requestContext.http.method	이 요청에 사용되는 HTTP 메서드입니다. 유효한 값에는 GET, POST, PUT, HEAD, OPTIONS, PATCH 및 DELETE가 있습니다.	GET

파라미터	설명	예
requestContext.http.path	요청 경로입니다. 예를 들어, 요청 URL이 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo 인 경우 경로 값은 /example/test/demo 입니다.	/example/test/demo
requestContext.http.protocol	요청의 프로토콜입니다.	HTTP/1.1
requestContext.http.sourceIp	요청하는 즉시 TCP 연결의 소스 IP 주소입니다.	123.123.123.123
requestContext.http.userAgent	사용자 에이전트 요청 헤더 값입니다.	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0
requestContext.requestId	호출 요청의 ID입니다. 이 ID를 사용하여 함수와 관련된 호출 로그를 추적할 수 있습니다.	e1506fd5-9e7b-434f-bd42-4f8fa224b599
requestContext.routeKey	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 자리표시자를 의미하는 $default로 설정합니다.	$default
requestContext.stage	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 자리표시자를 의미하는 $default로 설정합니다.	$default

파라미터	설명	예
requestContext.time	요청의 타임스탬프입니다.	"07/Sep/2021:22:50:22 +0000"
requestContext.timeEpoch	Unix Epoch 시간으로 표시된 요청의 타임스탬프입니다.	"1631055022677"
body	요청의 본문입니다. 요청의 콘텐츠 유형이 바이너리인 경우 본문은 base64로 인코딩됩니다.	{"key1": "value1", "key2": "value2"}
pathParameters	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 null로 설정하거나 JSON에서 제외합니다.	null
isBase64Encoded	본문이 이진 페이로드이고 base64로 인코딩되는 경우 TRUE입니다. 그렇지 않으면 FALSE입니다.	FALSE
stageVariables	함수 URL은 이 파라미터를 사용하지 않습니다. Lambda는 이 파라미터를 null로 설정하거나 JSON에서 제외합니다.	null

응답 페이로드 형식

함수가 응답을 반환하면 Lambda는 응답을 구문 분석하여 HTTP 응답으로 변환합니다. 함수 응답 페이로드는 다음 형식을 취합니다.

```
{
   "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
}
```

Lambda는 응답 형식을 추론합니다. 함수가 유효한 JSON을 반환하고 `statusCode`을 반환하지 않는 경우 Lambda가 다음과 같은 가정을 합니다.

- `statusCode`는 `200`입니다.
- `content-type`은 `application/json`입니다.
- `body`는 함수의 응답입니다.
- `isBase64Encoded`는 `false`입니다.

다음 예제에서는 Lambda 함수의 출력이 응답 페이로드에 매핑되는 방법과 응답 페이로드가 최종 HTTP 응답에 매핑되는 방법을 보여 줍니다. 클라이언트가 함수 URL을 호출하면 HTTP 응답이 표시됩니다.

문자열 응답에 대한 예제 출력

Lambda 함수 출력	해석된 응답 출력	HTTP 응답(클라이언트에게 표시되는 내용)

`"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 응답에 대한 예제 출력

Lambda 함수 출력	해석된 응답 출력	HTTP 응답(클라이언트에게 표시되는 내용)

<pre>{
  "message": "Hello,
 world!"
}</pre>
	<pre>{
	  "statusCode": 200,
	  "body": {
	  "message": "Hello,
	 world!"
	  },
	  "headers": {
	  "content-type":
	 "application/json"
	  },
	  "isBase64Encoded":
	 false
	}</pre>
		<pre>HTTP/2 200
		date: Wed, 08 Sep 2021
		 18:02:24 GMT
		content-type: applicati
		on/json
		content-length: 34
		
		{
		  "message": "Hello,
		 world!"
		}</pre>

사용자 지정 응답에 대한 예제 출력

Lambda 함수 출력	해석된 응답 출력	HTTP 응답(클라이언트에게 표시되는 내용)

<pre>{
  "statusCode": 201,
  "headers": {
  "Content-Type":
 "application/json",
  "My-Custom-
Header": "Custom Value"
  },
  "body": JSON.stri
ngify({
  "message":
 "Hello, world!"
  }),
  "isBase64Encoded":
 false
}</pre>
	<pre>{
	  "statusCode": 201,
	  "headers": {
	  "Content-Type":
	 "application/json",
	  "My-Custom-
	Header": "Custom Value"
	  },
	  "body": JSON.stri
	ngify({
	  "message":
	 "Hello, world!"
	  }),
	  "isBase64Encoded":
	 false
	}</pre>
		<pre>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!"
		}</pre>

쿠키

함수에서 쿠키를 반환하려면 수동으로 `set-cookie` 헤더를 추가하지 않습니다. 대신 응답 페이로드 객체에 쿠키를 포함합니다. Lambda는 이를 자동으로 해석하고 다음 예제와 같이 HTTP 응답의 `set-cookie` 헤더로서 추가합니다.

응답 반환 쿠키에 대한 예제 출력

Lambda 함수 출력

```
{
   "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=7
8000"
    ],
    "isBase64Encoded": false
}
```

HTTP 응답(클라이언트에게 표시되는 내용)

```
HTTP/2 201
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!"
}
```

Lambda 함수 URL 모니터링

AWS CloudTrail과 Amazon CloudWatch를 사용하여 함수 URL을 모니터링할 수 있습니다.

주제

- CloudTrail을 사용하여 함수 URL 모니터링
- 함수 URL에 대한 CloudWatch 지표

CloudTrail을 사용하여 함수 URL 모니터링

함수 URL에 대해 Lambda는 CloudTrail 로그 파일에 다음 API 작업을 이벤트로 로깅하는 것을 자동 지원합니다.

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

각 로그 항목에는 호출자 자격 증명, 요청이 이루어진 시기, 기타 세부 정보에 관한 정보가 포함되어 있습니다. CloudTrail 이벤트 기록(Event history)을 확인하면 지난 90일 이내의 모든 이벤트를 볼 수 있습니다. 90일이 지난 레코드를 보존하려면 추적을 생성할 수 있습니다.

기본적으로 CloudTrail은 데이터 이벤트로 간주하는 `InvokeFunctionUrl` 요청을 로그하지 않습니다. 그러나 CloudTrail에서 데이터 이벤트 로깅을 활성화할 수 있습니다. 자세한 내용은 AWS CloudTrail 사용 설명서의 추적을 위해 데이터 이벤트 로깅을 참조하십시오.

함수 URL에 대한 CloudWatch 지표

Lambda는 함수 URL 요청에 대한 집계된 지표를 CloudWatch로 전송합니다. 이러한 지표를 사용하면 CloudWatch 콘솔에서 함수 URL을 모니터링하고 대시보드를 구축하고 경보를 구성할 수 있습니다.

함수 URL은 다음 호출 지표를 지원합니다. `Sum` 통계를 사용하여 이러한 지표를 볼 것을 권장합니다.

- `UrlRequestCount` – 이 함수 URL에 수행된 요청 수.
- `Url4xxCount` – 4XX HTTP 상태 코드를 반환한 요청 수. 4XX 시리즈 코드는 잘못된 요청과 같은 클라이언트 측 오류를 나타냅니다.

- Url5xxCount – 5XX HTTP 상태 코드를 반환한 요청 수. 5XX 시리즈 코드는 함수 오류 및 제한 시간과 같은 서버 측 오류를 나타냅니다.

함수 URL은 다음과 같은 성능 지표도 지원합니다. Average 또는 Max 통계를 사용하여 이러한 지표를 볼 것을 권장합니다.

- UrlRequestLatency – 함수 URL이 요청을 수신하는 시점부터 함수 URL이 응답을 반환하는 시점까지의 시간입니다.

이러한 각 호출 및 성능 지표는 다음 차원을 지원합니다.

- FunctionName – 함수의 $LATEST 게시되지 않은 버전 또는 함수의 별칭에 할당된 함수 URL에 대한 집계 지표를 확인합니다. 예를 들면 hello-world-function입니다.
- Resource – 특정 함수 URL에 대한 지표를 확인합니다 함수 이름과 함수의 $LATEST 게시되지 않은 버전 또는 함수의 별칭 중 하나로 정의합니다. 예를 들면 hello-world-function:$LATEST입니다.
- ExecutedVersion – 실행된 버전을 기반으로 특정 함수 URL에 대한 지표를 확인합니다. 이 차원을 사용하여 주로 $LATEST 게시되지 않은 버전에 할당된 함수 URL을 추적할 수 있습니다.

자습서: 함수 URL을 사용하여 Lambda 함수 생성

이 자습서에서는 두 숫자로 이루어진 PRODUCT 함수를 반환하는 함수 URL 엔드포인트가 있는 .zip 파일 아카이브로 정의된 public 함수를 생성합니다. 함수 구성에 대한 자세한 내용은 함수 URL 생성 및 관리 섹션을 참조하세요.

필수 조건

이 자습서에서는 사용자가 기본 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 그렇지 않은 경우 콘솔로 Lambda 함수 생성의 지침에 따라 첫 Lambda 함수를 생성합니다.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 만들기

Lambda 함수에 AWS 리소스에 액세스할 수 있는 권한을 제공하는 실행 역할을 만듭니다.

실행 역할을 만들려면

1. AWS Identity and Access Management(IAM) 콘솔의 역할 페이지를 엽니다.

2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형으로 AWS 서비스를 선택하고 사용 사례로 Lambda를 선택합니다.
4. 다음을 선택합니다.
5. 권한 정책 창의 검색 상자에 **AWSLambdaBasicExecutionRole**을 입력합니다.
6. AWSLambdaBasicExecutionRole AWS 관리형 정책 옆에 있는 확인란을 선택한 후 다음을 선택합니다.
7. 역할 이름에 **lambda-url-role**을 입력한 다음 역할 생성을 선택합니다.

AWSLambdaBasicExecutionRole 정책은 함수가 Amazon CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다. 자습서 뒷부분에서는 Lambda 함수를 생성하기 위해 역할의 Amazon 리소스 이름(ARN)이 필요합니다.

실행 역할의 ARN을 찾으려면 다음을 수행합니다.

1. AWS Identity and Access Management(IAM) 콘솔의 역할 페이지를 엽니다.
2. 방금 생성한 역할을 선택합니다(lambda-url-role).
3. 요약 창에서 ARN을 복사합니다.

함수 URL(.zip 파일 아카이브)이 있는 Lambda 함수 생성

.zip 파일 아카이브를 사용하여 함수 URL 엔드포인트가 있는 Lambda 함수 생성

함수를 만들려면

1. 다음 코드 예제를 index.js라는 파일에 복사합니다.

   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. 배포 패키지를 만듭니다.

```
zip function.zip index.js
```

3. `create-function` 명령을 사용해 Lambda 함수를 만듭니다. 역할 ARN을 자습서 앞부분에서 복사한 고유한 실행 역할의 ARN으로 바꿔야 합니다.

```
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. 함수 URL에 대한 공개 액세스를 허용하는 권한을 부여하는 리소스 기반 정책을 함수에 추가합니다.

```
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` 명령을 사용하여 함수에 대한 URL 엔드포인트를 만듭니다.

```
aws lambda create-function-url-config \
    --function-name my-url-function \
    --auth-type NONE
```

함수 URL 엔드포인트 테스트

curl 또는 Postman과 같은 HTTP 클라이언트를 사용하여 함수 URL 엔드포인트를 호출하여 Lambda 함수를 호출합니다.

```
curl 'https://abcdefg.lambda-url.us-east-1.on.aws/' \
-H 'Content-Type: application/json' \
-d '{"num1": "10", "num2": "10"}'
```

다음 결과가 표시됩니다.

```
The product of 10 and 10 is 100
```

함수 URL이 있는 Lambda 함수 생성(CloudFormation)

AWS CloudFormation 유형 `AWS::Lambda::Url`을 사용하여 함수 URL 엔드포인트가 있는 Lambda 함수를 생성할 수도 있습니다.

```
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
```

함수 URL(AWS SAM)이 있는 Lambda 함수 생성(CloudFormation)

AWS Serverless Application Model(AWS SAM)을 사용하여 함수 URL로 구성된 Lambda 함수를 생성할 수도 있습니다.

```
ProductFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: function/.
    Handler: index.handler
    Runtime: nodejs18.x
    AutoPublishAlias: live
    FunctionUrlConfig:
      AuthType: NONE
```

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

AWS Lambda 함수 관리

Lambda API 또는 콘솔을 사용하여 Lambda 함수와 관련한 리소스를 조정하고 보호하는 방법을 알아봅니다.

AWS CLI에서 Lambda 사용

AWS Command Line Interface를 사용하여 함수 및 기타 AWS Lambda 리소스를 관리할 수 있습니다. AWS CLI는 AWS SDK for Python (Boto)를 사용하여 Lambda API와 상호 작용합니다. 이 자습서에서는 AWS CLI를 사용하여 Lambda 함수를 관리하고 호출합니다.

함수 규모 조정

예약된 동시성 및 프로비저닝된 동시성이라는 두 가지 함수 수준 동시성 제어를 구성할 수 있습니다. 동시성은 활성 상태인 함수의 인스턴스 수이며 동시성을 구성하여 중요 함수가 제한되지 않도록 보장할 수 있습니다.

코드 서명

Lambda에 대한 코드 서명은 승인된 개발자가 게시한 변경되지 않은 코드만 Lambda 함수에 배포되었는지 확인할 수 있는 신뢰 및 무결성 제어 기능을 제공합니다.

태그로 구성

Lambda 함수에 태그를 지정하여 ABAC(속성 기반 액세스 제어)를 활성화하고 소유자, 프로젝트 또는 부서별로 구성할 수 있습니다.

계층 사용

이전에 생성된 계층을 적용하여 배포 패키지 크기를 줄이고 코드 공유 및 책임 분리를 촉진하여 비즈니스 로직 작성을 더 빠르게 반복 할 수 있습니다.

AWS CLI에서 Lambda 사용

AWS Command Line Interface를 사용하여 함수 및 기타 AWS Lambda 리소스를 관리할 수 있습니다. AWS CLI는 AWS SDK for Python (Boto)를 사용하여 Lambda API와 상호 작용합니다. 이를 통해 API에 대해 배운 후 그 지식을 바탕으로 AWS SDK에서 Lambda를 사용하는 애플리케이션을 빌드할 수 있습니다.

이 자습서에서는 AWS CLI를 사용하여 Lambda 함수를 관리하고 호출합니다. 자세한 내용은 AWS Command Line Interface 사용 설명서의 AWS CLI란 무엇입니까? 단원을 참조하세요.

사전 조건

이 자습서에서는 사용자가 기본적인 Lambda 작업과 Lambda 콘솔에 대해 어느 정도 알고 있다고 가정합니다. 아직 the section called “콘솔로 Lambda 함수 생성”의 지침을 따르지 않았다면 지금 따르세요.

다음 단계를 완료하려면 AWS Command Line Interface(AWS CLI) 버전 2가 필요합니다. 명령과 예상 결과는 별도의 블록에 나열됩니다.

```
aws --version
```

다음 결과가 표시됩니다.

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

긴 명령의 경우 이스케이프 문자(\)를 사용하여 명령을 여러 행으로 분할합니다.

Linux 및 macOS는 선호 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다. 이 안내서의 예제 CLI 명령은 Linux 형식을 사용합니다. Windows CLI를 사용하는 경우 인라인 JSON 문서를 포함하는 명령의 형식을 다시 지정해야 합니다.

실행 역할 생성

함수에 AWS 리소스에 액세스할 수 있는 권한을 제공하는 실행 역할을 만듭니다. AWS CLI를 사용하여 실행 역할을 생성하려면 `create-role` 명령을 사용합니다.

다음 예에서는 신뢰 정책 인라인을 지정합니다. JSON 문자열의 인용 부호 요구 사항은 셸에 따라 다릅니다.

```
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"}]}'
```

JSON 파일을 사용하여 역할에 대한 신뢰 정책을 정의할 수도 있습니다. 다음 예제에서 `trust-policy.json`은 최신 디렉터리의 파일입니다. 이 신뢰 정책을 통해 Lambda는 서비스 보안 주체에 `lambda.amazonaws.com` 권한을 제공하여 AWS Security Token Service(AWS STS)`AssumeRole` 작업을 호출하기 위해 역할의 권한을 사용할 수 있습니다.

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
```

다음 결과가 표시됩니다.

```
{
    "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"
                }
            ]
        }
    }
}
```

역할에 권한을 추가하려면 attach-policy-to-role 명령을 사용하세요. 먼저 `AWSLambdaBasicExecutionRole` 관리형 정책을 추가합니다.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

AWSLambdaBasicExecutionRole정책에는 함수가 로그를 로그에 기록하는 데 필요한 권한이 있습니다. CloudWatch

함수 생성

다음 예제는 환경 변수의 값과 이벤트 객체를 로깅합니다.

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
}
```

함수를 만들려면

1. 샘플 코드를 `index.js` 파일에 복사합니다.

2. 배포 패키지를 만듭니다.

```
zip function.zip index.js
```

3. `create-function` 명령을 사용해 Lambda 함수를 만듭니다. 역할 ARN에서 강조 표시된 텍스트를 계정 ID로 바꿉니다.

```
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
```

다음 결과가 표시됩니다:

```
{
    "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",
    ...
}
```

명령줄에서 호출에 대한 로그를 가져오려면 `--log-type` 옵션을 사용하세요. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
```

```
}
```

base64 유틸리티를 사용하면 로그를 디코딩할 수 있습니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text |  base64 -d
```

다음 결과가 표시됩니다:

```
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
```

base64는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS의 경우, 명령은 base64 -D입니다.

명령줄에서 전체 로그 이벤트를 가져오기 위해 앞의 예와 같이 함수 출력에 로그 스트림 이름을 포함할 수 있습니다. 다음 예제 스크립트는 my-function이라는 함수를 호출하고 마지막 5개 로그 이벤트를 다운로드합니다.

Example get-logs.sh 스크립트

이 예에서는 my-function이 로그 스트림 ID를 반환해야 합니다.

```
#!/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
```

스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다.


```
{
    "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"
}
```

함수 업데이트

함수를 만든 후 트리거, 네트워크 액세스, 파일 시스템 액세스와 같은 함수에 대한 추가 기능을 구성할 수 있습니다. 함수와 관련된 리소스(예: 메모리 및 동시성)를 조정할 수도 있습니다. 이러한 구성은 .zip 파일 아카이브로 정의된 함수와 컨테이너 이미지로 정의된 함수에 적용됩니다.

update-function-configuration명령을 사용하여 함수를 구성합니다. 다음 예제에서는 함수 메모리를 256MB로 설정합니다.

Example update-function-configuration 명령

```
aws lambda update-function-configuration \
--function-name my-function \
--memory-size 256
```

계정의 Lambda 함수 목록 표시

다음 AWS CLI `list-functions` 명령을 실행하여, 생성한 함수의 목록을 검색할 수 있습니다.

```
aws lambda list-functions --max-items 10
```

다음 결과가 표시됩니다:

```
{
    "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="
}
```

응답에서 Lambda는 최대 10개의 함수로 이루어진 목록을 반환합니다. 검색할 수 있는 함수가 추가로 있는 경우, NextToken은 다음 `list-functions` 요청에서 사용할 수 있도록 마커를 제공합니다. 다음의 `list-functions` AWS CLI 명령은 `--starting-token` 파라미터를 보여주는 예제입니다.

```
aws lambda list-functions --max-items 10 --starting-
token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxMH0=
```

Lambda 함수 검색

Lambda CLI `get-function` 명령은 함수의 배포 패키지를 다운로드할 때 사용할 수 있도록 Lambda 함수 메타데이터와 미리 서명된 URL을 반환합니다.

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

다음 결과가 표시됩니다.

```
{
    "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-..."
```

```
    }
}
```

자세한 내용은 을 참조하십시오 GetFunction.

정리

다음 `delete-function` 명령을 실행하여 `my-function` 함수를 삭제합니다.

```
aws lambda delete-function --function-name my-function
```

IAM 콘솔에서 생성한 IAM 역할을 삭제합니다. 역할 삭제에 대한 내용은 IAM 사용 설명서의 역할 또는 인스턴스 프로필 삭제를 참조하세요.

Lambda 함수 규모 조정 이행

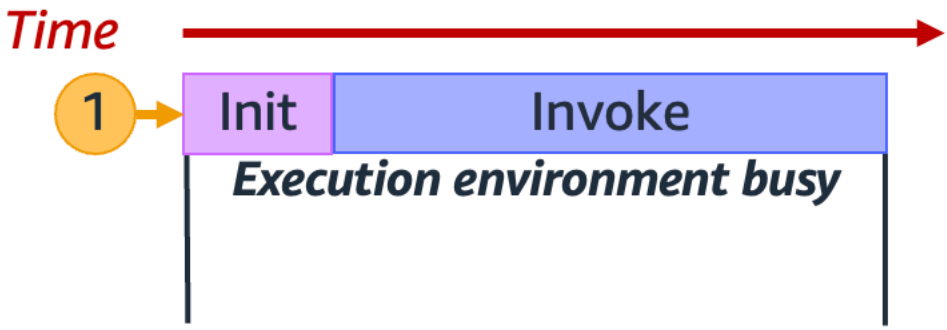
동시성은 AWS Lambda 함수가 동시에 처리하는 전송 중인 요청 수입니다. 각 동시 요청마다 Lambda는 실행 환경의 개별 인스턴스를 프로비저닝합니다. 함수가 요청을 더 많이 수신하면 Lambda는 사용자가 계정의 동시성 한도에 도달할 때까지 실행 환경 수를 자동으로 조정합니다. 기본적으로 Lambda는 사용자 계정에 AWS 리전 내 모든 함수에 걸쳐 총 1,000개 동시 실행 한도의 동시성을 제공합니다. 특정 계정 요구 사항을 지원하기 위해, 할당량 증대를 요청하고 함수 수준의 동시성 제어를 구성하여 중요한 함수에 대해 제한이 발생하지 않도록 할 수 있습니다.

이 주제에서는 Lambda의 동시성 개념과 함수 스케일 인에 대해 설명합니다. 이 주제를 마치면 동시성을 계산하고, 두 가지 주요 동시성 제어 옵션(예약 및 프로비저닝)을 시각화하고, 적절한 동시성 제어 설정을 추정하며, 추가 최적화를 위한 지표를 보는 방법을 이해할 수 있습니다.

Sections

동시성 이해 및 시각화

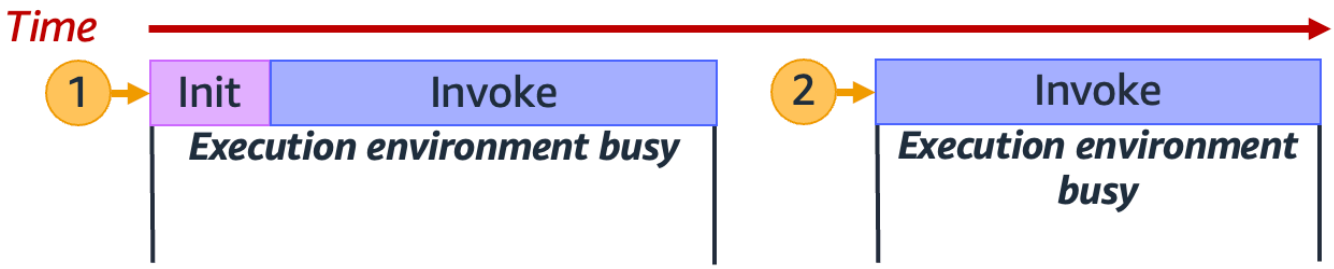
Lambda는 안전하고 격리된 실행 환경에서 함수를 호출합니다. 요청을 처리하려면 Lambda가 실행 환경을 사용하여 함수를 호출(호출 단계)하기 전에 먼저 실행 환경을 초기화(초기화 단계)해야 합니다 .



> **Note**
>
> 실제 초기화 및 호출 지속 시간은 선택한 런타임 및 Lambda 함수 코드와 같은 여러 요인에 따라 달라질 수 있습니다. 위 다이어그램은 초기화 및 호출 단계 지속 시간의 정확한 비율을 보여주기 위한 것이 아닙니다.

위 다이어그램에서는 사각형을 사용하여 단일 실행 환경을 나타냅니다. 함수가 첫 번째 요청(1 레이블이 있는 노란색 원으로 표시됨)을 수신하면 Lambda는 새 실행 환경을 생성하고 초기화 단계 중에 기본 핸들러 외부에서 코드를 실행합니다. 그런 다음 Lambda는 호출 단계에서 함수의 기본 핸들러 코드를 실행합니다. 이 전체 프로세스를 실행하는 동안 이 실행 환경은 사용량이 많아 다른 요청을 처리할 수 없습니다.

Lambda가 첫 번째 요청 처리를 마치면 이 실행 환경에서 동일한 함수에 대한 추가 요청을 처리할 수 있습니다. 후속 요청을 처리하기 위해 Lambda는 환경을 다시 초기화할 필요가 없습니다.

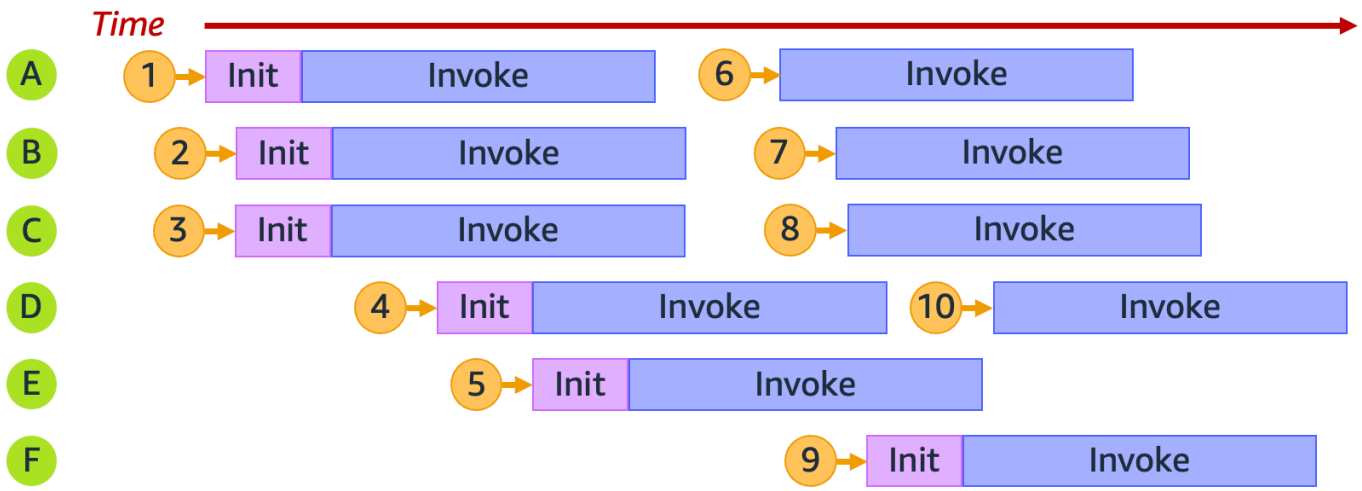


이전 다이어그램에서 Lambda는 실행 환경을 재사용하여 두 번째 요청(2 레이블이 있는 노란색 원으로 표시됨)을 처리합니다.

지금까지는 실행 환경의 단일 인스턴스(즉, 동시성 1)에만 초점을 맞췄습니다. 실제 환경에서 Lambda는 들어오는 모든 요청을 처리하기 위해 여러 실행 환경 인스턴스를 병렬로 프로비저닝해야 할 수 있습니다. 함수가 새 요청을 받으면 다음 두 가지 중 하나가 발생할 수 있습니다.

- 사전 초기화된 실행 환경 인스턴스를 사용할 수 있는 경우 Lambda는 이를 사용하여 요청을 처리합니다.
- 그렇지 않으면 Lambda는 요청을 처리하기 위해 새 실행 환경 인스턴스를 생성합니다.

예를 들어 함수가 10개의 요청을 수신하면 어떻게 되는지 살펴보겠습니다.



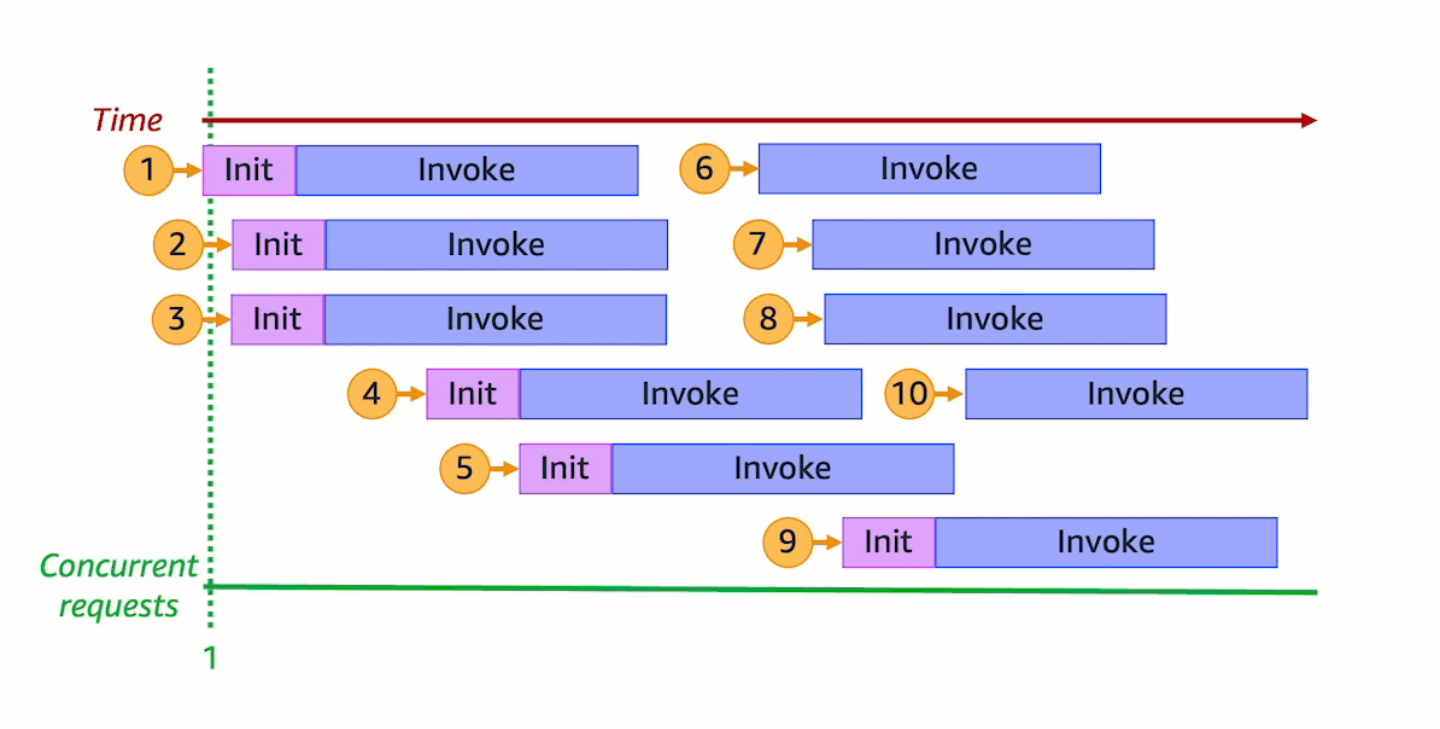
위 다이어그램에서 각 수평면은 단일 실행 환경 인스턴스(A~F의 레이블로 표시됨)를 나타냅니다. Lambda가 각 요청을 처리하는 방법은 다음과 같습니다.

요청 1~10에 대한 Lambda 동작

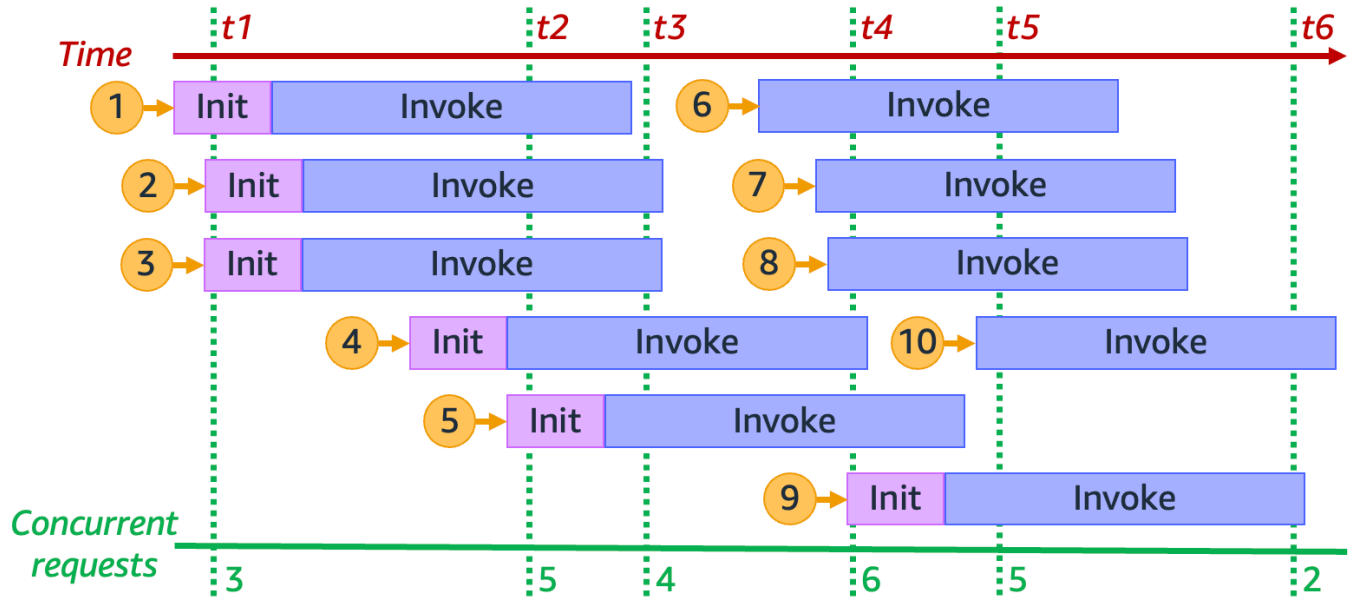
요청	Lambda 동작	추론
1	새로운 환경 A 프로비저닝	첫 번째 요청, 사용 가능한 실행 환경 인스턴스가 없습니다.
2	새로운 환경 B 프로비저닝	기존 실행 환경 인스턴스 A가 사용 중입니다.
3	새로운 환경 C 프로비저닝	기존 실행 환경 인스턴스 A 및 B가 모두 사용 중입니다.

요청	Lambda 동작	추론
4	새로운 환경 D 프로비저닝	기존 실행 환경 인스턴스 A, B 및 C가 모두 사용 중입니다.
5	새로운 환경 E 프로비저닝	기존 실행 환경 인스턴스 A, B, C 및 D가 모두 사용 중입니다.
6	환경 A 재사용	실행 환경 인스턴스 A가 요청 1의 처리를 완료했으며 이제 사용 가능한 상태입니다.
7	환경 B 재사용	실행 환경 인스턴스 B가 요청 2의 처리를 완료했으며 이제 사용 가능한 상태입니다.
8	환경 C 재사용	실행 환경 인스턴스 C가 요청 3의 처리를 완료했으며 이제 사용 가능한 상태입니다.
9	새로운 환경 F 프로비저닝	기존 실행 환경 인스턴스 A, B, C, D 및 E가 모두 사용 중입니다.
10	환경 D 재사용	실행 환경 인스턴스 D가 요청 4의 처리를 완료했으며 이제 사용 가능한 상태입니다.

함수가 더 많은 동시 요청을 수신함에 따라 Lambda는 이에 대응하여 실행 환경 인스턴스의 수를 늘립니다. 다음 애니메이션에서는 시간 경과에 따른 동시 요청 수를 추적합니다.



이전 애니메이션을 서로 다른 여섯 개 시점에서 고정하면 다음과 같은 다이어그램을 얻을 수 있습니다.



이전 다이어그램에서는 어느 지점에서든 수직선을 그리고 이 선과 교차하는 환경의 수를 계산할 수 있습니다. 이를 통해 해당 시점의 동시 요청 수를 알 수 있습니다. 예를 들어 `t1` 시점에는 세 개의 동시 요청을 처리하는 세 개의 활성 환경이 있습니다. 이 시뮬레이션의 최대 동시 요청 수는 활성 환경 여섯 개에서 여섯 개의 동시 요청을 처리하는 `t4` 시점에 발생합니다.

한마디로, 함수의 동시성은 동시에 처리하는 동시 요청의 수입니다. 함수의 동시성이 증가함에 따라 Lambda는 요청 수요를 충족하기 위해 더 많은 실행 환경 인스턴스를 프로비저닝합니다.

함수의 동시성 계산

일반적으로 시스템의 동시성은 둘 이상의 작업을 동시에 처리할 수 있는 능력입니다. Lambda에서 동시성은 함수가 동시에 처리하는 전송 중인 요청의 수입니다. Lambda 함수의 동시성을 측정하는 빠르고 실용적인 방법은 다음 수식을 사용하는 것입니다.

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

동시성은 초당 요청 수와 다릅니다. 예를 들어 함수가 초당 평균 100개의 요청을 수신한다고 가정해 보겠습니다. 평균 요청 지속 시간이 1초인 경우에는 동시성도 100입니다.

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

하지만 평균 요청 지속 시간이 500ms인 경우에는 그럼 동시성은 50입니다.

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

50이라는 동시성은 실제로 무엇을 의미할까요? 평균 요청 지속 시간이 500ms인 경우 그럼 함수 인스턴스가 초당 두 개의 요청을 처리할 수 있다고 생각할 수 있습니다. 그러면, 초당 100개의 요청 로드를 처리하려면 함수의 인스턴스가 50개 필요합니다. 동시성이 50이라는 것은 Lambda가 제한 없이 이 워크로드를 효율적으로 처리하기 위해서는 50개의 실행 환경 인스턴스를 프로비저닝해야 한다는 것을 의미합니다. 이를 방정식 형식으로 표현하면 다음과 같습니다.

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

함수가 두 배의 요청 수(초당 200개 요청)를 수신하지만 각 요청을 처리하는 데 걸리는 시간이 절반인 경우(250ms) 그럼 동시성은 여전히 50입니다.

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

동시성에 대한 이해도 테스트

실행하는 데 평균 200ms가 걸리는 함수가 있다고 가정해 보겠습니다. 최대 부하에서는 초당 5,000개의 요청이 관찰됩니다. 최대 부하 시 함수의 동시성은 얼마일까요?

정답

평균 함수 지속 시간은 200밀리초, 즉 0.2초입니다. 동시성 수식을 사용하여 숫자를 대입하면 1,000이라는 동시성을 구할 수 있습니다.

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

다르게 표현하면, 평균 함수 지속 시간이 200ms이면 함수에서 초당 5개의 요청을 처리할 수 있습니다. 초당 5,000개의 요청 워크로드를 처리하려면 1,000개의 실행 환경 인스턴스가 필요합니다. 따라서 동시성은 1,000입니다.

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

동시성과 초당 요청 수 구분

이전 섹션에서 설명한 것처럼 동시성은 초당 요청 수와 다릅니다. 이는 평균 요청 시간이 100ms 미만인 함수 작업 시 특히 중요한 차이점입니다.

일반적으로 실행 환경의 각 인스턴스는 초당 최대 10개의 요청을 처리할 수 있습니다. 이 제한은 프로비저닝된 동시성을 사용하는 함수뿐만 아니라 동기식 온디맨드 함수에도 적용됩니다. 이 제한에 익숙하지 않은 경우 특정 시나리오에서 이러한 함수에 제한이 발생할 수 있는 이유를 알지 못할 수 있습니다.

평균 요청 기간이 50ms인 함수를 예로 들어 보겠습니다. 초당 200개의 요청에서 이 함수의 동시성은 다음과 같습니다.

```
Concurrency = (200 requests/second) * (0.05 second/request) = 10
```

이 결과에 따라 이 로드를 처리하는 데 오직 10개의 실행 환경 인스턴스만 필요하다고 예상할 수 있습니다. 그러나 각 실행 환경은 초당 10개의 실행만 처리할 수 있습니다. 즉, 10개의 실행 환경에서는 함수가 총 200개의 요청 중 초당 100개의 요청만 처리할 수 있습니다. 이 함수에 제한이 발생합니다.

이 과정에서 반드시 함수에 대한 동시성 설정을 구성할 때 동시성과 초당 요청 수를 모두 고려해야 합니다. 이 경우 함수의 동시성은 오직 10개뿐이지만 함수에는 20개의 실행 환경이 필요합니다.

동시성에 대한 이해도 테스트(100ms 미만 함수)

실행하는 데 평균 20ms가 걸리는 함수가 있다고 가정합니다. 최대 부하에서는 초당 3,000개의 요청이 관찰됩니다. 최대 부하 시 함수의 동시성은 얼마일까요?

정답

평균 함수 지속 시간은 20밀리초, 즉 0.02초입니다. 동시성 수식을 사용하여 숫자를 대입하면 60이라는 동시성을 구할 수 있습니다.

```
Concurrency = (3,000 requests/second) * (0.02 seconds/request) = 60
```

그러나 각 실행 환경은 초당 10개의 요청만 처리할 수 있습니다. 60개의 실행 환경에서 함수는 초당 최대 600개의 요청을 처리할 수 있습니다. 3,000개의 요청을 완전히 수용하려면 함수에는 최소 300개의 실행 환경 인스턴스가 필요합니다.

예약된 동시성 및 프로비저닝된 동시성 이해

기본적으로 리전 내 모든 함수에 걸쳐 사용자 계정의 동시성 한도는 1,000개의 동시 실행입니다. 함수는 이 1,000개의 동시성 풀을 온디맨드 방식으로 공유합니다. 사용 가능한 동시성이 부족하면 함수에서 제한이 발생합니다(즉, 요청 삭제가 시작됨).

일부 함수는 다른 함수보다 더 중요할 수 있습니다. 따라서 중요한 함수가 필요한 동시성을 확보하도록 동시성 설정을 구성해야 할 수 있습니다. 예약된 동시성 및 프로비저닝된 동시성의 두 가지 유형의 동시성 제어가 제공됩니다.

- 예약된 동시성을 사용하여 함수에 대해 계정의 동시성 부분을 예약합니다. 이는 사용 가능한 예약되지 않은 동시성을 다른 함수가 모두 차지하지 않도록 하려는 경우에 유용합니다.
- 프로비저닝된 동시성을 사용하여 함수의 여러 환경 인스턴스를 미리 초기화합니다. 이는 콜드 스타트 지연 시간을 줄이는 데 유용합니다.

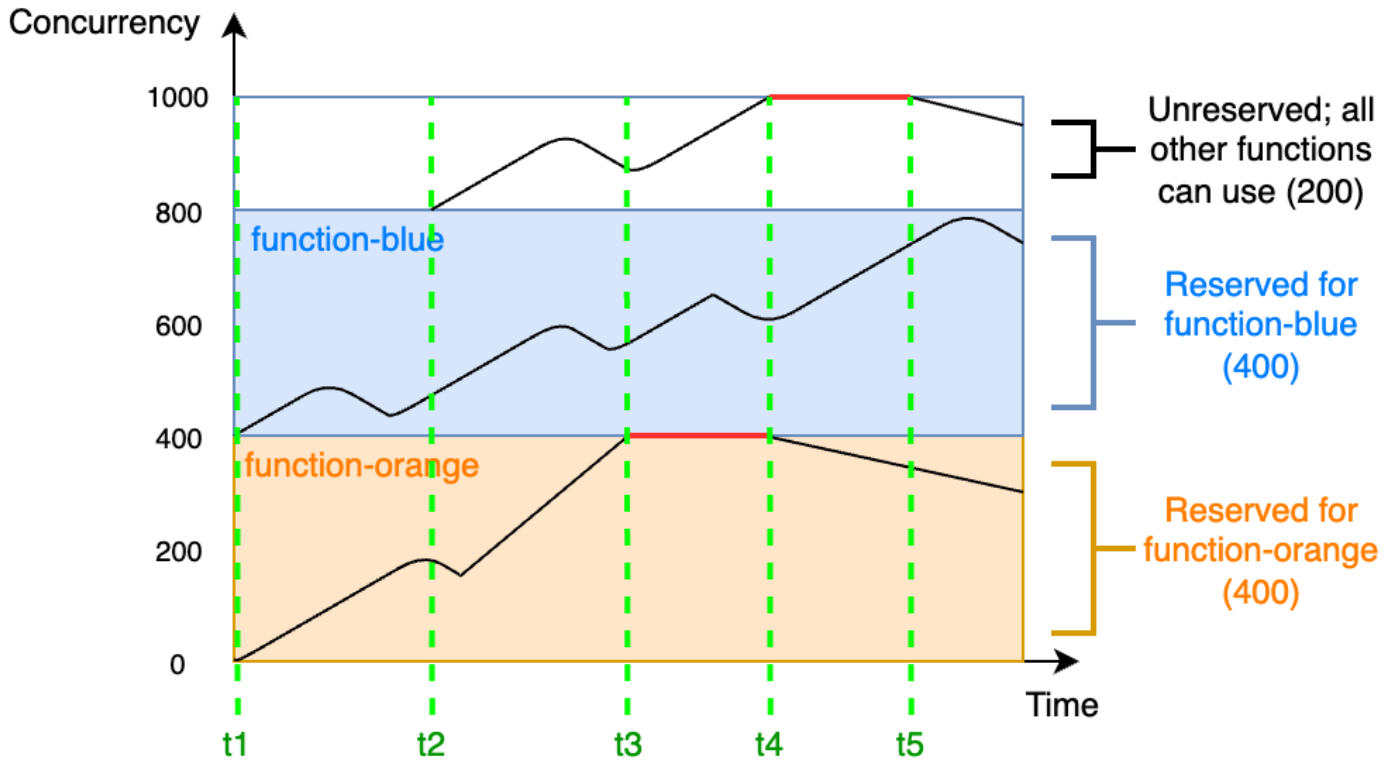
예약된 동시성

함수가 언제든지 일정량의 동시성을 사용할 수 있도록 하려면 예약된 동시성을 사용합니다.

예약된 동시성은 함수에 할당하려는 최대 동시 인스턴스 수입니다. 예약된 동시성을 한 함수에 전용으로 사용하면 다른 함수는 해당 동시성을 사용할 수 없습니다. 즉, 예약된 동시성 설정은 다른 함수가 사용할 수 있는 동시성 풀에 영향을 미칠 수 있습니다. 예약된 동시성이 없는 함수는 예약되지 않은 동시성의 나머지 풀을 공유합니다.

예약된 동시성을 구성하면 전체 계정 동시성 한도에 반영됩니다. 함수에 대해 예약된 동시성을 구성하는 데는 요금이 부과되지 않습니다.

예약된 동시성을 더 잘 이해하려면 다음 다이어그램을 살펴보세요.



이 다이어그램에서 이 리전의 모든 함수에 대한 계정 동시성 한도는 기본 한도인 1,000입니다. `function-blue`와 `function-orange`라는 두 개의 중요한 함수가 있고 일상적으로 호출량이 많을 것으로 예상된다고 가정해 보겠습니다. `function-blue`에 예약된 동시성 400단위를 제공하고 `function-orange`에 예약된 동시성 400단위를 제공하기로 결정합니다. 이 예에서는 계정의 다른 모든 함수가 나머지 200단위의 예약되지 않은 동시성을 공유해야 합니다.

이 다이어그램에는 주목해야 할 다섯 가지 사항이 있습니다.

- `t1`에 `function-orange`와 `function-blue`는 둘 다 요청을 수신하기 시작합니다. 각 함수는 예약된 동시성 단위의 할당된 부분을 사용하기 시작합니다.
- `t2`에 `function-orange`와 `function-blue`는 꾸준히 더 많은 요청을 수신합니다. 그와 동시에, 사용자가 다른 Lambda 함수를 배포하고 해당 함수가 요청 수신을 시작합니다. 이러한 다른 함수에는 예약된 동시성을 할당하지 않습니다. 해당 함수들은 나머지 200단위의 예약되지 않은 동시성을 사용하기 시작합니다.
- `t3`에 `function-orange`는 최대 동시성 400에 도달합니다. 계정의 다른 곳에 사용하지 않은 동시성이 있지만 `function-orange`는 거기에 액세스할 수 없습니다. 빨간색 선은 `function-orange`에서 제한이 발생하고 있음을 나타내며 Lambda가 요청을 삭제할 수 있습니다.

- t4에는 function-orange가 요청을 더 적게 받기 시작하고 더 이상 제한되지 않습니다. 하지만 다른 함수에서는 트래픽이 급증하여 제한이 시작됩니다. 계정의 다른 곳에 사용하지 않은 동시성이 있지만 이 다른 함수들은 거기에 액세스할 수 없습니다. 빨간색 선은 다른 함수에서 제한이 발생하고 있음을 나타냅니다.
- t5에는 다른 함수들이 요청을 더 적게 받기 시작하고 더 이상 제한되지 않습니다.

이 예에서 동시성을 예약하면 다음과 같은 효과가 있음을 알 수 있습니다.

- 함수는 계정의 다른 함수와 독립적으로 확장할 수 있습니다. 동일한 리전에 있는 모든 계정의 함수 중 예약된 동시성이 없는 함수는 예약되지 않은 동시성 풀을 공유합니다. 예약된 동시성이 없으면 다른 함수는 사용 가능한 모든 동시성을 다 사용하게 될 수 있습니다. 이렇게 하면 필요할 때 중요한 함수가 스케일 업되지 않습니다.
- 함수는 제어 범위를 벗어나 스케일 아웃될 수 없습니다. 예약된 동시성은 함수의 최대 동시성을 제한합니다. 즉, 함수는 다른 함수에 예약된 동시성 또는 예약되지 않은 풀의 동시성을 사용할 수 없습니다. 동시성을 예약하여 함수가 계정에서 사용 가능한 모든 동시성을 다 사용하지 못하게 하거나 다운스트림 리소스를 오버로드하지 못하게 할 수 있습니다.
- 계정에서 사용 가능한 모든 동시성을 사용하지 못할 수도 있습니다. 동시성 예약은 계정 동시성 한도에 반영되지만, 이는 다른 함수가 예약된 동시성 청크를 사용할 수 없다는 의미이기도 합니다. 함수가 예약한 동시성을 모두 사용하지 않으면 사실상 해당 동시성을 낭비하고 있는 셈입니다. 이는 계정의 다른 함수가 낭비되는 동시성을 활용했을 때 이점이 있는 경우가 아니라면 문제가 되지 않습니다.

함수에 대해 예약된 동시성 설정을 관리하는 방법에 대해 알아보려면 함수에 대해 예약된 동시성 구성을 참조하세요.

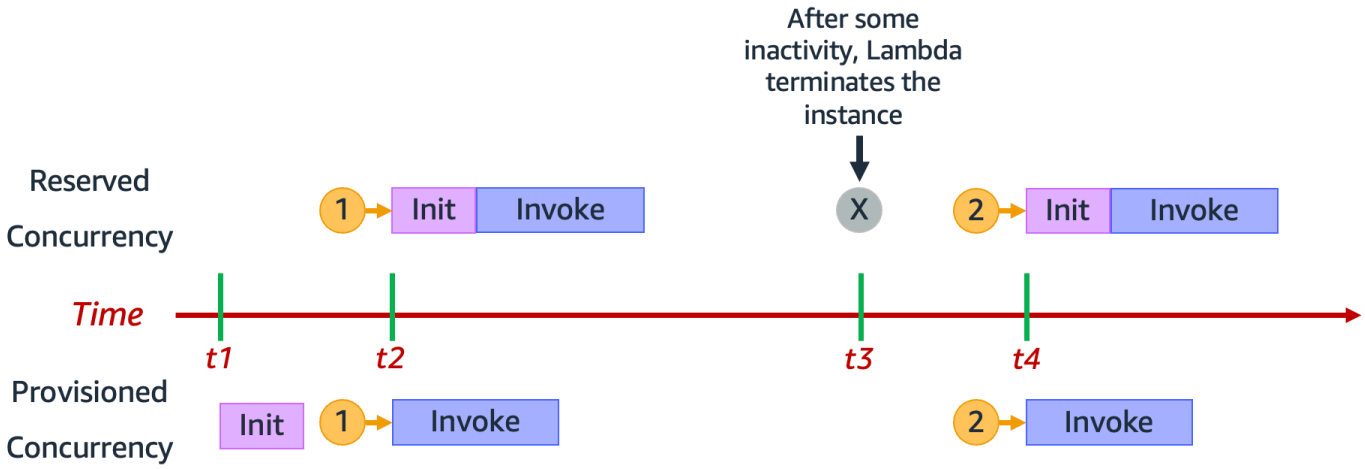
프로비저닝된 동시성

예약된 동시성을 사용하여 Lambda 함수에 예약된 최대 실행 환경 수를 정의합니다. 하지만 이러한 환경 중 어느 것도 사전 초기화된 상태로 제공되지 않습니다. 따라서 함수를 호출하는 데 새 환경을 사용하려면 먼저 Lambda가 새 환경을 초기화해야 하기 때문에 함수 호출 시간이 더 오래 걸릴 수 있습니다. 호출을 수행하기 위해 Lambda가 새 환경을 초기화해야 하는 경우 이를 콜드 스타트라고 합니다. 콜드 스타트를 해결하기 위해 프로비저닝된 동시성을 사용할 수 있습니다.

프로비저닝된 동시성은 함수에 할당하려는 사전 초기화된 실행 환경의 수입니다. 함수에 대해 프로비저닝된 동시성을 설정하면 Lambda는 함수의 요청에 즉시 응답할 준비가 되도록 해당하는 수의 실행 환경을 초기화합니다.

> **Note**
>
> 프로비저닝된 동시성을 사용하면 계정에 추가 요금이 부과됩니다. Java 11 또는 Java 17 런타임 작업 시 추가 비용 없이 Lambda SnapStart를 사용하여 콜드 스타트 문제를 완화할 수도 있습니다. SnapStart는 실행 환경의 캐시된 스냅샷을 사용하여 시작 성능을 크게 향상시킵니다. 동일한 함수 버전에 SnapStart와 프로비저닝된 동시성을 모두 사용할 수는 없습니다. SnapStart 기능, 제한 사항 및 지원되는 리전에 대한 자세한 내용은 Lambda SnapStart를 사용하여 시작 성능 개선을 참조하세요.

프로비저닝된 동시성을 사용하는 경우에도 Lambda는 여전히 백그라운드에서 실행 환경을 재활용합니다. 하지만 Lambda는 항상 사전 초기화된 환경의 수가 함수의 프로비저닝된 동시성 설정 값과 같도록 합니다. 이 동작은 Lambda가 일정 기간 동안 비활성 상태인 환경을 완전히 종료할 수 있는 예약된 동시성과 다릅니다. 다음 다이어그램은 예약된 동시성을 사용하여 함수를 구성할 경우 단일 실행 환경의 수명 주기를 프로비저닝된 동시성과 비교하여 이를 보여줍니다.

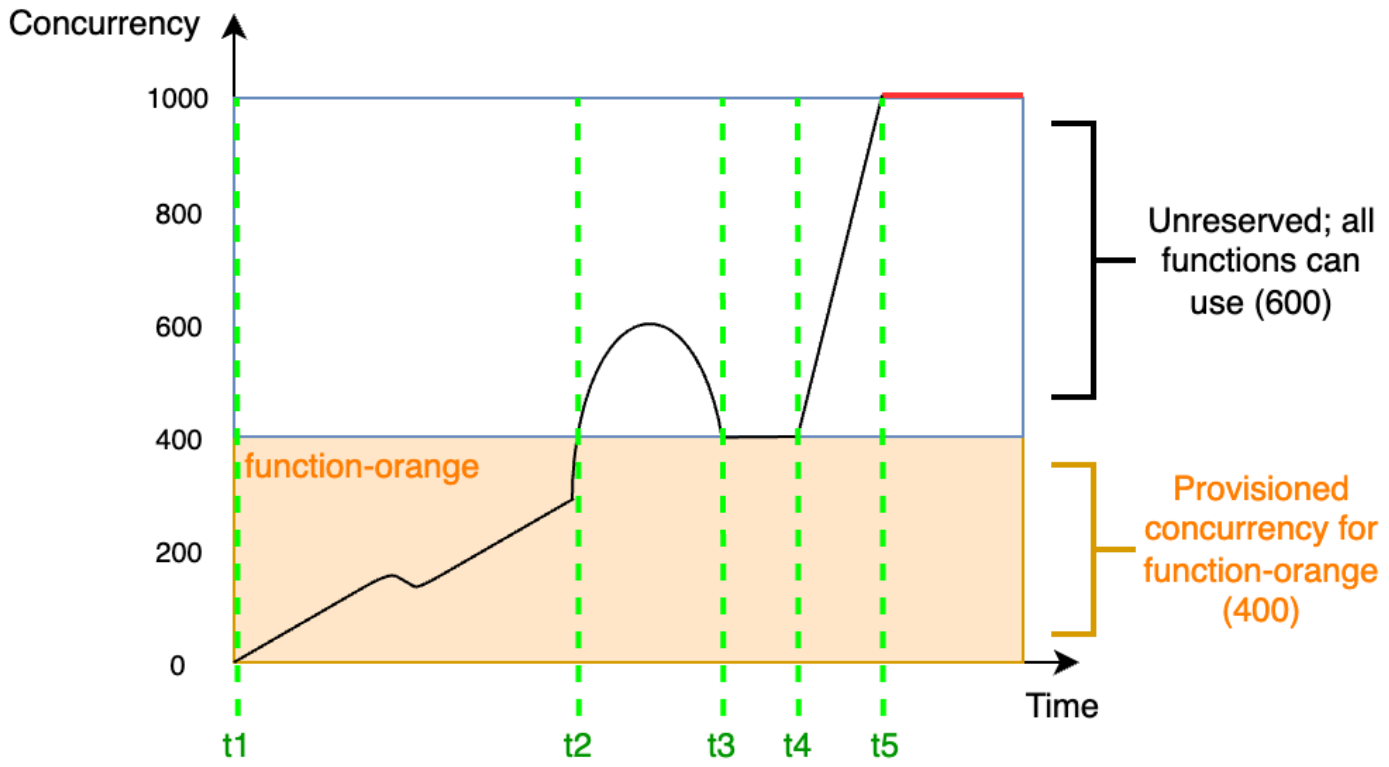


이 다이어그램에는 주목해야 할 4가지 사항이 있습니다.

Time	예약된 동시성	프로비저닝된 동시성
t1	아무 일도 일어나지 않습니다.	Lambda가 실행 환경 인스턴스를 사전 초기화합니다.
t2	요청 1이 수신됩니다. Lambda가 새 실행 환경 인스턴스를 초기화해야 합니다.	요청 1이 수신됩니다. Lambda가 사전 초기화된 환경 인스턴스를 사용합니다.

Time	예약된 동시성	프로비저닝된 동시성
t3	일정 시간 동안 비활성 상태로 있으면 Lambda가 해당 활성 환경 인스턴스를 종료합니다.	아무 일도 일어나지 않습니다.
t4	요청 2가 수신됩니다. Lambda가 새 실행 환경 인스턴스를 초기화해야 합니다.	요청 2가 수신됩니다. Lambda가 사전 초기화된 환경 인스턴스를 사용합니다.

프로비저닝된 동시성을 더 잘 이해하려면 다음 다이어그램을 살펴보세요.

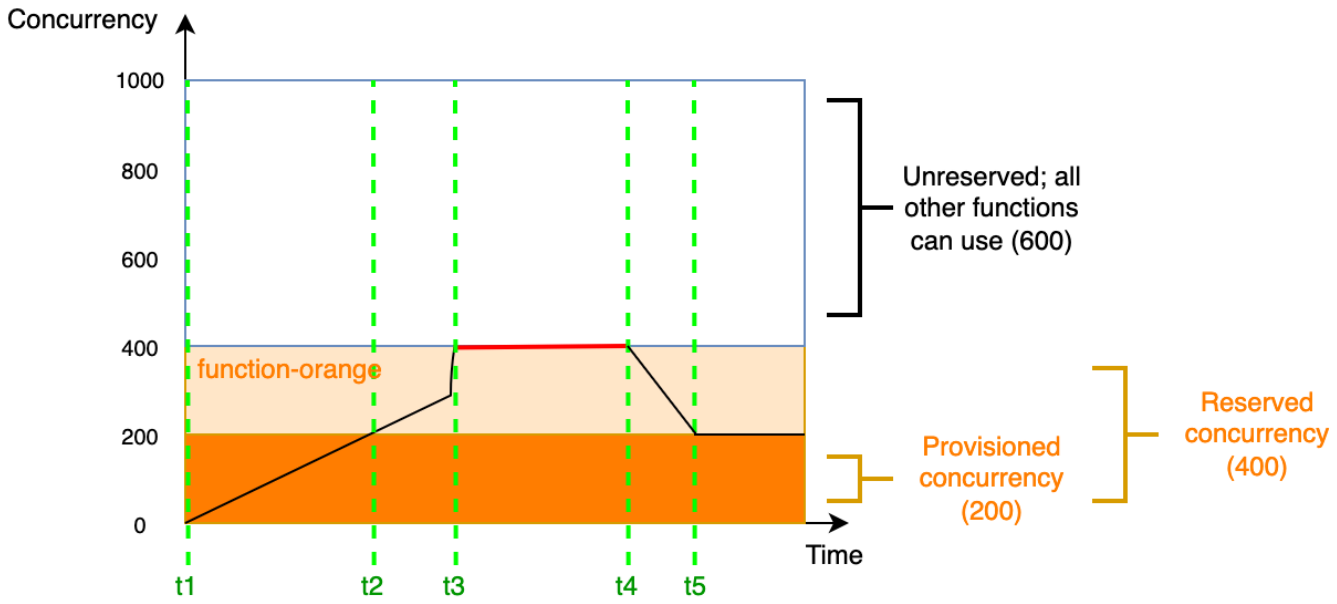


이 다이어그램의 계정 동시성 한도는 1,000입니다. 400단위의 프로비저닝된 동시성을 `function-orange`에 제공하기로 결정합니다. `function-orange`를 포함한 계정의 모든 함수는 예약되지 않은 나머지 동시성 600단위를 사용할 수 있습니다.

이 다이어그램에는 주목해야 할 다섯 가지 사항이 있습니다.

- t1에 function-orange가 요청을 수신하기 시작합니다. Lambda가 400개의 실행 환경 인스턴스를 사전 초기화했으므로 function-orange는 즉시 호출을 처리할 수 있습니다.
- t2에 function-orange가 동시 요청 400개에 도달합니다. 따라서 function-orange의 프로비저닝된 동시성이 소진됩니다. 하지만 아직 예약되지 않은 동시성을 사용할 수 있으므로, Lambda는 이를 사용하여 function-orange에 대한 추가 요청을 처리할 수 있습니다(제한 없음). Lambda는 이러한 요청을 처리하기 위해 새 인스턴스를 생성해야 하며 함수에서 콜드 스타트 지연 시간이 발생할 수 있습니다.
- t3에는 function-orange에 대한 트래픽이 잠시 급증한 후 400개의 동시 요청으로 돌아갑니다. Lambda는 다시 콜드 스타트 지연 없이 모든 요청을 처리할 수 있게 됩니다.
- t4에 계정의 함수에서 트래픽이 급증합니다. 이 버스트는 function-orange 또는 계정의 다른 함수에서 발생한 것일 수 있습니다. Lambda는 예약되지 않은 동시성을 사용하여 이러한 요청을 처리합니다.
- t5에 계정의 함수가 최대 동시성 한도인 1,000단위에 도달하여 제한이 발생합니다.

이전 예에서는 오직 프로비저닝된 동시성만 고려했습니다. 실제 상황에서는 함수에 대해 프로비저닝된 동시성과 예약된 동시성을 모두 설정할 수 있습니다. 주중에 일정한 간접 호출 로드를 처리하지만 주말에 정기적으로 트래픽이 급증하는 함수가 있는 경우 이 방법을 사용할 수 있습니다. 이 경우 프로비저닝된 동시성을 사용하여 평일에 요청을 처리할 기존 환경 수를 설정하고, 예약된 동시성을 사용하여 주말에 급증하는 트래픽을 처리할 수 있습니다. 다음 다이어그램을 살펴보세요.



이 다이어그램에서 function-orange에 대해 프로비저닝된 동시성 200단위와 예약된 동시성 400단위를 구성한다고 가정해 보겠습니다. 예약된 동시성을 구성했으므로, function-orange는 예약되지 않은 동시성 600단위를 사용할 수 없습니다.

이 다이어그램에는 주목해야 할 다섯 가지 사항이 있습니다.

- t1에 function-orange가 요청을 수신하기 시작합니다. Lambda가 200개의 실행 환경 인스턴스를 사전 초기화했으므로 function-orange는 즉시 호출을 처리할 수 있습니다.
- t2에는 function-orange가 프로비저닝된 동시성을 모두 사용합니다. function-orange는 예약된 동시성을 사용하여 요청을 계속 처리할 수 있지만 요청에서 콜드 스타트 지연 시간이 발생할 수 있습니다.
- t3에 function-orange가 동시 요청 400개에 도달합니다. 따라서 function-orange는 예약된 동시성을 모두 사용합니다. function-orange는 예약되지 않은 동시성을 사용할 수 없으므로 요청이 제한되기 시작합니다.
- t4에는 function-orange가 요청을 더 적게 받기 시작하고 더 이상 제한되지 않습니다.
- t5에는 function-orange의 동시 요청이 200개로 감소하므로 모든 요청이 다시 프로비저닝된 동시성을 사용할 수 있습니다(즉, 콜드 스타트 지연 시간 없음).

예약된 동시성과 프로비저닝된 동시성 수는 모두 계정 동시성 한도와 리전별 할당량에 반영됩니다. 즉, 예약된 동시성과 프로비저닝된 동시성을 할당하면 다른 함수가 사용할 수 있는 동시성 풀에 영향을 미칠 수 있습니다. 프로비저닝된 동시성을 구성하면 AWS 계정에 요금이 부과됩니다.

> **Note**
>
> 함수의 버전과 별칭에서 프로비저닝된 동시성의 양이 함수의 예약된 동시성까지 더해지면 그럼 모든 간접 호출은 프로비저닝된 동시성에서 실행됩니다. 이 구성은 게시되지 않은 버전의 함수($LATEST)를 조절하여 실행을 금지하는 효과가 있습니다. 함수에 대해 예약된 동시성보다 더 많은 프로비저닝된 동시성을 할당할 수 없습니다.

함수에 대해 예약된 동시성 설정을 관리하려면 함수에 대해 프로비저닝된 동시성 구성을 참조하세요. 일정 또는 애플리케이션 활용도를 기준으로 프로비저닝된 동시성 스케일을 관리하려면 Application Auto Scaling을 사용하여 프로비저닝된 동시성 관리 자동화을 참고하세요.

Lambda에서 프로비저닝된 동시성을 할당하는 방법

프로비저닝된 동시성이 구성 직후에 온라인 상태가 되는 것은 아닙니다. Lambda는 1~2분의 준비 시간을 가진 후에 프로비저닝된 동시성 할당을 시작합니다. 각 기능에 대해 Lambda는 AWS 리전에 관계없이 매분 최대 6,000개의 실행 환경을 프로비저닝할 수 있습니다. 이는 함수의 동시성 확장 속도와 완전히 동일합니다.

프로비저닝된 동시성 할당 요청을 제출하면 Lambda가 할당을 완전히 마칠 때까지 해당 환경에 액세스할 수 없습니다. 예를 들어 5,000개의 프로비저닝된 동시성을 요청하는 경우, Lambda가 5,000개의 실행 환경 할당을 완전히 완료할 때까지 어떤 요청도 프로비저닝된 동시성을 사용할 수 없습니다.

예약된 동시성과 프로비저닝된 동시성 비교

다음 표는 예약된 동시성과 프로비저닝된 동시성을 요약하고 비교한 것입니다.

주제	예약된 동시성	프로비저닝된 동시성
정의	함수의 최대 실행 환경 인스턴스 수입니다.	함수의 사전 프로비저닝된 실행 환경 인스턴스 수를 설정합니다.
프로비저닝 동작	Lambda가 온디맨드로 새 인스턴스를 프로비저닝합니다.	Lambda가 인스턴스를 사전 프로비저닝합니다(즉, 함수가 요청 수신을 시작하기 전).
콜드 스타트 동작	Lambda가 온디맨드로 새 인스턴스를 생성해야 하므로 콜드 스타트 지연 시간이 발생할 수 있습니다.	Lambda가 온디맨드로 인스턴스를 생성할 필요가 없으므로 콜드 스타트 지연 시간이 가능하지 않습니다.
제한 동작	예약된 동시성 한도에 도달하면 함수가 제한됩니다.	예약된 동시성이 설정되지 않은 경우: 프로비저닝된 동시성 제한에 도달하면 함수가 예약되지 않은 동시성을 사용합니다. 예약된 동시성이 설정된 경우: 예약된 동시성 한도에 도달하면 함수가 제한됩니다.

주제	예약된 동시성	프로비저닝된 동시성
설정하지 않은 경우 기본 동작	함수가 계정에서 사용 가능한 예약되지 않은 동시성을 사용합니다.	Lambda가 인스턴스를 사전 프로비저닝하지 않습니다. 예약된 동시성이 설정되지 않은 경우: 함수가 계정에서 사용 가능한 예약되지 않은 동시성을 사용합니다. 예약된 동시성이 설정된 경우: 함수가 예약된 동시성을 사용합니다.
요금	추가 요금이 없습니다.	추가 요금이 발생합니다.

동시성 할당량

Lambda가 리전의 모든 함수에 걸쳐 사용 가능한 총 동시성 수량에 대한 할당량을 설정합니다. 이 할당량은 두 가지 수준으로 존재합니다.

- 계정 수준에서 함수는 기본적으로 최대 1,000단위의 동시성을 가질 수 있습니다. 이 한도를 늘리려면 Service Quotas 사용 설명서에서 할당량 증가 요청을 참조하세요.
- 함수 수준에서 기본적으로 모든 함수에 걸쳐 최대 900단위의 동시성을 예약할 수 있습니다. 총 계정 동시성 한도에 관계없이 Lambda는 명시적으로 동시성을 예약하지 않는 함수에 대해 항상 100개의 동시성 단위를 예약합니다. 예를 들어 계정 동시성 한도를 2,000단위로 늘린 경우 그럼 함수 수준에서 최대 1,900단위의 동시성을 예약할 수 있습니다.

현재 계정 수준의 동시성 할당량을 확인하려면 AWS Command Line Interface(AWS CLI)을 사용하여 다음 명령을 실행합니다.

```
aws lambda get-account-settings
```

그러면 다음과 같은 결과가 표시됩니다.

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
```

```
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

ConcurrentExecutions는 총 계정 수준 동시성 할당량입니다.
UnreservedConcurrentExecutions는 여전히 함수에 할당할 수 있는 예약된 동시성의 양입니다.

함수가 더 많은 요청을 수신하면 Lambda는이 계정 동시성 할당량에 도달할 때까지 이러한 요청을 처리할 수 있도록 실행 환경 수를 자동으로 스케일 업합니다. 그러나 갑작스러운 트래픽 폭증에 대응한 오버스케일링을 방지하기 위해 Lambda는 함수의 스케일 속도를 제한합니다. 이 동시성 확장 속도는 요청 증가에 대응하여 계정의 기능이 스케일 인할 수 있는 최대 속도입니다. (즉, Lambda가 새 실행 환경을 얼마나 빨리 생성할 수 있는지를 의미합니다.) 동시성 확장 속도는 함수에 사용할 수 있는 총 동시성의 양인 계정 수준 동시성 한도와는 다릅니다.

각 AWS 리전에서, 각 함수에 대한 동시성 스케일 속도는 10초당 1,000개의 실행 환경 인스턴스입니다. 즉, Lambda는 10초당 최대 1,000개의 추가 실행 환경 인스턴스를 각 함수에 할당할 수 있습니다.

일반적으로 이 제한 사항에 대해 걱정할 필요는 없습니다. Lambda의 스케일 속도는 대부분의 사용 사례에 충분합니다.

동시성 확장 속도가 함수 수준 제한이라는 점은 중요합니다. 즉, 계정의 각 함수가 다른 함수와 독립적으로 스케일할 수 있다는 의미입니다.

스케일 동작에 대한 자세한 내용은 Lambda 조정 동작을 참조하세요.

함수에 대해 예약된 동시성 구성

Lambda에서 동시성은 함수가 동시에 처리하는 전송 중인 요청의 수입니다. 두 가지 유형의 동시성 제어를 사용할 수 있습니다.

- 예약된 동시성 - 함수에 할당된 최대 동시 인스턴스 수를 나타냅니다. 한 함수가 동시성을 예약하면 다른 함수는 해당 동시성을 사용할 수 없습니다. 예약된 동시성은 가장 중요한 함수가 수신 요청을 처리하기에 충분한 동시성을 항상 확보하도록 하는 데 유용합니다. 함수에 예약된 동시성을 구성하는 경우 추가 요금이 부과되지 않습니다.
- 프로비저닝된 동시성 - 함수에 할당된 사전 초기화된 실행 환경의 수입니다. 이러한 실행 환경은 수신 함수 요청에 즉시 응답할 수 있도록 준비되어 있습니다. 프로비저닝된 동시성은 함수에 대해 콜드 스타트 지연 시간을 줄이는 데 유용합니다. 프로비저닝된 동시성을 구성하는 경우 AWS 계정에 추가 요금이 부과됩니다.

이 주제에서는 예약된 동시성을 관리하고 구성하는 방법에 대해 자세히 설명합니다. 이 두 가지 유형의 동시성 제어에 대한 개념적 개요는 예약된 동시성 및 프로비저닝된 동시성을 참조하세요. 프로비저닝된 동시성 구성에 대한 자세한 내용은 the section called “프로비저닝된 동시성 구성” 섹션을 참조하세요.

> **Note**
>
> Amazon MQ 이벤트 소스 매핑에 연결된 Lambda 함수는 기본 최대 동시성을 지원합니다. Apache Active MQ의 경우 최대 동시 인스턴스의 수는 5입니다. Rabbit MQ의 경우 최대 동시 인스턴스의 수는 1입니다. 함수에 예약 또는 프로비저닝된 동시성을 설정해도 이 한도는 변경되지 않습니다. Amazon MQ를 사용할 때 기본 최대 동시성 증가를 요청하려면 AWS Support에 문의하십시오.

Sections

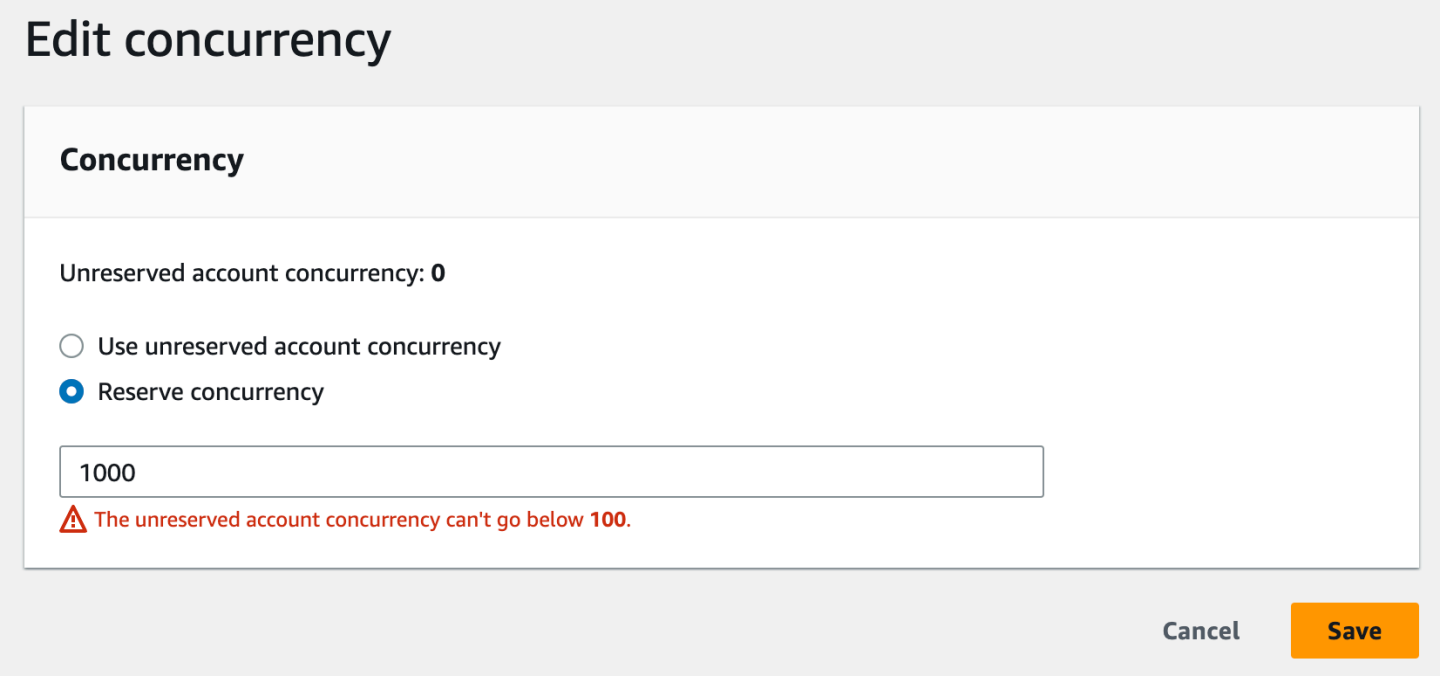
예약된 동시성 구성

Lambda 콘솔 또는 Lambda API를 사용하여 함수에 대해 예약된 동시성 설정을 구성할 수 있습니다.

함수에 대해 동시성 예약(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 동시성을 예약하려는 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 동시성(Concurrency)을 선택합니다.
4. 동시성에서 편집을 선택합니다.
5. 동시성 예약을 선택합니다. 함수에 예약할 동시성의 크기를 입력합니다.
6. Save(저장)를 선택합니다.

예약되지 않은 계정 동시성 값에서 100을 뺀 값까지 예약할 수 있습니다. 나머지 100단위의 동시성은 예약된 동시성을 사용하지 않는 함수용입니다. 예를 들어, 계정의 동시성 한도가 1,000인 경우 1,000단위의 동시성을 모두 단일 함수에 예약할 수 없습니다.



함수에 대해 동시성을 예약하면 다른 함수가 사용할 수 있는 동시성 풀에 영향이 있습니다. 예를 들어 `function-a`에 대해 100단위의 동시성을 예약하는 경우 `function-a`가 예약된 동시성 100단위를 모두 사용하지 않더라도 계정의 다른 함수는 나머지 900단위의 동시성을 공유해야 합니다.

함수를 의도적으로 제한하려면 예약된 동시성을 0으로 설정합니다. 이렇게 하면 제한을 제거할 때까지 함수가 이벤트를 처리하지 않습니다.

Lambda API를 사용하여 예약된 동시성을 구성하려면 다음 API 작업을 사용합니다.

- PutFunctionConcurrency

- GetFunctionConcurrency
- DeleteFunctionConcurrency

예를 들어, AWS Command Line Interface(CLI)를 사용하여 예약된 동시성을 구성하려면 `put-function-concurrency` 명령을 사용합니다. 다음 명령은 `my-function`이라는 함수에 대해 동시성 100단위를 예약합니다.

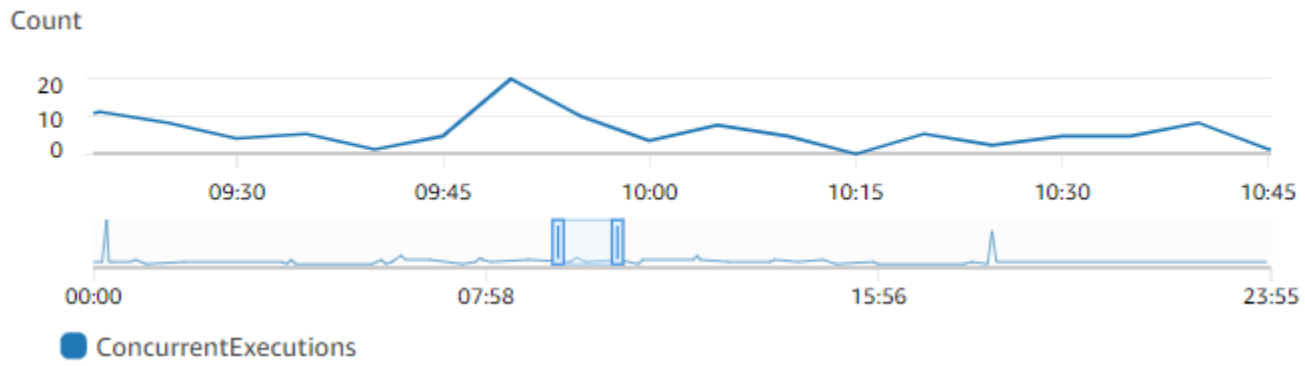
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

그러면 다음과 같은 결과가 표시됩니다.

```
{
    "ReservedConcurrentExecutions": 100
}
```

함수에 필요한 예약된 동시성 정확히 예측

함수가 현재 트래픽을 처리하고 있는 경우 CloudWatch 지표를 사용하여 해당 함수의 동시성 지표를 손쉽게 확인할 수 있습니다. 구체적으로, `ConcurrentExecutions` 지표는 계정의 각 함수에 대한 동시 호출 수를 보여줍니다.



이전 그래프는 이 함수가 주어진 시간에 평균 5~10개의 동시 요청을 처리하고 보통 하루에 최대 20개의 요청을 처리함을 보여줍니다. 계정에 다른 함수가 많이 있다고 가정해 보겠습니다. 이 함수가 애플리케이션에 중요하고 요청을 삭제하지 않으려면 예약된 동시성 설정으로 20 이상의 숫자를 사용합니다.

또는 다음 수식을 사용하여 동시성을 계산할 수도 있습니다.

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

초당 평균 요청과 초 단위의 평균 요청 시간을 곱하면 예약해야 하는 동시성 양을 대략적으로 예측할 수 있습니다. `Invocation` 지표를 사용하여 초당 평균 요청 수를 추정하고 `Duration` 지표를 사용하여 평균 요청 지속 시간을 초 단위로 추정할 수 있습니다. 자세한 내용은 Lambda 함수 지표 작업 섹션을 참조하세요.

함수에 대해 프로비저닝된 동시성 구성

Lambda에서 동시성은 함수가 동시에 처리하는 전송 중인 요청의 수입니다. 두 가지 유형의 동시성 제어를 사용할 수 있습니다.

- 예약된 동시성 - 함수에 할당된 최대 동시 인스턴스 수를 나타냅니다. 한 함수가 동시성을 예약하면 다른 함수는 해당 동시성을 사용할 수 없습니다. 예약된 동시성은 가장 중요한 함수가 수신 요청을 처리하기에 충분한 동시성을 항상 확보하도록 하는 데 유용합니다. 함수에 예약된 동시성을 구성하는 경우 추가 요금이 부과되지 않습니다.
- 프로비저닝된 동시성 - 함수에 할당된 사전 초기화된 실행 환경의 수입니다. 이러한 실행 환경은 수신 함수 요청에 즉시 응답할 수 있도록 준비되어 있습니다. 프로비저닝된 동시성은 함수에 대해 콜드 스타트 지연 시간을 줄이는 데 유용합니다. 프로비저닝된 동시성을 구성하는 경우 AWS 계정에 추가 요금이 부과됩니다.

이 주제에서는 프로비저닝된 동시성을 관리하고 구성하는 방법에 대해 자세히 설명합니다. 이 두 가지 유형의 동시성 제어에 대한 개념적 개요는 예약된 동시성 및 프로비저닝된 동시성을 참조하세요. 예약된 동시성 구성에 대한 자세한 내용은 the section called “예약된 동시성 구성” 섹션을 참조하세요.

> **Note**
>
> Amazon MQ 이벤트 소스 매핑에 연결된 Lambda 함수는 기본 최대 동시성을 지원합니다. Apache Active MQ의 경우 최대 동시 인스턴스의 수는 5입니다. Rabbit MQ의 경우 최대 동시 인스턴스의 수는 1입니다. 함수에 예약 또는 프로비저닝된 동시성을 설정해도 이 한도는 변경되지 않습니다. Amazon MQ를 사용할 때 기본 최대 동시성 증가를 요청하려면 AWS Support에 문의하십시오.

Sections

프로비저닝된 동시성 구성

Lambda 콘솔 또는 Lambda API를 사용하여 함수에 대해 프로비저닝된 동시성 설정을 구성할 수 있습니다.

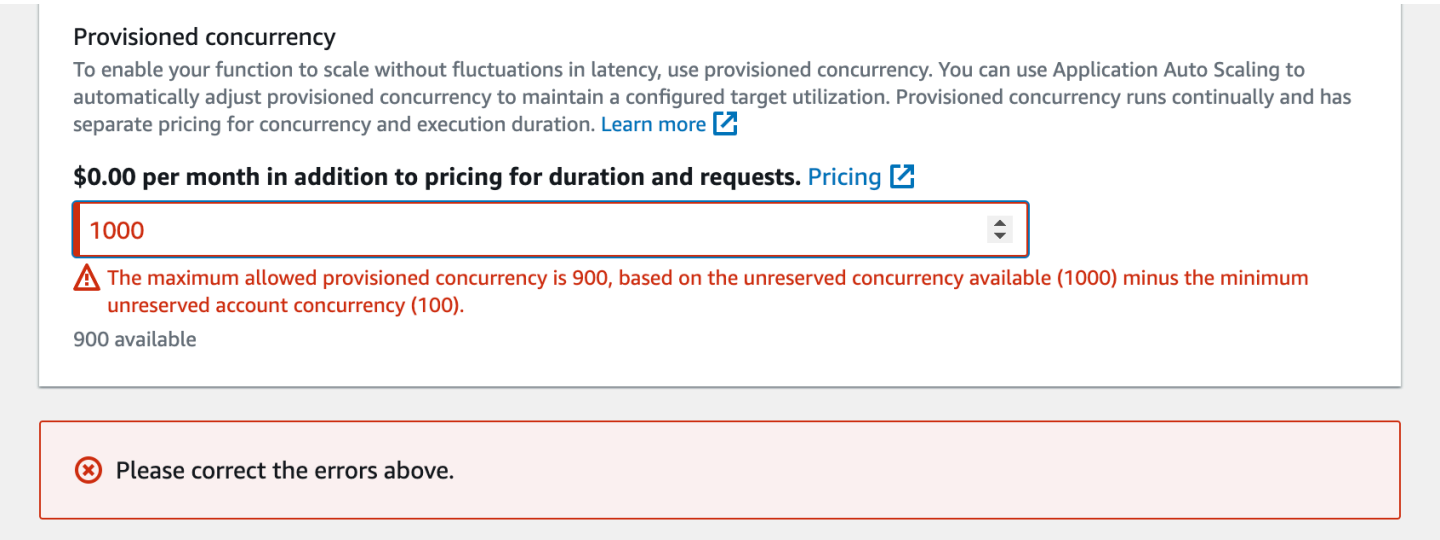
함수에 대해 프로비저닝된 동시성 할당(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 프로비저닝된 동시성을 할당하려는 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 동시성(Concurrency)을 선택합니다.
4. 프로비저닝된 동시성 구성(Provisioned concurrency configurations)에서 구성 추가(Add configuration)를 선택합니다.
5. 한정자 유형과 별칭 또는 버전을 선택합니다.

> **Note**
>
> 함수의 $LATEST 버전에서는 프로비저닝된 동시성을 사용할 수 없습니다.
> 함수에 이벤트 소스가 있는 경우 이벤트 소스가 올바른 함수 별칭 또는 버전을 가리키는지 확인합니다. 그렇지 않으면 함수가 프로비저닝된 동시성 환경을 사용하지 않습니다.

6. 프로비저닝된 동시성 아래에 숫자를 입력합니다. Lambda가 월별 예상 비용을 제공합니다.
7. Save(저장)를 선택합니다.

계정에서 예약되지 않은 계정 동시성(- 100)까지 구성할 수 있습니다. 나머지 100단위의 동시성은 예약된 동시성을 사용하지 않는 함수용입니다. 예를 들어, 계정의 동시성 한도가 1,000인데 다른 함수에 예약되거나 프로비저닝된 동시성을 할당하지 않은 경우 단일 함수에 대해 최대 프로비저닝된 동시성 900단위를 구성할 수 있습니다.



함수에 프로비저닝된 동시성을 구성하면 다른 함수가 사용할 수 있는 동시성 풀에 영향이 줍니다. 예를 들어 `function-a`에 100의 프로비저닝된 동시성 단위를 구성하는 경우 계정의 다른 함수는 남은 900의 동시성 단위를 공유해야 합니다. `function-a`에서 100의 모든 단위를 사용하지 않더라도 마찬가지입니다.

동일한 함수에 예약된 동시성과 프로비저닝된 동시성을 모두 할당할 수 있습니다. 이 경우 프로비전된 동시성은 예약된 동시성을 초과할 수 없습니다.

이 제한은 함수 버전에도 적용됩니다. 특정 함수 버전에 할당할 수 있는 최대 프로비저닝된 동시성은 함수의 예약된 동시성에서 다른 함수 버전의 프로비저닝된 동시성을 뺀 값과 같습니다.

Lambda API를 사용하여 프로비저닝된 동시성을 구성하려면 다음 API 작업을 사용합니다.

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

예를 들어, AWS Command Line Interface(CLI)를 사용하여 프로비저닝된 동시성을 구성하려면 `put-provisioned-concurrency-config` 명령을 사용합니다. 다음 명령은 `my-function`이라는 함수의 `BLUE` 별칭에 대해 100단위의 프로비저닝된 동시성을 할당합니다.

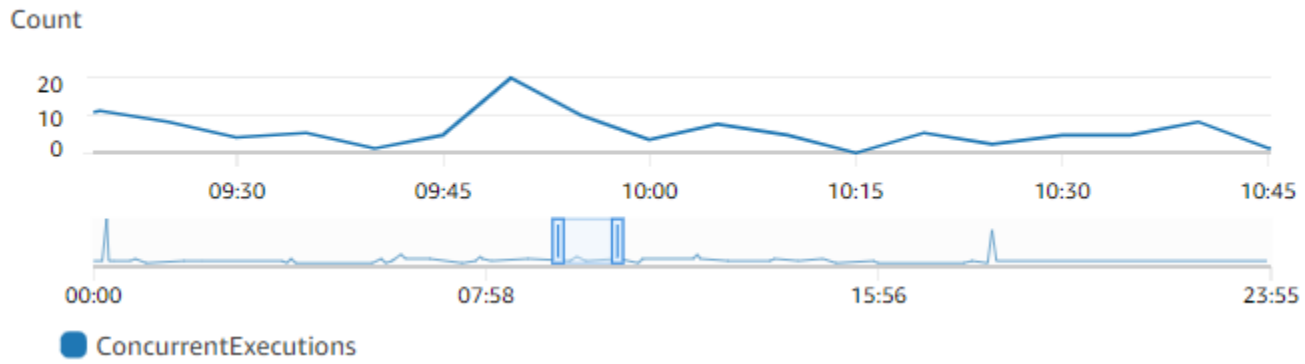
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

그러면 다음과 같은 결과가 표시됩니다.

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

함수에 필요한 프로비저닝된 동시성 정확히 예측

CloudWatch 지표를 사용하여 모든 활성 함수의 동시성 지표를 볼 수 있습니다. 구체적으로, `ConcurrentExecutions` 지표는 계정의 함수에 대한 동시 간접 호출 수를 보여줍니다.



이전 그래프는 이 함수가 주어진 시간에 평균 5~10개의 동시 요청을 처리하고 최대 20개의 요청을 처리함을 보여줍니다. 계정에 다른 함수가 많이 있다고 가정해 보겠습니다. 이 함수가 애플리케이션에 중요하고 호출할 때마다 지연 시간이 짧은 응답이 필요한 경우 프로비저닝된 동시성 단위로 최소 20을 구성합니다.

다음 수식을 사용하여 동시성을 계산할 수도 있습니다.

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

필요한 동시성을 예측하려면 초당 평균 요청 수에 평균 요청 시간(초 단위)을 곱해야 합니다. `Invocation` 지표를 사용하여 초당 평균 요청 수를 추정하고 `Duration` 지표를 사용하여 평균 요청 지속 시간을 초 단위로 추정할 수 있습니다.

프로비저닝된 동시성을 구성할 때 Lambda는 일반적으로 함수에 필요한 동시성의 양 외에 10% 버퍼를 추가하도록 제안합니다. 예를 들어, 함수의 최대 동시 요청 수가 보통 200개인 경우 프로비저닝된 동시성을 220으로 설정합니다(200개의 동시 요청 + 10% = 프로비저닝된 동시성 220).

프로비저닝된 동시성 사용 시 함수 코드 최적화

프로비저닝된 동시성을 사용하는 경우 짧은 지연 시간을 위해 함수 코드를 재구성하여 최적화하는 것이 좋습니다. 프로비저닝된 동시성을 사용하는 함수의 경우 Lambda는 할당 중 모든 초기화 코드(예: 라이브러리 로드 및 클라이언트 인스턴스화)를 실행합니다. 따라서 실제 함수 간접 호출 중에 지연 시간에 영향이 없도록 초기화를 최대한 많이 기본 함수 핸들러 외부로 이동하는 것이 좋습니다. 반대로 기본 핸들러 코드 내부에서 라이브러리를 초기화하거나 클라이언트를 인스턴스화하면 프로비저닝된 동시성 사용 여부에 관계없이 간접적으로 호출할 때마다 함수가 이를 실행해야 합니다.

온디맨드 간접 호출의 경우, 함수에서 콜드 스타트가 나타날 때마다 Lambda는 초기화 코드를 다시 실행해야 할 수 있습니다. 이러한 함수의 경우 함수에 필요할 때까지 특정 기능의 초기화를 지연할 수 있습니다. 예를 들어, Lambda 핸들러에 대한 다음 제어 흐름을 고려하세요.

```
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

이전 예제에서는 개발자가 함수 기본 핸들러 외부에서 `CLIENT_A`를 초기화하는 대신 `if` 문 내부에서 초기화했습니다. 이를 통해 Lambda는 `some_condition`이 충족되는 경우에만 이 코드를 실행합니다. 기본 핸들러 외부에서 `CLIENT_A`를 초기화하면 Lambda는 모든 콜드 스타트에서 해당 코드를 실행합니다. 이로 인해 전체 지연 시간이 늘어날 수 있습니다.

환경 변수를 사용하여 프로비저닝된 동시성 동작 확인 및 제어

함수가 프로비저닝된 동시성을 모두 사용할 수 있습니다. Lambda는 온디맨드 인스턴스를 사용하여 초과 트래픽을 처리합니다. Lambda가 특정 환경에 사용하는 초기화 유형을 결정하려면 `AWS_LAMBDA_INITIALIZATION_TYPE` 환경 변수의 값을 확인합니다. 이 변수에는 두 가지 가능한 값(`provisioned-concurrency` 또는 `on-demand`)이 있습니다. `AWS_LAMBDA_INITIALIZATION_TYPE`의 값은 변경할 수 없으며 환경의 전체 수명 주기 동안 일정하게 유지됩니다. 함수 코드에서 환경 변수의 값을 확인하려면 ??? 섹션을 참조하세요.

.NET 6 또는 .NET 7 런타임을 사용하는 경우 프로비저닝된 동시성을 사용하지 않더라도 함수의 지연 시간을 개선하도록 `AWS_LAMBDA_DOTNET_PREJIT` 환경 변수를 구성할 수 있습니다. .NET 런타임은 코드가 처음 직접 호출하는 각 라이브러리에서 지연 컴파일 및 초기화를 수행합니다. 결과적으로 Lambda 함수의 첫 번째 간접 호출은 후속 간접 호출보다 오래 걸릴 수 있습니다. 이를 완화하기 위해 `AWS_LAMBDA_DOTNET_PREJIT`에 대해 세 가지 값 중 하나를 선택할 수 있습니다.

- `ProvisionedConcurrency`: Lambda가 프로비저닝된 동시성을 사용하여 모든 환경에 대해 사전 JIT 컴파일을 수행합니다. 이것이 기본값입니다.
- `Always`: 함수가 프로비저닝된 동시성을 사용하지 않는 경우에도 Lambda가 모든 환경에 대해 사전 JIT 컴파일을 수행합니다.
- `Never`: Lambda가 모든 환경에 대해 사전 JIT 컴파일을 비활성화합니다.

프로비저닝된 동시성을 사용한 로깅 및 청구 동작 이해

프로비저닝된 동시성 환경의 경우 Lambda가 환경의 인스턴스를 재활용하므로 함수의 초기화 코드가 할당 중에 실행됩니다. 환경 인스턴스가 요청을 처리한 후 로그와 트레이스에서 초기화 시간을 확인할 수 있습니다. Lambda는 환경 인스턴스가 요청을 처리하지 않더라도 초기화 비용을 청구한다는 점에 주의합니다. 프로비저닝된 동시성은 계속 실행되며 초기화 및 간접 호출 비용과는 별도로 비용이 청구됩니다. 자세한 내용은 AWS Lambda 요금을 참조하세요.

또한 프로비저닝된 동시성을 사용하여 Lambda 함수를 구성하면 Lambda는 함수 간접 호출 요청에 앞서 해당 실행 환경을 사용할 수 있도록 사전 초기화합니다. 그러나 함수는 함수가 실제로 간접 호출된 경우에만 CloudWatch에 호출 로그를 게시합니다. 따라서 초기화가 미리 수행된 경우에도 첫 번째 함수 간접 호출의 `REPORT` 로그 줄에 초기화 지속 시간 필드가 나타납니다. 이는 함수가 콜드 스타트를 경험했다는 의미는 아닙니다.

Application Auto Scaling을 사용하여 프로비저닝된 동시성 관리 자동화

Application Auto Scaling을 사용하여 일정에 따라 또는 사용률을 기준으로 프로비저닝된 동시성을 관리할 수 있습니다. 함수에 예측 가능한 트래픽 패턴이 수신되는 경우 예약된 크기 조정을 사용합니다. 함수가 특정 사용률(%)을 유지하도록 하려면 대상 추적 스케일링 정책을 사용합니다.

예약된 조정

Application Auto Scaling을 사용하면 예측 가능한 로드 변경에 따라 자체 조정 일정을 설정할 수 있습니다. 자세한 내용 및 예제는 Application Auto Scaling 사용 설명서의 Application Auto Scaling의 예약된 조정과 AWS Compute Blog의 Scheduling AWS Lambda Provisioned Concurrency for recurring peak usage를 참조하세요.

대상 추적

대상 추적을 통해 Application Auto Scaling은 스케일링 정책 정의 방식에 따라 CloudWatch 경보 세트를 생성하고 관리합니다. 이러한 경보가 활성화되면 Application Auto Scaling은 프로비저닝된 동시성을 사용하여 할당된 환경의 양을 자동으로 조정합니다. 예측 가능한 트래픽 패턴이 없는 애플리케이션에서는 대상 추적을 사용합니다.

대상 추적을 사용하여 프로비저닝된 동시성을 스케일링하려면 `RegisterScalableTarget` 및 `PutScalingPolicy` Application Auto Scaling API 작업을 사용합니다. 예를 들어, AWS Command Line Interface(CLI)를 사용하는 경우 다음 단계를 따르세요.

1. 함수의 별칭을 조정 대상으로 등록합니다. 다음 예제에서는 `my-function` 함수의 BLUE 별칭을 등록합니다.

```
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. 조정 정책을 대상에 적용합니다. 다음 예제에서는 별칭에 대해 프로비저닝된 동시성 구성을 조정하여 사용률을 70% 가까이 유지하도록 애플리케이션 Auto Scaling을 구성하지만 10%에서 90% 사이의 값을 적용할 수 있습니다.

```
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" }}'
```

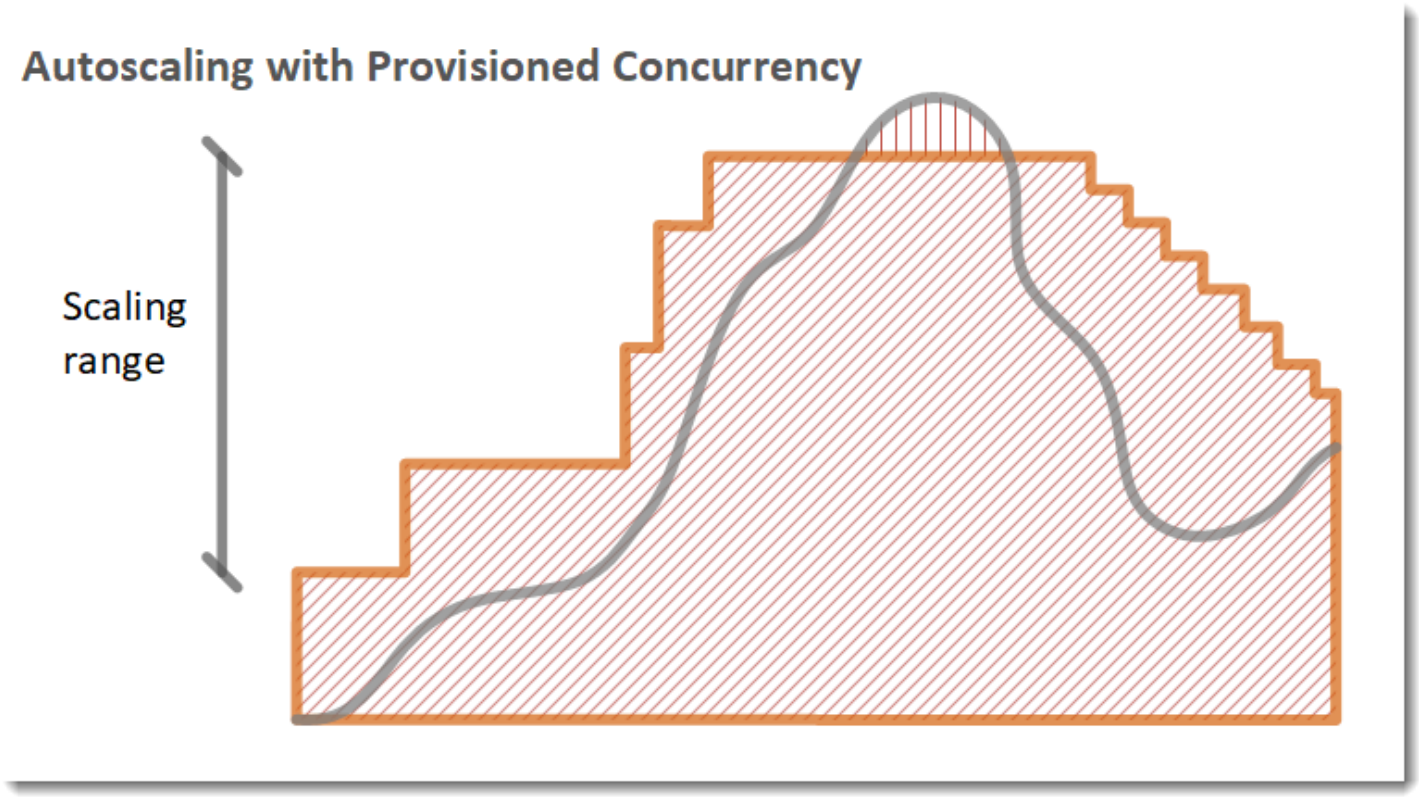
다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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 Scaling은 CloudWatch에서 두 개의 경보를 생성합니다. 프로비저닝된 동시성의 사용률이 지속적으로 70%를 초과하면 첫 번째 경보가 트리거됩니다. 이 경우 Application Auto Scaling은 프로비저닝된 동시성을 더 많이 할당하여 사용률을 줄입니다. 두 번째 경보는 사용률이 지속적으로 63%(70% 대상의 90%) 미만인 경우에 트리거됩니다. 이 경우 Application Auto Scaling이 별칭의 프로비저닝된 동시성을 줄입니다.

다음 예제에서는 프로비저닝된 동시성의 최소 양과 최대 양 사이에서 함수 크기가 사용률에 따라 조정됩니다.



범례

- 함수 인스턴스
- 미결 요청
- 프로비저닝된 동시성
- 표준 동시성

미결 요청 수가 증가하면 구성된 최대값에 도달할 때까지 Application Auto Scaling이 프로비저닝된 동시성을 큰 폭으로 늘립니다. 이후 계정 동시성 한도에 도달하지 않은 경우 함수는 예약되지 않은 표준 동시성으로 계속 크기를 조정할 수 있습니다. 사용률이 떨어지고 낮게 유지되는 경우 Application Auto Scaling은 프로비저닝된 동시성을 주기적인 작은 단계로 줄입니다.

두 Application Auto Scaling 경보에서는 모두 기본적으로 평균 통계를 사용합니다. 트래픽이 빠르게 버스트되는 함수는 이러한 경보를 트리거하지 않을 수 있습니다. 예를 들어, Lambda 함수가 빠르게 실행되고(예: 20ms~100ms) 트래픽에서 빠른 버스트가 발생한다고 가정합니다. 이 경우 버스트 중에 요청 수가 할당된 프로비저닝된 동시성을 초과합니다. 하지만 Application Auto Scaling에서는 추가 환경을 프로비저닝하기 위해 최소 3분 동안 버스트 로드가 지속되어야 합니다. 또한 두 CloudWatch 경보 모두에는 Auto Scaling 정책을 활성화하기 위해 목표 평균에 도달하는 3개의 데이터 포인트가 필요합니다. 함수에서 트래픽이 빠르게 급증하는 경우 평균 통계 대신 최대 통계를 사용하면 프로비저닝된 동시성의 규모를 조정하여 콜드 스타트를 최소화하는 데 더 효과적일 수 있습니다.

대상 추적 조정 정책에 대한 자세한 내용은 Application Auto Scaling의 대상 추적 조정 정책을 참조하세요.

Lambda 조정 동작

함수가 더 많은 요청을 수신하면 Lambda는이 계정 동시성 할당량에 도달할 때까지 이러한 요청을 처리할 수 있도록 실행 환경 수를 자동으로 스케일 업합니다. 그러나 갑작스러운 트래픽 폭증에 대응한 오버스케일링을 방지하기 위해 Lambda는 함수의 스케일 속도를 제한합니다. 이 동시성 조정 비율은 요청 증가에 따라 계정 내 함수를 확장할 수 있는 최대 비율입니다. (즉, Lambda가 새 실행 환경을 얼마나 빨리 생성할 수 있는지를 의미합니다.) 동시성 조정 비율은 함수에 사용할 수 있는 동시성 총량인 계정 수준 동시성 한도와는 다릅니다.

동시성 확장 속도

각 AWS 리전에서, 각 함수에 대한 동시성 스케일 속도는 10초당 1,000개의 실행 환경 인스턴스입니다. 즉, Lambda는 10초당 최대 1,000개의 추가 실행 환경 인스턴스를 각 함수에 할당할 수 있습니다.

일반적으로 이 제한 사항에 대해 걱정할 필요는 없습니다. Lambda의 스케일 속도는 대부분의 사용 사례에 충분합니다.

중요한 점은 동시성 조정 비율이 함수 수준의 한도라는 점입니다. 즉, 계정의 각 함수가 다른 함수와 독립적으로 스케일할 수 있다는 의미입니다.

> **Note**
>
> 실제로 Lambda는 10초당 1,000개의 유닛을 한 번 다시 채우는 대신 시간이 지남에 따라 지속적으로 다시 동시성 확장 속도를 다시 채우기 위해 최선을 다합니다.

Lambda는 동시성 스케일 요금의 미사용 부분을 누적시키지 않습니다. 즉, 어느 시점에서든 조정 속도는 항상 최대 1,000개의 동시 실행 단위라는 의미입니다. 예를 들어 10초 간격으로 사용 가능한 1,000개의 동시 실행 단위를 사용하지 않으면 다음 10초 간격으로 1,000개의 추가 단위가 누적되지 않습니다. 다음 10초 간격에도 동시성 스케일 비율은 여전히 1,000입니다.

함수가 계속해서 점점 더 많은 요청을 수신하는 한, Lambda는 계정의 동시 실행 한도까지 가능한 가장 빠른 속도로 스케일합니다. 예약된 동시성을 구성하여 개별 함수가 사용할 수 있는 동시성의 양을 제한할 수 있습니다. 함수가 확장하는 속도보다 더 빠르게 요청이 수신되거나 함수가 최대 동시성에 도달한 경우 그런 다음 추가 요청은 조절 오류(429 상태 코드)로 인해 실패합니다.

동시성 모니터링

Lambda는 CloudWatch Amazon 지표를 생성하여 함수의 동시성을 모니터링하는 데 도움이 됩니다. 이 토픽에서는 이러한 지표와 이를 해석하는 방법을 설명합니다.

Sections

일반 동시성 지표

다음 지표를 사용하여 Lambda 함수의 동시성을 모니터링합니다. 각 지표의 단위는 1분입니다.

- `ConcurrentExecutions` - 지정된 시점의 활성 동시 호출 수입니다. Lambda는 모든 함수, 버전 및 별칭에 대해 이 지표를 내보냅니다. Lambda 콘솔의 모든 함수에 대해 Lambda는 기본적으로 지표 아래의 모니터링 탭에 `ConcurrentExecutions`에 대한 그래프를 표시합니다. MAX를 사용하여 이 지표를 봅니다.
- `UnreservedConcurrentExecutions` - 예약되지 않은 동시성을 사용하는 활성 동시 호출 수입니다. Lambda는 리전의 모든 함수에 대해 이 지표를 내보냅니다. MAX를 사용하여 이 지표를 봅니다.
- `ClaimedAccountConcurrency` - 온디맨드 간접 호출에 사용할 수 없는 동시 실행 수입니다. `ClaimedAccountConcurrency`는 `UnreservedConcurrentExecutions`에 할당된 동시성 크기를 더한 값과 같습니다(즉, 예약된 총 동시성과 프로비저닝된 총 동시성 합계). `ClaimedAccountConcurrency`가 계정의 동시성 한도를 초과하는 경우 더 높은 계정 동시성 한도를 요청할 수 있습니다. MAX를 사용하여 이 지표를 봅니다. 자세한 설명은 `ClaimedAccountConcurrency` 지표 작업 섹션을 참조하세요.

프로비저닝된 동시성 지표

다음 지표를 사용하여 프로비저닝된 동시성을 사용하는 Lambda 함수를 모니터링합니다. 각 지표의 단위는 1분입니다.

- `ProvisionedConcurrentExecutions` – 프로비저닝된 동시성에 대한 호출을 능동적으로 처리하는 실행 환경 인스턴스의 수입니다. Lambda는 프로비저닝된 동시성이 구성된 각 함수 버전 및 별칭에 대해 이 지표를 내보냅니다. MAX를 사용하여 이 지표를 봅니다.

ProvisionedConcurrentExecutions는 할당하는 프로비저닝된 동시성의 총 수와 동일하지 않습니다. 예를 들어, 100단위의 프로비저닝된 동시성을 함수 버전에 할당한다고 가정해 보겠습니다. 지정된 시간 동안 실행 환경 100개 중 최대 50개가 호출을 동시에 처리하는 경우 MAX(ProvisionedConcurrentExecutions) 값은 50입니다.

- ProvisionedConcurrentInvocations – Lambda가 프로비저닝된 동시성을 사용하여 함수 코드를 호출되는 횟수입니다. Lambda는 프로비저닝된 동시성이 구성된 각 함수 버전 및 별칭에 대해 이 지표를 내보냅니다. SUM을 사용하여 이 지표를 봅니다.

ProvisionedConcurrentInvocations는 총 호출 수를 계산하고 ProvisionedConcurrentExecutions는 활성 환경 수를 계산한다는 점에서 ProvisionedConcurrentInvocations는 ProvisionedConcurrentExecutions와 다릅니다. 이러한 차이를 이해하려면 다음 시나리오를 고려하세요.

이 예제에서는 분당 1개의 호출을 수신하고 각 호출을 완료하는 데 2분이 걸린다고 가정해 보겠습니다. 각 주황색 가로 막대는 단일 요청을 나타냅니다. 각 요청이 프로비저닝된 동시성에서 실행되도록 이 함수에 10단위의 프로비저닝된 동시성을 할당한다고 가정해 보겠습니다.

0분에서 1분 사이에 Request 1이 들어옵니다. 1분에서 MAX(ProvisionedConcurrentExecutions)의 값은 1입니다. 지난 1분 동안 최대 1개의 실행 환경이 활성화되었기 때문입니다. SUM(ProvisionedConcurrentInvocations) 값도 1입니다. 지난 1분 동안 1개의 새 요청이 들어왔기 때문입니다.

1분에서 2분 사이에 Request 2가 들어오고 Request 1이 계속 실행됩니다. 2분에서 MAX(ProvisionedConcurrentExecutions)의 값은 2입니다. 지난 1분 동안 최대 2개의 실행 환경이 활성화되었기 때문입니다. 그러나 SUM(ProvisionedConcurrentInvocations) 값은 1입

니다. 지난 1분 동안 1개의 새 요청만 들어왔기 때문입니다. 이 지표 동작은 예제가 끝날 때까지 계속됩니다.

- `ProvisionedConcurrencySpilloverInvocations` – 모든 프로비저닝된 동시성을 사용 중인 경우 Lambda가 표준 동시성(예약되거나 예약되지 않은 동시성)을 사용하여 함수를 호출하는 횟수입니다. Lambda는 프로비저닝된 동시성이 구성된 각 함수 버전 및 별칭에 대해 이 지표를 내보냅니다. SUM을 사용하여 이 지표를 봅니다. `ProvisionedConcurrentInvocations` + `ProvisionedConcurrencySpilloverInvocations`의 값은 총 함수 호출 수(즉, `Invocations` 지표)와 같아야 합니다.

  `ProvisionedConcurrencyUtilization` – 사용 중인 프로비저닝된 동시성의 백분율(즉, `ProvisionedConcurrentExecutions` 값을 할당된 프로비저닝된 동시성의 총량으로 나눈 값). Lambda는 프로비저닝된 동시성이 구성된 각 함수 버전 및 별칭에 대해 이 지표를 내보냅니다. MAX를 사용하여 이 지표를 봅니다.

예를 들어, 100단위의 프로비저닝된 동시성을 함수 버전에 프로비저닝한다고 가정해 보겠습니다. 지정된 시간 동안 실행 환경 100개 중 최대 60개가 호출을 동시에 처리하는 경우 MAX(`ProvisionedConcurrentExecutions`) 값은 60이고 MAX(`ProvisionedConcurrentUtilization`) 값은 0.6입니다.

`ProvisionedConcurrencySpilloverInvocations` 값이 높으면 함수에 프로비저닝된 동시성을 추가로 할당해야 할 수 있습니다. 또는 사전 정의된 임계값을 기반으로 프로비저닝된 동시성의 자동 크기 조정을 처리하도록 Application Auto Scaling을 구성할 수 있습니다.

반대로 `ProvisionedConcurrencyUtilization`의 값이 계속 낮으면 함수에 대해 프로비저닝된 동시성이 과도하게 할당되었을 수 있습니다.

`ClaimedAccountConcurrency` 지표 작업

Lambda는 `ClaimedAccountConcurrency` 지표를 사용하여 온디맨드 간접 호출에 사용할 수 있는 계정의 동시성 수를 결정합니다. Lambda는 다음 수식을 `ClaimedAccountConcurrency`를 계산합니다.

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

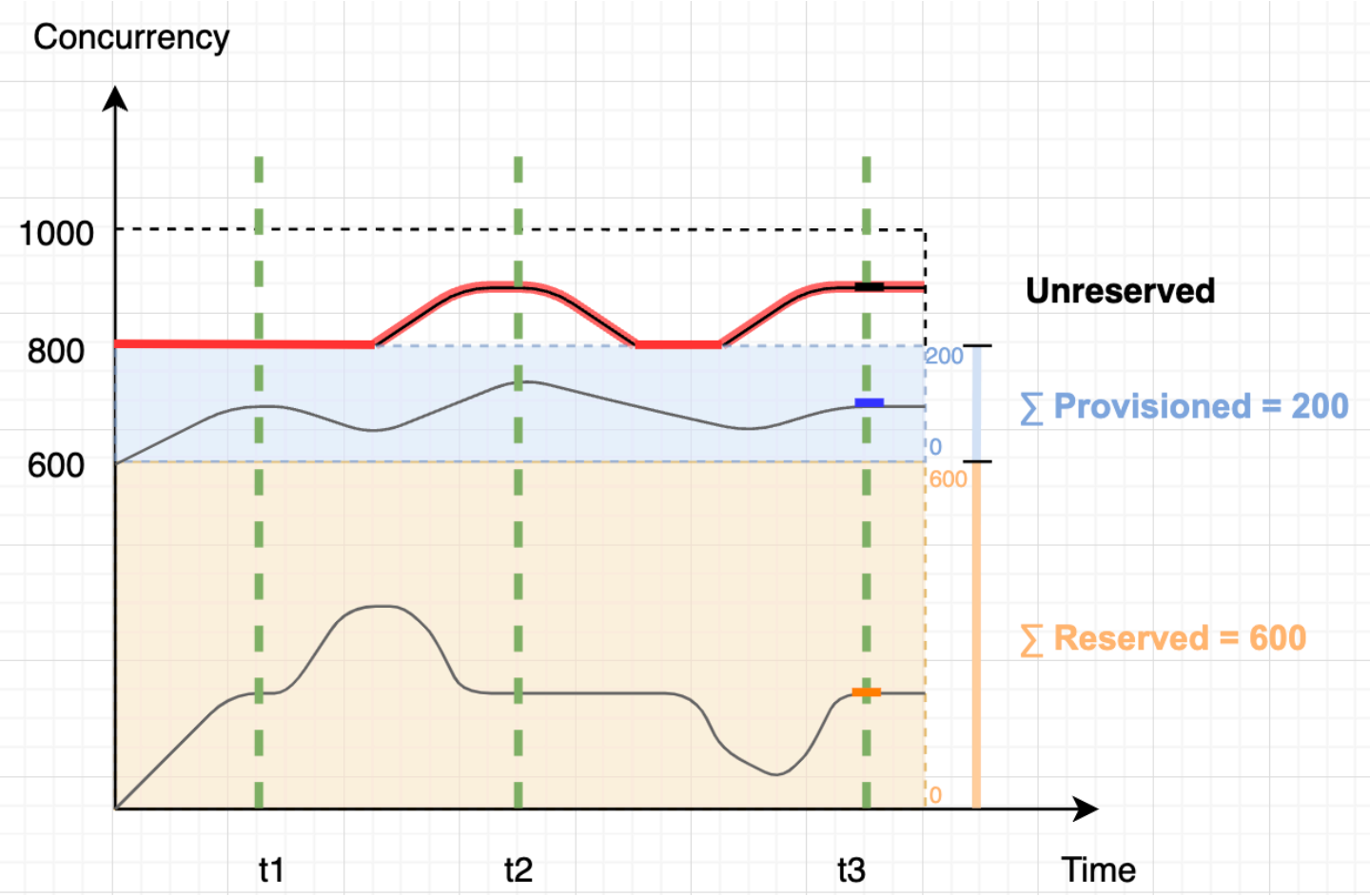
`UnreservedConcurrentExecutions`는 예약되지 않은 동시성을 사용하는 활성 동시 간접 호출 수입니다. 할당된 동시성은 다음 두 부분의 합계입니다(RC는 '예약된 동시성', PC는 '프로비저닝된 동시성'으로 대체).

- 리전의 모든 함수에 대한 RC 합계.
- RC를 사용하는 함수를 제외하고 PC를 사용하는 리전의 모든 함수에서 PC 합계.

> **Note**
>
> 함수에서 RC보다 더 많은 PC를 할당할 수 없습니다. 따라서 함수의 RC는 항상 해당 PC 이상입니다. PC 및 RC를 모두 사용하는 함수에 할당된 동시성의 기여도를 계산할 때 Lambda는 둘 중 최댓값인 RC만 고려합니다.

Lambda는 `ConcurrentExecutions` 대신 `ClaimedAccountConcurrency` 지표를 사용하여 온디맨드 간접 호출에 사용할 수 있는 동시성 수를 결정합니다. `ConcurrentExecutions` 지표는 활성 동시 간접 호출 수를 추적하는 데 유용하지만 실제 동시 실행 가능 여부를 항상 반영하는 것은 아닙니다. Lambda는 예약 동시성과 프로비저닝된 동시성도 고려하여 가용성을 결정하기 때문입니다.

`ClaimedAccountConcurrency`를 설명하기 위해 함수 전체에서 많은 예약 동시성과 프로비저닝된 동시성을 구성하지만 거의 사용되지 않는 시나리오를 살펴봅니다. 다음 예제에서는 계정 동시성 한도가 1,000이고 계정에 두 가지 주요 함수(`function-orange` 및 `function-blue`)가 있다고 가정합니다. `function-orange`의 예약된 동시성 단위로 600을 할당합니다. `function-blue`의 프로비저닝된 동시성 단위로 200을 할당합니다. 시간이 지남에 따라 추가 함수를 배포하고 다음 트래픽 패턴을 관찰한다고 가정합니다.



이전 다이어그램에서 검은색 선은 시간 경과에 따른 실제 동시성 수를 나타내고 빨간색 선은 시간 경과에 따른 `ClaimedAccountConcurrency` 값을 나타냅니다. 함수 전체에서 실제 동시성 사용률은 낮지만 이 시나리오에서는 `ClaimedAccountConcurrency`의 최솟값이 800입니다. `function-orange` 및 `function-blue`에 총 800의 동시성 단위를 할당했기 때문입니다. Lambda의 관점에서 이 동시성을 사용하도록 '청구'했으므로 다른 함수에 사용할 수 있는 남은 동시성 단위는 실제로 200입니다.

이 시나리오에서 할당된 동시성은 `ClaimedAccountConcurrency` 수식에서 800입니다. 그러면 다이어그램의 다양한 지점에서 `ClaimedAccountConcurrency`의 값을 파생시킬 수 있습니다.

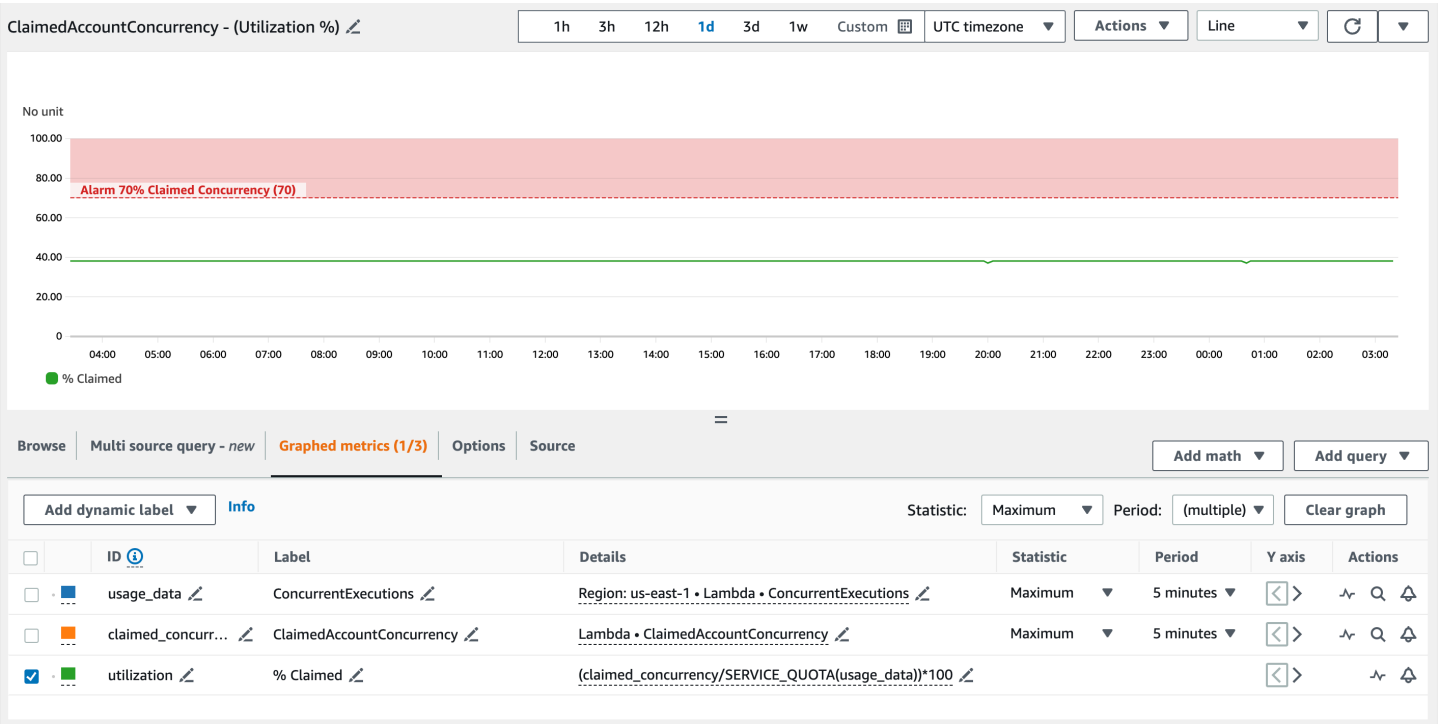
- `t1`에서 `ClaimedAccountConcurrency`는 800(800 + `UnreservedConcurrentExecutions` 0)입니다.
- `t2`에서 `ClaimedAccountConcurrency`는 900(800 + `UnreservedConcurrentExecutions` 100)입니다.
- `t3`에서 `ClaimedAccountConcurrency`는 다시 900(800 + `UnreservedConcurrentExecutions` 100)입니다.

측정치 설정: **ClaimedAccountConcurrency** CloudWatch

Lambda는 메트릭을 ClaimedAccountConcurrency 내보냅니다. CloudWatch 다음 수식과 같이 계정의 동시성 사용률 퍼센트를 얻기 위해 SERVICE_QUOTA(ConcurrentExecutions)의 값과 함께 이 지표를 사용합니다.

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

다음 스크린샷은 이 공식을 그래프로 표시하는 방법을 보여줍니다. CloudWatch 녹색 claim_utilization 선은 이 계정의 동시성 사용률을 나타내며, 약 40%입니다.



이전 스크린샷에는 동시 사용률이 70% 를 초과할 때 ALARM 상태가 되는 CloudWatch 경보도 포함되어 있습니다. 유사한 경보와 함께 ClaimedAccountConcurrency 지표를 사용하여 계정 동시성 한도 상향 조정을 요청해야 하는 시기를 사전에 결정할 수 있습니다.

AWS Lambda에 대한 코드 서명 구성

AWS Lambda에 대한 코드 서명은 신뢰할 수 있는 코드만 Lambda 함수에서 실행되도록 하는 데 도움이 됩니다. 함수에 코드 서명을 활성화하면 Lambda는 모든 코드 배포를 검사하고 코드 패키지가 신뢰할 수 있는 소스에 의해 서명되었는지 확인합니다.

> **Note**
>
> 컨테이너 이미지로 정의된 함수는 코드 서명을 지원하지 않습니다.

코드 무결성을 확인하려면 AWS Signer를 사용하여 함수 및 계층에 대해 디지털 방식으로 서명된 코드 패키지를 생성합니다. 사용자가 코드 패키지를 배포하려고 경우 Lambda는 배포를 수락하기 전에 코드 패키지에 대한 유효성 검사를 수행합니다. 코드 서명 유효성 검사는 배포 시 실행되므로 함수 실행 시의 성능에는 영향을 주지 않습니다.

AWS Signer를 사용하여 서명 프로필을 생성할 수도 있습니다. 서명 프로필을 사용하여 서명된 코드 패키지를 생성합니다. AWS Identity and Access Management(IAM)를 사용하여 코드 패키지에 서명할 수 있고 서명 프로필을 생성할 수 있는 사용자를 제어합니다. 자세한 내용은 AWS Signer 개발자 안내서의 인증 및 액세스 제어를 참조하세요.

함수에 대한 코드 서명을 활성화하려면 코드 서명 구성을 생성하고 함수에 연결합니다. 코드 서명 구성은 허용된 서명 프로필의 목록과 유효성 검사가 실패한 경우 수행할 정책 작업을 정의합니다.

Lambda 계층은 함수 코드 패키지와 동일한 서명된 코드 패키지 형식을 따릅니다. 코드 서명이 활성화된 함수에 계층을 추가하면 Lambda는 해당 계층이 허용된 서명 프로필에 의해 서명되었는지 확인합니다. 함수에 대해 코드 서명을 활성화하면 함수에 추가된 모든 계층도 허용된 서명 프로필 중 하나에 의해 서명되어야 합니다.

IAM을 사용하여 코드 서명 구성을 생성할 수 있는 사용자를 제어합니다. 일반적으로 특정 관리 사용자만 이 기능을 사용하도록 허용합니다. 또한 개발자가 코드 서명이 활성화된 함수만 생성하도록 강제하는 IAM 정책을 설정할 수 있습니다.

변경 사항을 AWS CloudTrail에 기록하도록 코드 서명을 구성할 수 있습니다. 함수에 대한 성공적인 배포와 차단된 배포는 서명 및 유효성 검사 관련 정보와 함께 CloudTrail에 기록됩니다.

Lambda 콘솔, AWS Command Line Interface(AWS CLI), AWS CloudFormation, 및 AWS Serverless Application Model(AWS SAM)를 사용하여 함수에 대한 코드 서명을 구성할 수 있습니다.

AWS Signer 사용 또는 AWS Lambda에 대한 코드 서명에 따르는 추가 요금은 없습니다.

단원

서명 검증

Lambda는 서명된 코드 패키지를 함수에 배포할 때 다음과 같은 검증을 수행합니다.

1. 무결성 - 코드 패키지가 서명된 이후 수정되지 않았는지 검증합니다. Lambda는 패키지의 해시를 서명의 해시와 비교합니다.
2. 만료 - 코드 패키지의 서명이 만료되지 않았는지 검증합니다.
3. 불일치 - 코드 패키지가 Lambda 함수에 대해 허용되는 서명 프로필 중 하나를 사용하여 서명되었는지 검증합니다. 서명이 없는 경우에도 불일치가 발생합니다.
4. 해지 - 코드 패키지의 서명이 해지되지 않았는지 검증합니다.

코드 서명 구성에 정의된 서명 검증 정책은 검증 확인 중 하나라도 실패하는 경우 다음 작업 중 Lambda가 수행할 작업을 결정합니다.

- 경고 - Lambda는 코드 패키지의 배포를 허용하지만 경고를 발생시킵니다. Lambda는 새로운 Amazon CloudWatch 지표를 발생시키고 경고를 CloudTrail 로그에 저장합니다.
- 적용 - Lambda는 경고를 발생시키고(경고 동작과 동일) 코드 패키지의 배포를 차단합니다.

만료, 불일치 및 해지 검증에 대한 정책을 구성할 수 있습니다. 무결성 검사에 대한 정책은 구성할 수 없습니다. 무결성 검사가 실패하면 Lambda는 배포를 차단합니다.

구성 사전 조건

Lambda 함수에 대한 코드 서명을 구성하려면 먼저 AWS Signer를 사용하여 다음을 수행해야 합니다.

- 하나 이상의 서명 프로필을 생성합니다.
- 서명 프로필을 사용하여 함수에 대한 서명된 코드 패키지를 생성합니다.

자세한 내용은 AWS Signer 개발자 안내서의 서명 프로필 생성(콘솔)을 참조하세요.

코드 서명 구성 생성

코드 서명 구성은 허용된 서명 프로필의 목록과 서명 검증 정책을 정의합니다.

코드 서명 구성을 생성하려면(콘솔)

1. Lambda 콘솔의 코드 서명 구성 페이지를 엽니다.
2. 구성 생성을 선택합니다.
3. 설명(Description)에 구성을 설명하는 이름을 입력합니다.
4. 서명 프로필(Signing profiles)에서 구성에 최대 20개의 서명 프로필을 추가합니다.
   a. 서명 프로필 버전 ARN(Signing profile version ARN)에서 프로필 버전의 Amazon 리소스 이름(ARN)을 선택하거나 ARN을 입력합니다.
   b. 다른 서명 프로필을 추가하려면 서명 프로필 추가(Add signing profiles)를 선택합니다.
5. 서명 검증 정책(Signature validation policy)에서 경고(Warn) 또는 적용(Enforce)을 선택합니다.
6. 구성 생성을 선택합니다.

코드 서명 구성 업데이트

코드 서명 구성을 업데이트할 경우 변경 사항은 코드 서명 구성이 연결된 함수의 향후 배포에 영향을 미칩니다.

코드 서명 구성을 업데이트하려면(콘솔)

1. Lambda 콘솔의 코드 서명 구성 페이지를 엽니다.
2. 업데이트할 코드 서명 구성을 선택한 다음 편집(Edit)을 선택합니다.
3. 설명(Description)에 구성을 설명하는 이름을 입력합니다.
4. 서명 프로필(Signing profiles)에서 구성에 최대 20개의 서명 프로필을 추가합니다.
   a. 서명 프로필 버전 ARN(Signing profile version ARN)에서 프로필 버전의 Amazon 리소스 이름(ARN)을 선택하거나 ARN을 입력합니다.

b. 다른 서명 프로필을 추가하려면 서명 프로필 추가(Add signing profiles)를 선택합니다.

5. 서명 검증 정책(Signature validation policy)에서 경고(Warn) 또는 적용(Enforce)을 선택합니다.
6. Save changes(변경 사항 저장)를 선택합니다.

코드 서명 구성 삭제.

코드 서명 구성을 사용하는 함수가 없는 경우에만 코드 서명 구성을 삭제할 수 있습니다.

코드 서명 구성을 삭제하려면(콘솔)

1. Lambda 콘솔의 코드 서명 구성 페이지를 엽니다.
2. 업데이트할 코드 서명 구성을 선택한 다음 삭제(Delete)를 선택합니다.
3. 확인하려면 삭제(Delete)를 다시 선택합니다.

함수에 대한 코드 서명 활성화

함수에 대한 코드 서명을 활성화하려면 코드 서명 구성을 함수와 연결합니다.

코드 서명 구성을 함수와 연결하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 코드 서명을 활성화하려는 함수를 선택합니다.
3. 구성 탭을 엽니다.
4. 아래로 스크롤하여 코드 서명을 선택합니다.
5. 편집을 선택합니다.
6. 코드 서명 편집(Edit code signing)에서 이 함수에 대한 코드 서명 구성을 선택합니다.
7. 저장을 선택합니다.

IAM 정책 구성

사용자에게 코드 서명 API 작업에 액세스할 수 있는 권한을 부여하려면 하나 이상의 정책 설명을 사용자 정책에 연결합니다. 사용자 정책에 대한 자세한 내용은 단원을 참조하세요Lambda에서 ID 기반 IAM 정책 작업

다음 예제 정책 설명은 코드 서명 구성을 생성, 업데이트 및 검색할 수 있는 권한을 부여합니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

관리자는 CodeSigningConfigArn 조건 키를 사용하여 개발자가 함수를 생성하거나 업데이트하는 데 사용해야 하는 코드 서명 구성을 지정할 수 있습니다.

다음 예제 정책 설명에서는 함수를 생성할 수 있는 권한을 부여합니다. 정책 설명에는 허용되는 코드 서명 구성을 지정할 수 있는 lambda:CodeSigningConfigArn 조건이 포함되어 있습니다. Lambda는 CodeSigningConfigArn 파라미터가 누락되었거나 조건의 값과 일치하지 않는 경우 CreateFunction API 요청을 차단합니다.

```
{
  "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 API를 사용하여 코드 서명 구성

AWS CLI 또는 AWS SDK를 사용하여 코드 서명 구성을 관리하려면 다음 API 작업을 사용합니다.

- ListCodeSigningConfigs
- CreateCodeSigningConfig
- GetCodeSigningConfig
- UpdateCodeSigningConfig
- DeleteCodeSigningConfig

함수에 대한 코드 서명 구성을 관리하려면 다음 API 작업을 사용합니다.

- CreateFunction
- GetFunctionCodeSigningConfig
- PutFunctionCodeSigningConfig
- DeleteFunctionCodeSigningConfig
- ListFunctionsByCodeSigningConfig

Lambda 함수에서 태그 사용

AWS Lambda 함수에 태그를 지정하여 ABAC(속성 기반 액세스 제어)를 활성화하고 소유자, 프로젝트 또는 부서별로 구성할 수 있습니다. 태그는 ABAC에서 리소스를 필터링하고 결제 보고서에 세부 정보를 추가하는 데 사용할 수 있도록 AWS 서비스 전체에서 지원되는 자유형 키-값 페어입니다.

태그는 버전이나 별칭이 아닌 함수 수준에서 적용됩니다. 태그는 버전을 게시할 때 스냅샷이 생성되는 버전별 구성에 포함되지 않습니다.

Sections

태그 작업에 필요한 권한

함수를 사용하는 사람의 AWS Identity and Access Management(IAM) 자격 증명(사용자, 그룹 또는 역할)에 적절한 권한을 부여합니다.

- lambda: ListTags — `ListTags` 함수에 태그가 있으면 함수를 `GetFunction` 호출하거나 호출해야 하는 모든 사람에게 이 권한을 부여하십시오.
- lambda: TagResource — 또는 를 호출해야 하는 모든 사람에게 이 권한을 부여합니다. `CreateFunction` `TagResource`

자세한 설명은 Lambda에서 ID 기반 IAM 정책 작업 섹션을 참조하세요.

Lambda 콘솔에서 태그 사용

Lambda 콘솔을 사용하여 태그가 있는 함수를 생성하고 기존 함수에 태그를 추가하고 추가한 태그를 기준으로 함수를 필터링할 수 있습니다.

함수를 생성할 때 태그를 추가하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 생성을 선택합니다.
3. 새로 작성(Author from scratch) 또는 컨테이너 이미지(Container image)를 선택합니다.

4. 기본 정보에서 다음과 같이 합니다.

   a. 함수 이름(Function name)에 함수 이름을 입력합니다. 함수 이름은 64자로 제한됩니다.

   b. 런타임에서 함수에 사용할 언어 버전을 선택합니다.

   c. (선택 사항) 아키텍처(Architecture)에서 함수에 사용할 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수의 배포 패키지를 빌드할 때 해당 패키지가 이 명령 세트 아키텍처와 호환되는지 확인합니다.

5. 고급 설정(Advanced settings)을 확장한 다음 태그 활성화(Enable tags)를 선택합니다.

6. 이렇게 하려면 태그 추가(Add new tag)를 선택한 다음 키(Key) 및 선택 사항인 값(Value)을 입력합니다. 태그를 더 추가하려면 이 단계를 반복합니다.

7. 함수 생성을 선택합니다.

기존 함수에 태그를 추가하려면

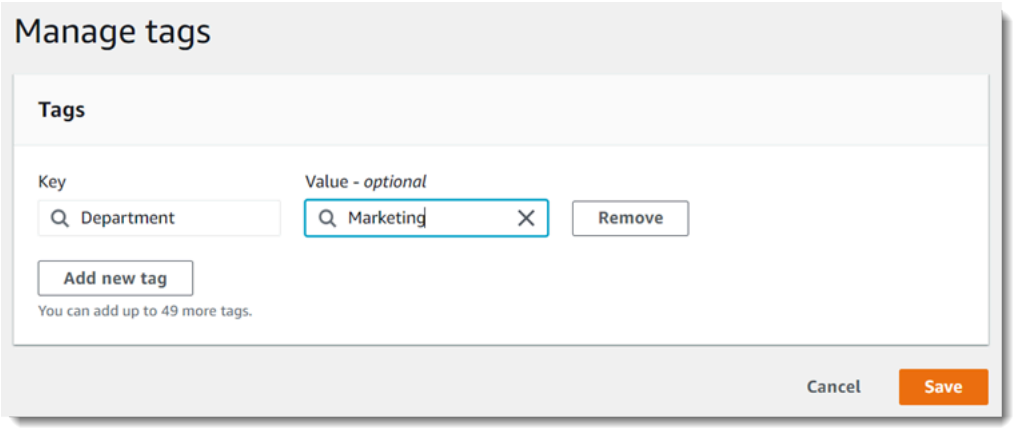
1. Lambda 콘솔의 함수 페이지를 엽니다.

2. 함수의 이름을 선택합니다.

3. 구성(Configuration)을 선택한 다음 태그(Tags)를 선택합니다.

4. 태그에서 태그 관리(Manage tags)를 선택합니다.

5. 이렇게 하려면 태그 추가(Add new tag)를 선택한 다음 키(Key) 및 선택 사항인 값(Value)을 입력합니다. 태그를 더 추가하려면 이 단계를 반복합니다.

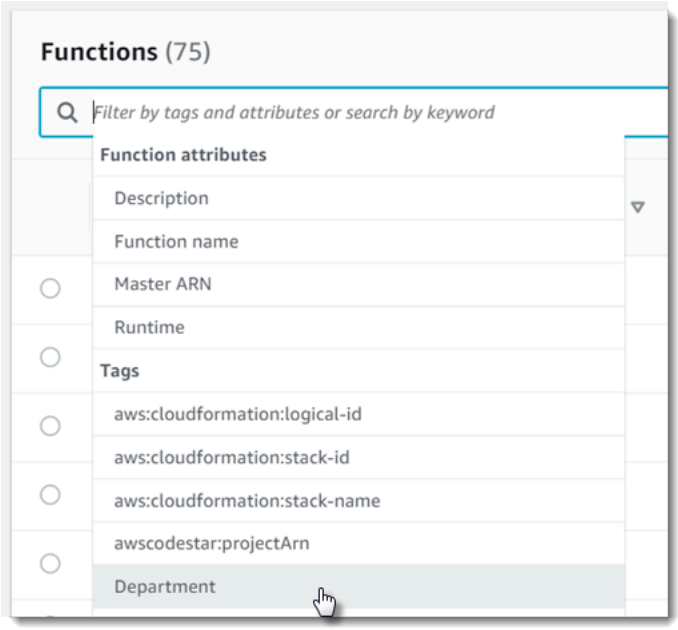


6. 저장을 선택합니다.

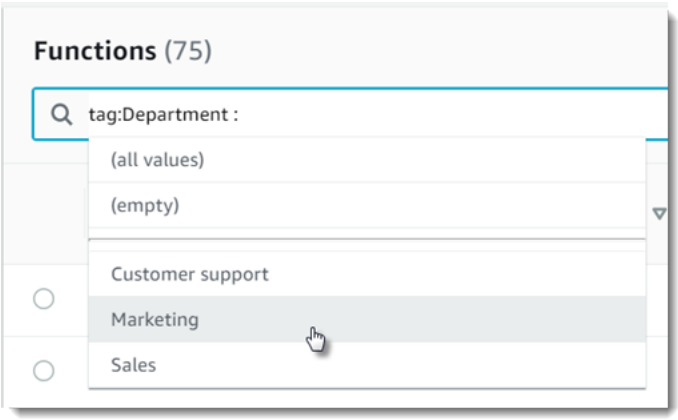
태그를 사용하여 함수를 필터링하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.

2. 검색 창을 선택하여 함수 속성 및 태그 키 목록을 표시합니다.



3. 태그 키를 선택하여 현재 AWS 리전에서 사용 중인 값 목록을 표시합니다.
4. 값을 선택하여 해당 값을 가진 함수를 표시하거나 (모든 값)을 선택하여 해당 키와 함께 태그가 있는 모든 함수를 표시합니다.



검색 창은 태그 키 검색도 지원합니다. `tag:`을(를) 입력하여 태그 키 목록만 표시하거나 키 이름을 입력하여 목록에서 찾습니다.

AWS CLI에서 태그 사용

태그 추가 및 제거

태그를 사용하여 새 Lambda 함수를 생성하려면 create-function 명령을 —tags 옵션과 함께 사용합니다.

```
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
```

기존 함수에 태그를 추가하려면 tag-resource 명령을 사용하세요.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:function:my-function \
--tags Department=Marketing,CostCenter=1234ABCD
```

태그를 제거하려면 untag-resource 명령을 사용합니다.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:function:my-
function \
--tag-keys Department
```

함수의 태그 보기

특정 Lambda 함수에 적용된 태그를 보려면 다음 AWS CLI 명령 중 하나를 사용할 수 있습니다.

- ListTags— 이 함수와 관련된 태그 목록을 보려면 Lambda 함수 ARN (Amazon 리소스 이름) 을 포함하십시오.

  ```
  aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:function:my-
  function
  ```

- GetFunction— 이 함수와 관련된 태그 목록을 보려면 Lambda 함수 이름을 포함하십시오.

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

태그를 기준으로 함수 필터링

AWS Resource Groups Tagging APIGetResourcesAPI 작업을 사용하여 태그를 기준으로 리소스를 필터링할 수 있습니다. GetResources 작업은 최대 10개의 필터를 수신하며 각 필터는 태그 키와 최대 10개의 태그 값을 포함합니다. GetResources에 ResourceType을 지정하면 특정 리소스 유형별로 필터링할 수 있습니다.

AWS Resource Groups에 대한 자세한 내용은 AWS Resource Groups 및 태그 사용 설명서에서 Resource Groups란 무엇입니까?를 참조하세요.

태그 요구 사항

태그에 적용되는 요구 사항은 다음과 같습니다.

- 리소스당 최대 태그 수: 50개
- 최대 키 길이: UTF-8의 유니코드 문자 128자
- 최대 값 길이: 유니코드 문자 256자(UTF-8)
- 태그 키와 값은 대/소문자를 구분합니다.
- 태그 이름이나 값에서 `aws:` 접두사는 사용하지 마세요. 이 단어는 AWS용으로 예약되어 있습니다. 이 접두사가 지정된 태그 이름이나 값은 편집하거나 삭제할 수 없습니다. 이 접두사가 지정된 태그는 리소스당 태그 수 제한에 포함되지 않습니다.
- 태그 지정 스키마를 여러 서비스와 리소스에서 사용하려는 경우, 서비스마다 허용되는 문자에 대한 제한이 다를 수 있음에 유의하십시오. 일반적으로 허용되는 문자는 UTF-8로 표현할 수 있는 문자, 공백 및 숫자와 특수 문자+ - = . _ : / @

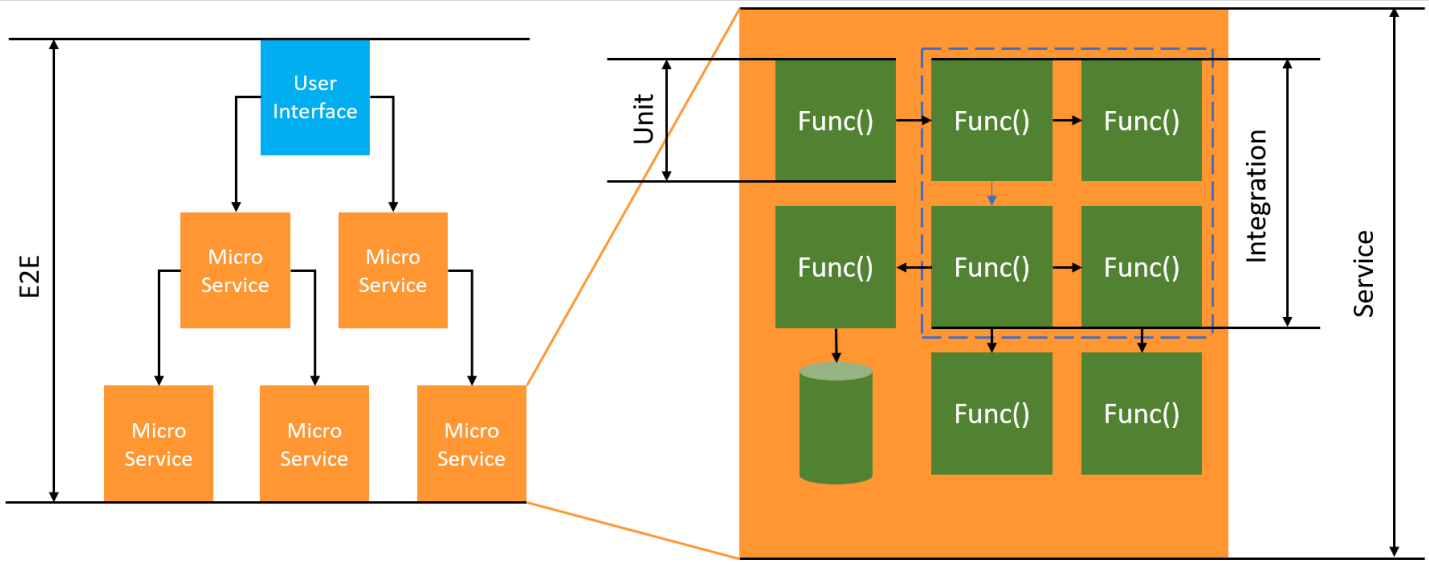
서버리스 함수 및 애플리케이션을 테스트하는 방법

서버리스 함수 테스트는 기존 테스트 유형과 기법을 사용하지만, 서버리스 애플리케이션을 전체적으로 테스트하는 것도 고려해야 합니다. 클라우드 기반 테스트는 함수와 서버리스 애플리케이션 모두의 품질을 가장 정확하게 측정합니다.

서버리스 애플리케이션 아키텍처에는 API 호출을 통해 중요한 애플리케이션 기능을 제공하는 관리형 서비스가 포함됩니다. 따라서 개발 주기에는 함수와 서비스가 상호 작용할 때 기능을 확인하는 자동화된 테스트가 포함되어야 합니다.

클라우드 기반 테스트를 생성하지 않으면 로컬 환경과 배포된 환경 간의 차이로 인해 문제가 발생할 수 있습니다. 지속적 통합 프로세스는 QA, 스테이징 또는 프로덕션과 같은 다음 배포 환경으로 코드를 승격하기 전에 클라우드에서 프로비저닝되는 리소스 제품군을 대상으로 테스트를 실행해야 합니다.

이 짧은 안내서를 계속 읽고 서버리스 애플리케이션의 테스트 전략에 대해 알아보거나 Serverless Test Samples 리포지토리를 방문하여 선택한 언어 및 런타임과 관련된 실제 예제를 자세히 살펴보세요.



서버리스 테스트의 경우에도 단위, 통합 및 엔드 투 엔드 테스트를 작성합니다.

- 단위 테스트 - 격리된 코드 블록에 대해 실행되는 테스트입니다. 예를 들어, 특정 항목과 대상에 대한 배송료를 계산하는 비즈니스 로직을 확인합니다.
- 통합 테스트 - 일반적으로 클라우드 환경에서 상호 작용하는 둘 이상의 구성 요소 또는 서비스를 포함하는 테스트입니다. 예를 들어, 함수가 대기열에서 이벤트를 처리하는지 확인합니다.
- 엔드 투 엔드 테스트 - 전체 애플리케이션의 동작을 확인하는 테스트입니다. 예를 들어, 인프라가 올바르게 설정되어 있고 고객의 주문 기록을 위해 예상대로 서비스 간에 이벤트가 흐르는지 확인합니다.

목표 비즈니스 성과

서버리스 솔루션을 테스트하려면 서비스 간의 이벤트 기반 상호 작용을 확인하는 테스트를 설정하는 데 약간 더 많은 시간이 필요할 수 있습니다. 이 안내서를 읽을 때 다음과 같은 실질적인 비즈니스 이유를 염두에 두세요.

- 애플리케이션 품질 향상
- 기능 빌드 및 버그 수정 시간 단축

애플리케이션 품질은 기능을 확인하기 위한 다양한 시나리오 테스트에 따라 달라집니다. 비즈니스 시나리오를 신중하게 고려하고 이러한 테스트를 자동화하여 클라우드 서비스에 대해 실행하면 애플리케이션의 품질이 향상됩니다.

소프트웨어 버그 및 구성 문제는 반복적인 개발 주기 중 발견될 때 비용 및 일정에 미치는 영향이 가장 적습니다. 개발 중 문제가 발견되지 않은 경우 프로덕션에서 발견하고 수정하려면 더 많은 사람이 더 많은 노력을 기울여야 합니다.

잘 계획된 서버리스 테스트 전략은 Lambda 함수와 애플리케이션이 클라우드 환경에서 예상대로 작동하는지 확인하여 소프트웨어 품질을 높이고 반복 시간을 개선합니다.

테스트 대상

관리형 서비스 동작, 클라우드 구성, 보안 정책 및 코드와의 통합을 테스트하여 소프트웨어 품질을 개선하는 테스트 전략을 채택하는 것이 좋습니다. 블랙 박스 테스팅이라고도 하는 동작 테스트는 모든 내부 요소를 파악하지 않고도 시스템이 예상대로 작동하는지 확인합니다.

- 단위 테스트를 실행하여 Lambda 함수 내부의 비즈니스 로직을 확인합니다.
- 통합 서비스가 실제로 호출되고 입력 파라미터가 올바른지 확인합니다.
- 이벤트가 워크플로에서 엔드 투 엔드로 예상되는 모든 서비스를 통과하는지 확인합니다.

기존 서버 기반 아키텍처에서는 팀이 애플리케이션 서버에서 실행되는 코드만 포함하도록 테스트 범위를 정의하는 경우가 많습니다. 다른 구성 요소, 서비스 또는 종속 항목은 외부 요소이며 테스트 범위를 벗어나는 것으로 간주되는 경우가 많습니다.

서버리스 애플리케이션은 데이터베이스에서 제품을 검색하거나 대기열에서 항목을 처리하거나 스토리지에서 이미지 크기를 조정하는 Lambda 함수와 같은 작은 작업 단위로 구성되는 경우가 많습니다. 각 구성 요소는 자체 환경에서 실행됩니다. 팀은 단일 애플리케이션 내에서 이러한 작은 단위 중 상당수를 담당하게 될 것입니다.

일부 애플리케이션 기능은 Amazon S3와 같은 관리형 서비스에 전적으로 위임하거나 내부에서 개발한 코드를 사용하지 않고 생성할 수 있습니다. 이러한 관리형 서비스를 테스트할 필요는 없지만 이러한 서비스와의 통합은 테스트해야 합니다.

서버리스 테스트 방법

로컬에 배포된 애플리케이션을 테스트하는 방법은 익히 알고 있을 것입니다. 데스크톱 운영 체제나 컨테이너 내에서 완전히 실행되는 코드를 대상으로 테스트를 작성합니다. 예를 들어, 요청으로 로컬 웹 서비스 구성 요소를 호출한 다음 응답에 대한 어설션을 만들 수 있습니다.

서버리스 솔루션은 함수 코드와 대기열, 데이터베이스, 이벤트 버스, 메시징 시스템 등의 클라우드 기반 관리형 서비스에서 구축됩니다. 이러한 구성 요소는 모두 이벤트라는 메시지가 한 리소스에서 다른 리소스로 흐르는 이벤트 기반 아키텍처를 통해 연결됩니다. 이러한 상호 작용은 웹 서비스가 결과를 즉시 반환하는 경우와 같이 동기식이거나 항목을 대기열에 배치하거나 워크플로 단계를 시작하는 것과 같이 나중에 완료되는 비동기식 작업일 수도 있습니다. 테스트 전략은 두 시나리오를 모두 포함하고 서비스 간 상호 작용을 테스트해야 합니다. 비동기 상호 작용의 경우 다운스트림 구성 요소에서 즉시 관찰할 수 없는 부작용을 감지해야 할 수 있습니다.

대기열, 데이터베이스 테이블, 이벤트 버스, 보안 정책 등을 포함한 전체 클라우드 환경을 복제하는 것은 실용적이지 않습니다. 로컬 환경과 클라우드에 배포된 환경의 차이로 인해 필연적으로 문제가 발생합니다. 환경의 차이로 인해 버그를 재현하고 수정하는 데 걸리는 시간이 늘어납니다.

서버리스 애플리케이션에서 아키텍처 구성 요소는 일반적으로 완전히 클라우드에 존재하므로 기능을 개발하고 버그를 수정하려면 클라우드의 코드 및 서비스에 대한 테스트가 필요합니다.

테스트 기법

실제로 테스트 전략에는 솔루션의 품질을 높이기 위한 다양한 기법이 포함될 수 있습니다. 콘솔에서 함수를 디버깅하기 위한 빠른 대화형 테스트, 격리된 비즈니스 로직을 확인하기 위한 자동화된 단위 테스트, 모의 객체로 외부 서비스에 대한 호출 확인, 서비스를 모방하는 에뮬레이터에 대한 간헐적 테스트 등을 사용합니다.

- 클라우드에서 테스트 - 실제 서비스, 보안 정책, 구성 및 인프라 관련 파라미터로 테스트할 인프라 및 코드를 배포합니다. 클라우드 기반 테스트는 코드의 품질을 가장 정확하게 측정합니다.

  콘솔에서 함수를 디버깅하면 클라우드에서 빠르게 테스트할 수 있습니다. 샘플 테스트 이벤트 라이브러리에서 선택하거나 함수를 격리하여 테스트하는 사용자 지정 이벤트를 생성할 수 있습니다. 콘솔을 통해 팀과 테스트 이벤트를 공유할 수도 있습니다.

개발 및 빌드 수명 주기에서 테스트를 자동화하려면 콘솔 외부에서 테스트해야 합니다. 자동화 전략 및 리소스는 이 안내서의 언어별 테스트 섹션을 참조하세요.

- 모의(가짜라고도 함) 객체로 테스트 - 모의 객체는 코드 내에서 외부 서비스를 시뮬레이션하고 대신하는 객체입니다. 모의 객체는 서비스 호출과 파라미터를 확인하기 위해 사전 정의된 동작을 제공합니다. 가짜는 성능을 단순화하거나 개선하기 위해 바로 가기를 사용하는 모의 구현입니다. 예를 들어, 가짜 데이터 액세스 객체는 메모리 내 데이터 스토어에서 데이터를 반환할 수 있습니다. 모의 객체는 복잡한 종속 항목을 모방하고 단순화할 수 있지만 중첩된 종속 항목을 대체하기 위한 더 많은 모의 객체로 이어질 수도 있습니다.
- 에뮬레이터로 테스트 - 로컬 환경에서 클라우드 서비스를 모방하도록 애플리케이션(경우에 따라 타사)을 설정할 수 있습니다. 속도는 강점이지만 설치 및 프로덕션 서비스와의 패리티는 약점입니다. 에뮬레이터는 적게 사용하세요.

클라우드에서 테스트

클라우드에서 테스트는 단위 테스트, 통합 테스트, 엔드 투 엔드 테스트 등의 모든 테스트 단계에서 중요합니다. 클라우드 기반 서비스와도 상호 작용하는 클라우드 기반 코드에 대해 테스트를 실행하면 코드 품질을 가장 정확하게 측정할 수 있습니다.

클라우드에서 Lambda 함수를 실행하는 편리한 방법은 AWS Management Console에서 테스트 이벤트를 사용하는 것입니다. 테스트 이벤트는 함수에 대한 JSON 입력입니다. 함수에 입력이 필요하지 않은 경우 이벤트는 빈 JSON 문서(`{}`)가 될 수 있습니다. 콘솔은 다양한 서비스 통합을 위한 샘플 이벤트를 제공합니다. 콘솔에서 이벤트를 생성한 후 팀과 공유하여 테스트를 더 쉽고 일관성 있게 만들 수도 있습니다.

콘솔에서 샘플 함수를 디버깅하는 방법을 알아보세요.

> **Note**
>
> 콘솔에서 함수 실행이 빠른 디버깅 방법이지만 애플리케이션 품질과 개발 속도를 높이는 데는 테스트 주기 자동화가 필수입니다.

테스트 자동화 샘플은 Serverless Test Samples 리포지토리에서 사용할 수 있습니다. 다음 명령줄은 자동화된 Python 통합 테스트 예제를 실행합니다.

```
python -m pytest -s tests/integration -v
```

테스트는 로컬에서 실행되지만 클라우드 기반 리소스와 상호 작용합니다. 이러한 리소스는 AWS Serverless Application Model 및 AWS SAM 명령줄 도구를 사용하여 배포되었습니다. 테스트 코드는 먼저 API 엔드포인트, 함수 ARN 및 보안 역할을 포함하는 배포된 스택 출력을 검색합니다. 다음으로 테스트에서 API 엔드포인트에 요청을 보내고, API 엔드포인트는 Amazon S3 버킷 목록으로 응답합니다. 이 테스트는 클라우드 기반 리소스에 대해 전적으로 실행되어 해당 리소스가 예상대로 배포, 보호 및 작동하는지 확인합니다.

```
========================= 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 ==========================
```

클라우드 네이티브 애플리케이션 개발의 경우 클라우드에서 테스트하면 다음과 같은 이점이 있습니다.

- 사용 가능한 모든 서비스를 테스트할 수 있습니다.
- 항상 최신 서비스 API와 반환 값을 사용하고 있습니다.
- 클라우드 테스트 환경은 프로덕션 환경과 매우 유사합니다.
- 테스트에서 보안 정책, 서비스 할당량, 구성 및 인프라 관련 파라미터를 다룰 수 있습니다.
- 모든 개발자는 클라우드에서 하나 이상의 테스트 환경을 빠르게 생성할 수 있습니다.
- 클라우드 테스트는 프로덕션에서 코드가 올바르게 실행될 것이라는 확신을 높여줍니다.

클라우드에서 테스트에는 몇 가지 단점이 있습니다. 클라우드에서 테스트의 가장 명백한 단점은 일반적으로 클라우드 환경에 배포가 로컬 데스크톱 환경에 배포보다 더 오래 걸린다는 것입니다.

다행히 AWS Serverless Application Model(AWS SAM) Accelerate, AWS Cloud Development Kit(AWS CDK) 감시 모드 및 SST(타사)와 같은 도구는 클라우드 배포 반복과 관련된 지연 시간을 줄입니다. 이러한 도구는 인프라와 코드를 모니터링하고 클라우드 환경에 자동으로 증분적 업데이트를 배포할 수 있습니다.

> **Note**
>
> AWS Serverless Application Model, AWS CloudFormation 및 AWS Cloud Development Kit (AWS CDK)에 대해 자세히 알아보려면 Serverless Developer Guide의 코드형 인프라 생성 방법을 참조하세요.

로컬 테스트와 달리 클라우드에서 테스트에는 서비스 비용이 발생할 수 있는 추가 리소스가 필요합니다. 격리된 테스트 환경을 생성하면 DevOps 팀의 부담이 가중될 수 있습니다. 특히 계정과 인프라에 대한 통제가 엄격한 조직에서는 더욱 그렇습니다. 그렇더라도 복잡한 인프라 시나리오 작업 시 복잡한 로컬 환경을 설정하고 유지 관리하는 데 드는 개발자 시간 비용은 코드형 인프라 자동화 도구로 생성한 일회용 테스트 환경을 사용하는 것과 비슷하거나 더 높을 수 있습니다.

이러한 고려 사항에도 불구하고 클라우드에서 테스트가 여전히 서버리스 솔루션의 품질을 보장하는 가장 좋은 방법입니다.

모의 객체로 테스트

모의 객체로 테스트는 코드에 대체 객체를 생성하여 클라우드 서비스의 동작을 시뮬레이션하는 기법입니다.

예를 들어, CreateObject 메서드가 호출될 때마다 특정 응답을 반환하는 Amazon S3 서비스의 모의 객체를 사용하는 테스트를 작성할 수 있습니다. 테스트가 실행되면 모의 객체는 Amazon S3나 다른 서비스 엔드포인트를 호출하지 않고 프로그래밍된 응답을 반환합니다.

모의 객체는 종종 개발 노력을 줄이기 위해 모의 프레임워크에 의해 생성됩니다. 일부 모의 프레임워크는 일반적이고 나머지는 AWS 서비스 및 리소스 모의를 위한 Python 라이브러리인 Moto와 같이 AWS SDK용으로 특별히 설계되었습니다.

모의 객체는 일반적으로 개발자가 테스트 코드의 일부로 생성하거나 구성한다는 점에서 에뮬레이터와 다른 반면 에뮬레이터는 에뮬레이션하는 시스템과 동일한 방식으로 기능을 노출하는 독립 실행형 애플리케이션입니다.

모의 객체를 사용하면 다음과 같은 이점이 있습니다.

- 모의 객체는 해당 서비스에 직접 액세스할 필요 없이 API 및 서비스형 소프트웨어(SaaS 공급자와 같이 애플리케이션의 제어 범위를 벗어나는 타사 서비스를 시뮬레이션할 수 있습니다.
- 모의 객체는 특히 서비스 중단과 같이 장애 조건을 시뮬레이트하기 어려운 경우 장애 조건을 테스트하는 데 유용합니다.
- 모의 객체는 일단 구성되면 빠른 로컬 테스트를 제공할 수 있습니다.
- 모의 객체는 거의 모든 종류의 객체에 대한 대체 동작을 제공할 수 있으므로 모의 전략은 에뮬레이터보다 더 다양한 서비스에 대한 적용 범위를 생성할 수 있습니다.
- 새로운 기능이나 동작을 사용할 수 있게 되면 모의 테스트를 통해 보다 신속하게 대응할 수 있습니다. 일반 모의 프레임워크를 사용하면 업데이트된 AWS SDK를 사용할 수 있게 되는 즉시 새로운 기능을 시뮬레이션할 수 있습니다.

모의 테스트에는 다음과 같은 단점이 있습니다.

- 모의 객체에는 일반적으로 상당한 양의 설정 및 구성 작업이 필요하며, 특히 응답을 제대로 모의하기 위해 다양한 서비스의 반환 값을 결정하려고 할 때 그렇습니다.
- 모의 객체는 개발자가 작성 및 구성하고 유지 관리해야 하므로 개발자의 책임이 가중됩니다.
- 서비스의 API 및 반환 값을 이해하려면 클라우드에 액세스해야 할 수 있습니다.
- 모의 객체는 유지 관리하기 어려울 수 있습니다. 모의된 클라우드 API 서명이 변경되거나 반환 값 스키마가 진화하면 모의 객체를 업데이트해야 합니다. 새 API를 호출하기 위해 애플리케이션 로직을 확장하는 경우 모의 객체도 업데이트가 필요합니다.

- 모의 객체를 사용하는 테스트는 데스크톱 환경에서는 통과하지만 클라우드에서는 실패할 수 있습니다. 결과가 현재 API와 일치하지 않을 수 있습니다. 서비스 구성과 할당량은 테스트할 수 없습니다.
- 모의 프레임워크는 AWS Identity and Access Management(IAM) 정책 또는 할당량 제한을 테스트하거나 감지하는 데 제한이 있습니다. 승인이 실패하거나 할당량이 초과될 때 모의 객체가 시뮬레이션을 더 잘 수행하지만 테스트를 통해 실제 프로덕션 환경에서 어떤 결과가 발생할지 결정할 수는 없습니다.

에뮬레이션으로 테스트

에뮬레이터는 일반적으로 프로덕션 AWS 서비스를 모방하는 로컬에서 실행되는 애플리케이션입니다.

에뮬레이터에는 클라우드의 해당 API와 유사한 API가 있으며 유사한 반환 값을 제공합니다. 또한 API 호출로 시작되는 상태 변경을 시뮬레이션할 수 있습니다. 예를 들어 AWS SAM을 사용하여 함수를 빠르게 호출할 수 있도록 AWS SAM 로컬로 함수를 실행하여 Lambda 서비스를 에뮬레이션할 수 있습니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서의 AWS SAM local을 참조하세요.

에뮬레이터로 테스트의 이점은 다음과 같습니다.

- 에뮬레이터는 빠른 로컬 개발 반복 및 테스트를 촉진할 수 있습니다.
- 에뮬레이터는 로컬 환경에서 코드를 개발하는 데 익숙한 개발자에게 친숙한 환경을 제공합니다. 예를 들어 n 계층 애플리케이션 개발에 익숙한 경우 프로덕션 환경에서 실행되는 것과 유사한 데이터베이스 엔진과 웹 서버가 로컬 컴퓨터에서 실행되어 신속하고 격리된 로컬 테스트 기능을 제공할 수 있습니다.
- 에뮬레이터는 개발자 클라우드 계정 등의 클라우드 인프라를 변경할 필요가 없으므로 기존 테스트 패턴으로 쉽게 구현할 수 있습니다.

에뮬레이터로 테스트에는 다음과 같은 단점이 있습니다.

- 에뮬레이터는 특히 CI/CD 파이프라인에서 사용할 때 설정 및 복제하기 어려울 수 있습니다. 이로 인해 자체 소프트웨어를 관리하는 IT 직원이나 개발자의 워크로드가 증가할 수 있습니다.
- 에뮬레이션된 기능과 API는 일반적으로 서비스 업데이트보다 지연됩니다. 이로 인해 테스트된 코드가 실제 API와 일치하지 않아 오류가 발생하고 새로운 기능의 채택에 방해가 될 수 있습니다.
- 에뮬레이터에는 지원, 업데이트, 버그 수정, 기능 패리티 개선이 필요합니다. 이는 타사일 수 있는 에뮬레이터 작성자의 책임입니다.
- 에뮬레이터에 의존하는 테스트는 로컬에서 성공적인 결과를 제공할 수 있지만 프로덕션 보안 정책, 서비스 간 구성 또는 Lambda 할당량 초과로 인해 클라우드에서는 실패합니다.
- 사용 가능한 에뮬레이터가 없는 AWS 서비스가 많습니다. 에뮬레이션에 의존하는 경우 애플리케이션 일부에 대해 만족스러운 테스트 옵션이 없을 수 있습니다.

모범 사례

다음 섹션에서는 성공적인 서버리스 애플리케이션 테스트를 위한 권장 사항을 제공합니다.

Serverless Test Samples 리포지토리에서 테스트 및 테스트 자동화의 실제 예를 찾아볼 수 있습니다.

클라우드에서 테스트 우선 순위 지정

클라우드에서 테스트는 가장 안정적이고 정확하며 완전한 테스트 범위를 제공합니다. 클라우드 컨텍스트에서 테스트를 수행하면 비즈니스 로직뿐만 아니라 보안 정책, 서비스 구성, 할당량, 최신 API 서명 및 반환 값까지 종합적으로 테스트할 수 있습니다.

테스트 가능성을 위한 코드 구성

핵심 비즈니스 로직에서 Lambda 관련 코드를 분리하여 테스트와 Lambda 함수를 간소화할 수 있습니다.

Lambda 함수 핸들러는 이벤트 데이터를 받아 비즈니스 로직 메서드에 중요한 세부 사항만 전달하는 슬림 어댑터여야 합니다. 이 전략을 사용하면 Lambda 관련 세부 사항에 대해 걱정하지 않고 비즈니스 로직을 중심으로 포괄적인 테스트를 수행할 수 있습니다. AWS Lambda 함수는 테스트 중인 구성 요소를 생성하고 초기화하기 위해 복잡한 환경이나 많은 양의 종속 항목을 설정할 필요가 없습니다.

일반적으로 수신 이벤트 및 컨텍스트 객체에서 데이터를 추출하고 검증한 다음 비즈니스 로직을 수행하는 메서드로 입력을 전송하는 핸들러를 작성해야 합니다.

개발 피드백 루프 가속화

개발 피드백 루프를 가속화하기 위한 도구와 기법이 있습니다. 예를 들어, AWS SAM Accelerate와 AWS CDK 감시 모드 모두 클라우드 환경을 업데이트하는 데 필요한 시간을 줄입니다.

GitHub Serverless Test Samples 리포지토리의 샘플에서 이러한 기법 중 일부를 살펴봅니다.

또한 소스 컨트롤을 체크인한 후뿐만 아니라 개발 과정에서 가능한 한 빨리 클라우드 리소스를 생성하고 테스트하는 것이 좋습니다. 이를 통해 솔루션을 개발할 때 더 빠르게 탐색하고 실험할 수 있습니다. 또한 개발 시스템에서 배포를 자동화하면 클라우드 구성 문제를 더 빨리 발견하고 업데이트 및 코드 검토 프로세스에 낭비되는 노력을 줄일 수 있습니다.

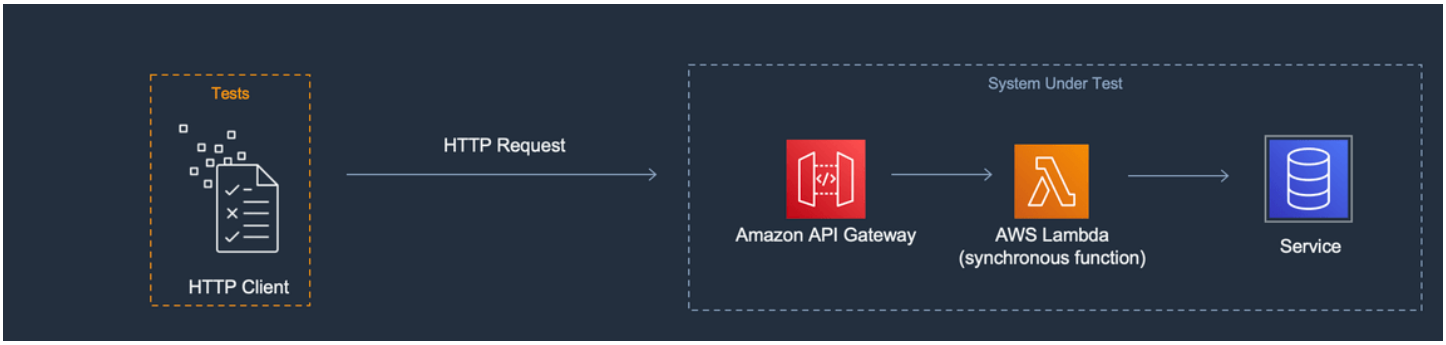
통합 테스트에 집중

Lambda로 애플리케이션을 빌드할 때 구성 요소를 함께 테스트하는 것이 가장 좋습니다.

여러 아키텍처 구성 요소에 대해 실행되는 테스트를 통합 테스트라고 합니다. 통합 테스트의 목표는 구성 요소에서 코드가 실행되는 방식뿐만 아니라 코드를 호스팅하는 환경이 어떻게 작동하는지 이해하는 것입니다. 엔드 투 엔드 테스트는 전체 애플리케이션에서 동작을 확인하는 특수 유형의 통합 테스트입니다.

통합 테스트를 빌드하려면 클라우드 환경에 애플리케이션을 배포합니다. 로컬 환경에서 또는 CI/CD 파이프라인을 통해 이를 수행할 수 있습니다. 그런 다음 테스트 대상 시스템(SUT)을 실행하고 예상되는 동작을 검증하는 테스트를 작성합니다.

예를 들어, 테스트 대상 시스템은 API Gateway, Lambda 및 DynamoDB를 사용하는 애플리케이션일 수 있습니다. 테스트는 API Gateway 엔드포인트에 대한 합성 HTTP 호출을 수행하고 응답에 예상 페이로드가 포함되었는지 확인할 수 있습니다. 이 테스트는 AWS Lambda 코드가 올바른지, 그리고 각 서비스가 서비스 간 IAM 권한을 포함하여 요청을 처리하도록 올바르게 구성되었는지 검증합니다. 또한 DynamoDB의 최대 레코드 크기와 같은 서비스 할당량이 제대로 설정되었는지 확인하기 위해 다양한 크기의 레코드를 작성하도록 테스트를 설계할 수 있습니다.



격리된 테스트 환경 생성

클라우드에서 테스트에는 일반적으로 테스트, 데이터 및 이벤트가 중복되지 않도록 격리된 개발자 환경이 필요합니다.

한 가지 접근 방식은 각 개발자에게 전용 AWS 계정을 제공하는 것입니다. 이를 통해 공유 코드베이스에서 작업하는 여러 개발자가 리소스를 배포하거나 API를 호출하려고 할 때 발생할 수 있는 리소스 이름 지정과의 충돌을 피할 수 있습니다.

자동화된 테스트 프로세스에서 각 스택에 대해 고유한 이름의 리소스를 생성해야 합니다. 예를 들어 AWS SAM CLI sam deploy 또는 sam sync 명령이 고유한 접두사가 있는 스택을 자동으로 지정하도록 스크립트 또는 TOML 구성 파일을 설정할 수 있습니다.

개발자가 AWS 계정을 공유하는 경우도 있습니다. 이는 운영 또는 프로비저닝 및 구성에 많은 비용이 드는 리소스가 스택에 있기 때문일 수 있습니다. 예를 들어, 데이터를 올바르게 설정하고 시드하기 쉽도록 데이터베이스를 공유할 수 있습니다.

개발자가 계정을 공유하는 경우 소유권을 식별하고 중복을 제거하기 위해 경계를 설정해야 합니다. 이를 수행하는 한 가지 방법은 스택 이름 앞에 개발자 사용자 ID를 추가하는 것입니다. 널리 사용되는 또 다른 접근 방식은 코드 분기를 기반으로 스택을 설정하는 것입니다. 분기 경계를 사용하면 환경이 분리되지만 개발자는 관계형 데이터베이스와 같은 리소스를 계속 공유할 수 있습니다. 이 접근 방식은 개발자가 한 번에 둘 이상의 분기에서 작업할 때 가장 좋은 방법입니다.

클라우드에서 테스트는 단위 테스트, 통합 테스트, 엔드 투 엔드 테스트 등의 모든 테스트 단계에서 중요합니다. 적절한 격리를 유지하는 것은 필수이지만, 대개 프로덕션 환경과 최대한 유사한 QA 환경을 원합니다. 이러한 이유로 팀은 QA 환경을 위한 변경 제어 프로세스를 추가합니다.

사전 프로덕션 및 프로덕션 환경의 경우 일반적으로 계정 수준에서 경계가 설정되어 잡음이 많은 이웃 문제로부터 워크로드를 격리하고 최소 권한 보안 제어를 구현하여 중요한 데이터를 보호합니다. 워크로드에는 할당량이 있습니다. 테스트에서 프로덕션(잡음이 많은 이웃)에 할당된 할당량을 사용하거나 고객 데이터에 액세스하지 못하도록 할 수 있습니다. 로드 테스트는 프로덕션 스택에서 분리해야 하는 또 다른 활동입니다.

어떤 경우든 불필요한 지출 방지를 위해 경고 및 제어 기능을 갖춘 환경을 구성해야 합니다. 예를 들어, 생성할 수 있는 리소스의 유형, 계층 또는 크기를 제한하고 예상 비용이 지정된 임계값을 초과할 때 이메일 알림을 설정할 수 있습니다.

분리된 비즈니스 로직에 모의 객체 사용

모의 프레임워크는 빠른 단위 테스트를 작성하는 데 유용한 도구입니다. 이는 테스트에서 수학 또는 재무 계산이나 시뮬레이션과 같은 복잡한 내부 비즈니스 로직을 다룰 때 특히 유용합니다. 테스트 사례 또는 입력 변형이 많고 이러한 입력이 다른 클라우드 서비스에 대한 호출 패턴이나 내용을 변경하지 않는 단위 테스트를 찾아보세요.

모의 객체를 사용한 단위 테스트에서 다루는 코드는 클라우드에서의 테스트에서도 다루어야 합니다. 개발자 랩톱 또는 빌드 시스템 환경을 클라우드의 프로덕션 환경과 다르게 구성할 수 있으므로 이 방법이 권장됩니다. 예를 들어, Lambda 함수는 특정 입력 파라미터로 실행할 때 할당된 것보다 더 많은 메모리나 시간을 사용할 수 있습니다. 또는 코드에 동일한 방식으로 또는 전혀 구성되지 않은 환경 변수가 포함될 수 있으며, 이러한 차이로 인해 코드가 다르게 동작하거나 실패할 수 있습니다.

연결 지점 수에 따라 필요한 모의 객체를 구현하는 노력 정도가 높아지기 때문에 통합 테스트에서는 모의 객체의 이점이 적습니다. 엔드 투 엔드 테스트에서는 대개 모의 프레임워크로는 쉽게 시뮬레이션할 수 없는 상태와 복잡한 로직을 다루기 때문에 모의 객체를 사용하면 안 됩니다.

마지막으로 서비스 호출의 적절한 구현을 검증하기 위해 모의 클라우드 서비스를 사용하지 마세요. 대신 클라우드에서 클라우드 서비스를 호출하여 동작, 구성 및 기능적 구현을 검증하세요.

에뮬레이터는 적게 사용하세요.

에뮬레이터는 인터넷 액세스가 제한적이거나 신뢰할 수 없거나 느린 개발 팀과 같은 일부 사용 사례에서 편리할 수 있습니다. 그러나 대부분의 경우 에뮬레이터를 적게 사용하는 것이 좋습니다.

에뮬레이터를 피하여 최신 서비스 기능과 최신 API로 빌드하고 혁신할 수 있습니다. 기능 패리티를 달성하기 위해 공급업체 릴리스를 기다리지 않아도 됩니다. 여러 개발 시스템과 빌드 시스템을 구매하고 구성하는 데 드는 초기 비용과 지속적인 비용을 줄일 수 있습니다. 또한 사용 가능한 에뮬레이터가 없는 클라우드 서비스가 많다는 문제를 피할 수 있습니다. 에뮬레이션에 의존하는 테스트 전략은 이러한 서비스를 사용하는 것을 불가능하게 하거나(잠재적으로 비용이 더 많이 드는 해결 방법으로 이어짐) 제대로 테스트되지 않은 코드와 구성을 생성합니다.

테스트에 에뮬레이션을 사용하는 경우에도 클라우드에서 테스트하여 구성을 확인하고 에뮬레이션된 환경에서만 시뮬레이션하거나 모의할 수 있는 클라우드 서비스와의 상호 작용을 테스트해야 합니다.

로컬 테스트 문제

에뮬레이터와 모의 호출을 사용하여 로컬 데스크톱에서 테스트하면 CI/CD 파이프라인의 환경 간에 코드가 진행되면서 테스트 불일치가 발생할 수 있습니다. 데스크톱에서 애플리케이션의 비즈니스 로직을 검증하기 위한 단위 테스트는 클라우드 서비스의 중요한 측면을 정확하게 테스트하지 못할 수 있습니다.

다음 예제에서는 모의 객체와 에뮬레이터를 사용하여 로컬에서 테스트할 때 주의해야 할 사례를 제공합니다.

예제: Lambda 함수에서 S3 버킷 생성

Lambda 함수의 로직이 S3 버킷 생성에 의존하는 경우 전체 테스트를 통해 Amazon S3가 호출되고 버킷이 성공적으로 생성되었는지 확인해야 합니다.

- 모의 테스트 설정에서는 성공 응답을 모의하고 잠재적으로 실패 응답을 처리하기 위한 테스트 사례를 추가할 수 있습니다.
- 에뮬레이션 테스트 시나리오에서 CreateBucket API가 호출될 수 있지만 로컬 호출을 수행하는 ID가 Lambda 서비스에서 시작되지 않는다는 점을 알고 있어야 합니다. 호출 ID는 클라우드에서와 같이 보안 역할을 수임하지 않으므로 자리 표시자 인증이 대신 사용되며 클라우드에서 실행될 때 달라질 수 있는 보다 허용적인 역할 또는 사용자 ID가 있을 수 있습니다.

모의 및 에뮬레이션 설정에서는 Lambda 함수가 Amazon S3를 호출할 경우 Lambda 함수가 수행할 작업을 테스트하지만, 구성된 대로 Lambda 함수가 Amazon S3 버킷을 성공적으로 생성할 수 있는지는

확인되지 않습니다. 함수에 할당된 역할에 함수가 `s3:CreateBucket` 작업을 수행할 수 있도록 허용하는 연결된 보안 정책이 있는지 확인해야 합니다. 그렇지 않으면 클라우드 환경에 배포할 때 함수가 실패할 수 있습니다.

예제: Lambda 함수에서 Amazon SQS 대기열의 메시지 처리

Amazon SQS 대기열이 Lambda 함수의 소스인 경우 전체 테스트를 통해 메시지가 대기열에 추가될 때 Lambda 함수가 성공적으로 호출되었는지 확인해야 합니다.

에뮬레이션 테스트와 모의 테스트는 일반적으로 Lambda 함수 코드를 직접 실행하고 JSON 이벤트 페이로드 또는 역직렬화된 객체를 함수 핸들러의 입력으로 전달하여 Amazon SQS 통합을 시뮬레이션하도록 설정됩니다.

Amazon SQS 통합을 시뮬레이션하는 로컬 테스트에서는 Amazon SQS가 지정된 페이로드와 함께 호출 시 Lambda 함수가 어떤 작업을 수행하는지 테스트하지만, Lambda 함수가 클라우드 환경에 배포될 때 Amazon SQS가 해당 함수를 성공적으로 호출하는지는 확인되지 않습니다.

다음은 Amazon SQS 및 Lambda에서 발생할 수 있는 구성 문제의 몇 가지 예입니다.

- Amazon SQS 가시성 제한 시간이 너무 짧아 한 번만 의도했는데 여러 번 호출됩니다.
- Lambda 함수의 실행 역할은 대기열에서 메시지 읽기를 허용하지 않습니다 (`sqs:ReceiveMessage`, `sqs:DeleteMessage` 또는 `sqs:GetQueueAttributes`를 통해).
- Lambda 함수에 전달된 샘플 이벤트가 Amazon SQS 메시지 크기 할당량을 초과합니다. 따라서 Amazon SQS에서는 해당 크기의 메시지를 전송할 수 없으므로 테스트가 유효하지 않습니다.

이러한 예에서 볼 수 있듯이 비즈니스 로직은 다루지만 클라우드 서비스 간의 구성은 다루지 않는 테스트는 신뢰할 수 없는 결과를 제공할 수 있습니다.

FAQ

다른 서비스를 호출하지 않고 계산을 수행하고 결과를 반환하는 Lambda 함수가 있습니다. 정말 클라우드에서 테스트해야 하나요?

예. Lambda 함수에는 테스트 결과를 변경할 수 있는 구성 파라미터가 있습니다. 모든 Lambda 함수 코드는 타임아웃 및 메모리 설정에 종속되므로 해당 설정이 제대로 설정되지 않으면 함수가 실패할 수 있습니다. 또한 Lambda 정책은 Amazon CloudWatch에 표준 출력 로깅을 가능하게 합니다. 코드가 CloudWatch를 직접 호출하지 않더라도 로깅을 활성화하려면 권한이 필요합니다. 이 필수 권한은 정확하게 모의하거나 에뮬레이션할 수 없습니다.

클라우드에서 테스트가 단위 테스트에 어떻게 도움이 될 수 있나요? 클라우드에 있고 다른 리소스에 연결되면 통합 테스트가 아닌가요?

단위 테스트는 아키텍처 구성 요소에서 격리되어 작동하는 테스트로 정의되지만, 그렇다고 해서 다른 서비스를 호출하거나 일부 네트워크 통신을 사용할 수 있는 구성 요소가 테스트에 포함되지 않는 것은 아닙니다.

많은 서버리스 애플리케이션에는 클라우드에서도 격리하여 테스트할 수 있는 아키텍처 구성 요소가 있습니다. 한 가지 예는 입력을 받아 데이터를 처리하고 Amazon SQS 대기열로 메시지를 보내는 Lambda 함수입니다. 이 함수의 단위 테스트는 입력 값이 대기열에 있는 메시지에 특정 값을 표시하는지 여부를 테스트할 수 있습니다.

Arrange, Act, Assert 패턴을 사용하여 작성된 테스트를 고려하세요.

- Arrange: 리소스(메시지 수신을 위한 대기열 및 테스트 중인 함수)를 할당합니다.
- Act: 테스트 중인 함수를 호출합니다.
- Assert: 함수에서 보낸 메시지를 검색하고 출력을 검증합니다.

모의 테스트 접근 방식에는 처리 중인 모의 객체로 대기열을 모의하고 Lambda 함수 코드가 포함된 클래스 또는 모듈의 처리 중인 인스턴스를 생성하는 것이 포함됩니다. Assert 단계에서는 대기열에 있는 메시지가 모의 객체에서 검색됩니다.

클라우드 기반 접근 방식에서는 테스트를 위해 Amazon SQS 대기열을 생성하고 격리된 Amazon SQS 대기열을 출력 대상으로 사용하도록 구성된 환경 변수와 함께 Lambda 함수를 배포합니다. Lambda 함수를 실행한 후 테스트는 Amazon SQS 대기열에서 메시지를 검색합니다.

클라우드 기반 테스트는 동일한 코드를 실행하고, 동일한 동작을 어설션하고, 애플리케이션의 기능적 정확성을 검증합니다. 그러나 Lambda 함수의 설정, 즉 IAM 역할, IAM 정책, 함수의 제한 시간 및 메모리 설정을 검증할 수 있다는 추가적인 이점이 있습니다.

다음 단계 및 리소스

다음 리소스를 사용하여 자세히 알아보고 실제 테스트 예제를 살펴보세요.

샘플 구현

GitHub의 Serverless Test Samples 리포지토리에는 이 가이드에 설명된 패턴 및 모범 사례를 따르는 테스트의 구체적인 예제가 포함되어 있습니다. 리포지토리에는 이전 섹션에서 설명한 모의, 에뮬레이션 및 클라우드 테스트 프로세스의 샘플 코드와 안내된 설명이 포함되어 있습니다. 이 리포지토리를 사용하여 AWS의 최신 서버리스 테스트 지침을 빠르게 확인하세요.

참조 자료

Serverless Land를 방문하여 AWS 서버리스 기술에 대한 최신 블로그, 비디오 및 교육에 액세스하세요.

다음 AWS 블로그 게시물도 읽어 보는 것이 좋습니다.

- Accelerating serverless development with AWS SAM Accelerate(AWS 블로그 게시물)
- Increasing development speed with CDK Watch(AWS 블로그 게시물)
- Mocking service integrations with AWS Step Functions Local(AWS 블로그 게시물)
- Getting started with testing serverless applications(AWS 블로그 게시물)

도구

- AWS SAM - 서버리스 애플리케이션 테스트 및 디버깅
- AWS SAM – 자동 테스트와 통합
- Lambda – Lambda 콘솔에서 Lambda 함수 테스트

Node.js를 사용하여 Lambda 함수 빌드

AWS Lambda에서 Node.js로 JavaScript 코드를 실행할 수 있습니다. Lambda는 이벤트 처리를 위해 코드를 실행하는 Node.js를 위한 런타임을 제공합니다. 코드는 사용자가 관리하는 AWS Identity and Access Management(IAM) 역할의 자격 증명을 사용하여 AWS SDK for JavaScript가 포함된 환경에서 실행됩니다. Node.js 런타임에 포함된 SDK 버전에 대해 자세히 알아보려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

Lambda는 다음과 같은 Node.js 런타임을 지원합니다.

Node.js

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024년 6월 12일	2025년 2월 28일	2025년 3월 31일

> **Note**
>
> Node.js 18 및 이후의 런타임은 JavaScript v3용 AWS SDK를 사용합니다. 이전 런타임에서 함수를 마이그레이션하려면 GitHub에서 마이그레이션 워크숍을 따르세요. JavaScript 버전 3용 AWS SDK에 대한 자세한 내용은 모듈식 AWS SDK for JavaScript 정식 출시 블로그 게시물을 참조하세요.

Node.js 함수를 만들려면

1. Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. 다음 설정을 구성합니다:

- 함수 이름: 함수의 이름을 입력합니다.
- 런타임: Node.js 20.x를 선택합니다.

4. 함수 생성을 선택합니다.
5. 테스트 이벤트를 구성하려면 테스트를 선택합니다.
6. 이벤트 이름에 **test**를 입력합니다.
7. Save changes(변경 사항 저장)를 선택합니다.
8. 함수를 호출하려면 테스트를 선택합니다.

콘솔은 `index.js` 또는 `index.mjs`(이)라는 단일 소스 파일로 Lambda 함수를 생성합니다. 이 파일을 편집하고 기본 제공 코드 편집기에서 더 많은 파일을 추가할 수 있습니다. 변경 사항을 저장하려면 [Save]를 선택합니다. 그런 다음 코드를 실행하려면 테스트를 선택합니다.

> **Note**
>
> Lambda 콘솔은 AWS Cloud9를 사용하여 브라우저에서 통합 개발 환경(IDE)을 제공합니다. AWS Cloud9을 사용하면 자신의 환경에서 Lambda 함수를 개발할 수도 있습니다. 자세한 내용은 AWS Cloud9 사용 설명서의 Working with AWS Lambda functions using the AWS Toolkit을 참조하세요.

`index.js` 또는 `index.mjs` 파일은 이벤트 객체와 컨텍스트 객체를 취하는 `handler`라는 이름의 함수를 내보냅니다. 이는 함수가 호출될 때 Lambda가 호출하는 핸들러 함수입니다. Node.js 함수 런타임은 Lambda에서 호출 이벤트를 가져와 핸들러로 전달합니다. 함수 구성에서 핸들러 값은 `index.handler`입니다.

함수 코드를 저장하면 Lambda 콘솔에서 .zip 파일 아카이브 배포 패키지를 만듭니다. 콘솔 외부에서(IDE를 사용해) 함수 코드를 개발하는 경우 Lambda 함수에 코드를 업로드하려면 배포 패키지를 생성해야 합니다.

> **Note**
>
> 로컬 환경에서 애플리케이션 개발을 시작하려면 이 가이드의 GitHub 리포지토리에서 사용할 수 있는 샘플 애플리케이션 중 하나를 배포하세요.

Node.js의 샘플 Lambda 애플리케이션

- blank-nodejs – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Node.js 함수입니다.
- nodejs-apig - API Gateway의 이벤트를 처리하고 HTTP 응답을 반환하는 퍼블릭 API 엔드포인트가 있는 함수입니다.
- efs-nodejs – Amazon VPC에서 Amazon EFS 파일 시스템을 사용하는 함수입니다. 이 샘플에는 Lambda와 함께 사용하도록 구성된 VPC, 파일 시스템, 마운트 대상 및 액세스 포인트가 포함되어 있습니다.

함수 런타임은 호출 이벤트 외에도 컨텍스트 객체를 핸들러에 전달합니다. 컨텍스트 객체에는 호출, 함수 및 실행 환경에 관한 추가 정보가 포함되어 있습니다. 자세한 내용은 환경 변수에서 확인할 수 있습니다.

Lambda 함수는 CloudWatch Logs 로그 그룹을 함께 제공됩니다. 함수 런타임은 각 호출에 대한 세부 정보를 CloudWatch Logs에 보냅니다. 호출 중 함수가 출력하는 로그를 전달합니다. 함수가 오류를 반환하면 Lambda은 오류에 서식을 지정한 후 이를 호출자에게 반환합니다.

주제

- Node.js 초기화
- 런타임에 포함된 SDK 버전
- TCP 연결에 연결 유지 사용
- CA 인증서 로딩
- Node.js에서 Lambda 함수 핸들러 정의
- .zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포
- 컨테이너 이미지를 사용하여 Node.js Lambda 함수 배포
- AWS Lambda 컨텍스트 객체(Node.js)
- AWS Lambda 함수 로깅(Node.js)
- AWS Lambda에서 Node.js 코드 계측

Node.js 초기화

Node.js에는 초기화 동작을 다른 런타임과 다르게 만드는 고유한 이벤트 루프 모델이 있습니다. 구체적으로, Node.js는 비동기 작업을 지원하는 비차단 I/O 모델을 사용합니다. 이 모델을 사용하면 Node.js가 대부분의 워크로드에서 효율적으로 수행할 수 있습니다. 예를 들어, Node.js 함수가 네트워크 호출을 수행하는 경우 해당 요청이 비동기 작업으로 지정되어 콜백 대기열에 배치될 수 있습니다. 이 함수는 네트워크 호출이 반환될 때까지 기다림으로써 차단되는 경우 없이 기본 호출 스택 내의 다른 작업을 계속 처리할 수 있습니다. 네트워크 호출이 완료되면 해당 콜백이 실행된 다음 콜백 대기열에서 제거됩니다.

일부 초기화 태스크는 비동기로 실행될 수 있습니다. 이러한 비동기 태스크는 호출 전에 실행을 완료할 것을 보장하지 않습니다. 예를 들어, AWS 파라미터 스토어에서 파라미터를 가져오기 위해 네트워크 호출을 만드는 코드는 Lambda가 핸들러 함수를 실행할 때까지 완료되지 않을 수 있습니다. 따라서 호출하는 동안 변수가 null일 수 있습니다. 이를 방지하려면 나머지 함수의 핵심 비즈니스 로직을 계속하기 전에 변수 및 기타 비동기 코드가 완전히 초기화되었는지 확인하세요.

또는 함수 코드를 ES 모듈로 지정하여 함수 핸들러 범위 밖의 파일 최상위 레벨에서 `await`를 사용할 수 있습니다. 모든 `Promise`에 대해 `await`를 실행할 경우 핸들러 호출 전에 비동기 초기화 코드가 완료되어 콜드 스타트 대기 시간의 프로비저닝된 동시성이 가지는 효과를 극대화해 줍니다. 자세한 내용과 예제는 AWS Lambda에서 Node.js ES 모듈 및 최상위 레벨 대기를 참조하세요.

함수 핸들러를 ES 모듈로 지정

기본적으로 Lambda는 `.js` 접미사가 있는 파일을 CommonJS 모듈로 취급합니다. 선택적으로 코드를 ES 모듈로 지정할 수 있습니다. 이 작업은 두 가지 방법으로 수행할 수 있습니다. 즉, 함수의 `package.json` 파일에서 `type`을 `module`로 지정하거나 `.mjs` 파일명 확장자를 사용할 수 있습니다. 첫 번째 방법에서는 함수 코드가 모든 `.js` 파일을 ES 모듈로 취급하고, 두 번째 시나리오에서는 `.mjs`를 사용하여 지정한 파일만 ES 모듈입니다. `.mjs` 파일은 항상 ES 모듈이며 `.cjs` 파일은 항상 CommonJS 모듈이므로 ES 모듈과 CommonJS 모듈의 이름을 각각 `.mjs` 및 `.cjs`로 지정하여 모듈을 혼합할 수 있습니다.

Lambda가 ES 모듈을 로드할 때 `NODE_PATH` 환경 변수에서 폴더를 검색합니다. ES 모듈 `import` 명령문을 사용하여 런타임에 포함된 AWS SDK를 로드할 수 있습니다. 계층에서 ES 모듈을 로드할 수도 있습니다.

런타임에 포함된 SDK 버전

Node.js 런타임에 포함된 AWS SDK 버전은 런타임 버전 및 사용자의 AWS 리전에 따라 달라집니다. 사용 중인 런타임에 포함된 SDK 버전을 찾으려면 다음 코드를 사용하여 Lambda 함수를 생성합니다.

> **Note**
>
> Node.js 버전 18 이상에 대한 아래 예제 코드에서는 CommonJS 형식을 사용합니다. Lambda 콘솔에서 함수를 생성하는 경우 코드가 포함된 파일의 이름을 `index.js`으로 변경해야 합니다.

Example Node.js 18 이상

```
const { version } = require("@aws-sdk/client-s3/package.json");

exports.handler = async () => ({ version });
```

다음과 같은 형식으로 응답을 반환합니다.

```
{
  "version": "3.462.0"
}
```

TCP 연결에 연결 유지 사용

기본 Node.js HTTP/HTTPS 에이전트는 모든 새 요청에 대해 새로운 TCP 연결을 생성합니다. 새 연결을 설정하는 데 드는 비용을 피하기 위해 `keepAlive: true`를 사용하여 함수가 AWS SDK for JavaScript로 만든 연결을 재사용할 수 있습니다. 연결 유지는 SDK를 사용하여 여러 API를 호출하는 Lambda 함수의 요청 시간을 줄일 수 있습니다.

JavaScript 3.x용 AWS SDK(`nodejs18.x` 및 그 이후 Lambda 런타임에 포함)에서 keep-alive는 기본으로 활성화되어 있습니다. 연결 유지를 비활성화하려면 AWS SDK for JavaScript 3.x Developer Guide의 Reusing connections with keep-alive in Node.js를 참조하세요. 연결 유지 사용에 대한 자세한 내용은 AWS Developer Tools Blog의 HTTP keep-alive is on by default in modular AWS SDK for JavaScript를 참조하세요.

CA 인증서 로딩

Node.js 18까지의 Node.js 런타임 버전에서, Lambda는 Amazon 전용 CA(인증 기관) 인증서를 자동으로 로딩하여 다른 AWS 서비스와 상호 작용하는 함수를 더 쉽게 생성할 수 있습니다. 예를 들어, Lambda에는 Amazon RDS 데이터베이스에 설치된 서버 ID 인증서를 검증하는 데 필요한 Amazon RDS 인증서가 포함되어 있습니다. 이 동작은 콜드 스타트 동안 성능에 영향을 미칠 수 있습니다.

Node.js 20부터 Lambda는 더 이상 기본적으로 추가 CA 인증서를 로딩하지 않습니다. Node.js 20 런타임에는 `/var/runtime/ca-cert.pem`에 위치한 모든 Amazon CA 인증서가 있는 인증서 파일이 들어 있습니다. Node.js 18 및 이전 런타임에서 동일한 동작을 복원하려면 `NODE_EXTRA_CA_CERTS` 환경 변수를 `/var/runtime/ca-cert.pem`에 설정하십시오.

최적의 성능을 위해 필요한 인증서만 배포 패키지와 함께 번들로 묶고 `NODE_EXTRA_CA_CERTS` 환경 변수를 통해 로딩하는 것이 좋습니다. 인증서 파일은 PEM 형식으로 된 하나 이상의 신뢰할 수 있는 루트 또는 중간 CA 인증서로 구성되어야 합니다. 예를 들어, RDS의 경우 코드 옆에 필수 인증서를 `certificates/rds.pem`으로 포함합니다. 그런 다음 `NODE_EXTRA_CA_CERTS`를 `/var/task/certificates/rds.pem`에 설정하여 인증서를 로딩합니다.

Node.js에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

다음 예제 함수는 이벤트 객체의 내용을 로깅하고 로그의 위치를 반환합니다.

> **Note**
>
> 이 페이지에서는 CommonJS 및 ES 모듈 핸들러의 예를 보여줍니다. 이 두 핸들러 유형의 차이에 대해 자세히 알아보려면 함수 핸들러를 ES 모듈로 지정을 참조하세요.

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;
};
```

함수를 구성할 때, 핸들러 설정의 값은 파일의 이름과 내보낸 핸들러 모듈의 이름이며 점으로 구분됩니다. 콘솔의 기본값은 예를 들어, 이 가이드에서는 `index.handler`입니다. 이는 `handler` 파일에서 내보낸 `index.js` 메서드를 나타냅니다.

런타임은 인수를 핸들러 메서드에 전달합니다. 첫 번째 인수는 호출자로부터의 정보가 포함된 `event` 객체입니다. 이 정보는 Invoke를 호출할 때 호출자가 JSON 형식 문자열로 전달하고, 런타임은 이 정보를 객체로 변환합니다. AWS 서비스가 함수를 호출할 때, 이벤트 구조는 서비스별로 다릅니다.

두 번째 인수는 컨텍스트 객체이며, 여기에는 호출, 함수 및 실행 환경에 대한 정보가 포함되어 있습니다. 이전 예제에서는, 함수가 컨텍스트 객체로부터 로그 스트림의 이름을 가져와서 호출자에게 반환합니다.

응답을 전송하기 위해 비동기 핸들러에서 호출할 수 있는 함수인 콜백 인수를 사용할 수도 있습니다. 콜백 대신 비동기/대기를 사용하는 것이 좋습니다. 비동기/대기는 향상된 가독성, 오류 처리 및 효율성을 제공합니다. 비동기/대기 및 콜백 간의 차이에 대한 더 자세한 내용은 콜백 사용을 참조하십시오.

이름 지정

함수를 구성할 때, 핸들러 설정의 값은 파일의 이름과 내보낸 핸들러 모듈의 이름이며 점으로 구분됩니다. 콘솔에서 생성된 함수의 기본값은 예를 들어, 이 가이드에서는 `index.handler`입니다. 이는 `index.js` 또는 `index.mjs` 파일에서 내보낸 `handler` 메서드를 나타냅니다.

콘솔에서 다른 파일 이름 또는 함수 핸들러 이름을 사용하여 함수를 생성하는 경우 기본 핸들러 이름을 편집해야 합니다.

함수 핸들러 이름 변경(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. Code(코드) 탭을 선택합니다.
3. 아래로 스크롤하여 런타임 설정 창으로 이동한 다음 편집을 선택합니다.
4. 핸들러에서 함수 핸들러의 새 이름을 입력합니다.
5. Save(저장)를 선택합니다.

비동기/대기 사용

비동기식 작업을 수행하는 코드의 경우, 동기/대기 패턴을 사용하여 핸들러 실행을 완료하세요. 비동기/대기는 중첩된 콜백이나 체인 프라미스 없이 Node.js 내에서 비동기 코드를 작성할 수 있는 간결하고 읽기 쉬운 방법입니다. 비동기/대기를 사용하면 비동기 및 비차단 상태를 유지하면서 동기 코드처럼 읽는 코드를 작성할 수 있습니다.

`async` 키워드는 함수를 비동기로 표시하고 `await` 키워드는 `Promise`이 해결될 때까지 함수 실행을 일시 중지합니다.

> **Note**
>
> 비동기 이벤트가 완료될 때까지 기다려야 합니다. 비동기 이벤트가 완료되기 전에 함수가 반환되면 함수가 실패하거나 애플리케이션에서 예상치 못한 동작이 발생할 수 있습니다. 이는

forEach 루프에 비동기 이벤트가 포함되어 있을 때 발생할 수 있습니다. forEach 루프에는 동기 호출이 필요합니다. 자세한 내용은 Mozilla 설명서의 Array.prototype.forEach()를 참조하세요.

ES module handler

Example – async/await로 HTTP 요청

```
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/await로 HTTP 요청

```
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;
};
```

다음 예에서는 비동기/대기를 사용하여 Amazon Simple Storage Service 버킷을 나열합니다.

> **Note**
>
> 이 예제를 사용하기 전에 함수의 실행 역할에 Amazon S3 읽기 권한이 있는지 확인하십시오.

ES module handler

Example — 비동기/대기 기능이 있는 AWS SDK v3

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 AWS SDK for JavaScript v3를 사용합니다.

```
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 — 비동기/대기 기능이 있는 AWS SDK v3

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 AWS SDK for JavaScript v3를 사용합니다.

```
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;
};
```

콜백 사용

콜백을 사용하는 대신 비동기/대기를 사용하여 함수 핸들러를 선언하는 것이 좋습니다. 비동기/대기는 다음과 같은 여러 가지 이유로 더 나은 선택입니다.

- 가독성: 비동기/대기 코드는 콜백 코드보다 읽기 쉽고 이해하기 쉬우므로 따르기가 금방 어려워지고 콜백 문제가 발생할 수 있습니다.
- 디버깅 및 오류 처리: 콜백 기반 코드를 디버깅하는 것은 어려울 수 있습니다. 콜 스택은 추적하기 어려워지고 오류는 쉽게 삼킬 수 있습니다. 비동기/대기를 사용하면 try/catch 블록을 사용하여 오류를 처리할 수 있습니다.
- 효율성: 콜백은 종종 코드의 다른 부분을 전환해야 합니다. 비동기/대기는 컨텍스트 전환 수를 줄여 코드 효율성을 높일 수 있습니다.

핸들러에서 콜백을 사용하는 경우, 이벤트 루프가 비어 있거나 함수 제한 시간을 초과할 때까지 함수가 계속 실행됩니다. 응답은 모든 이벤트 루프 작업이 완료될 때까지 호출자에게 전송되지 않습니다. 함수 제한 시간을 초과하면, 대신 오류가 반환됩니다. context.callbackWaitsForEmptyEventLoop를 false로 설정하여 즉시 응답을 전송하도록 런타임을 구성할 수 있습니다.

콜백 함수는 두 개의 인수, `Error` 및 응답을 사용합니다. 응답 객체는 `JSON.stringify`와 호환되어야 합니다.

다음 예제 함수는 URL을 확인하고 상태 코드를 호출자에게 반환합니다.

ES module handler

Example – callback으로 HTTP 요청

```
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 – callback으로 HTTP 요청

```
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));
    });
};
```

아래 예제에서는, Amazon S3로부터의 응답이 사용 가능해지는 즉시 호출자에게 반환됩니다. 이벤트 루프에서 실행 중인 제한 시간은 동결되고 다음에 함수가 호출될 때 계속 실행됩니다.

> **Note**
>
> 이 예제를 사용하기 전에 함수의 실행 역할에 Amazon S3 읽기 권한이 있는지 확인하십시오.

ES module handler

Example — callbackWaitsForEmptyEventLoop 기능이 있는 AWS SDK v3

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 AWS SDK for JavaScript v3를 사용합니다.

```
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 기능이 있는 AWS SDK v3

이 예제에서는 `nodejs18.x` 이상 런타임에서 사용할 수 있는 AWS SDK for JavaScript v3를 사용합니다.

```
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 파일 아카이브를 사용하여 Node.js Lambda 함수 배포

AWS Lambda 함수의 코드는 코드가 의존하는 추가 패키지 및 모듈과 함께 함수의 핸들러 코드를 포함하는 .js 또는 .mjs 파일로 구성됩니다. Lambda에 이 함수 코드를 배포하려면 배포 패키지를 사용합니다. 이 패키지는 .zip 파일 아카이브 또는 컨테이너 이미지일 수 있습니다. Node.js에서 컨테이너 이미지를 사용하는 방법에 대한 자세한 내용은 컨테이너 이미지를 사용하여 Node.js Lambda 함수 배포를 참조하세요.

배포 패키지를 .zip 파일 아카이브로 생성하려면 명령줄 도구의 기본 제공 .zip 파일 아카이브 유틸리티 또는 7zip과 같은 기타 .zip 파일 유틸리티를 사용합니다. 다음 섹션에 표시된 예제에서는 Linux 또는 MacOS 환경에서 명령줄 `zip` 도구를 사용한다고 가정합니다. Windows에서 동일한 명령을 사용하려면 Windows Subsystem for Linux를 설치하여 Ubuntu 및 Bash의 Windows 통합 버전을 가져옵니다.

Lambda는 POSIX 파일 권한을 사용하므로 .zip 파일 아카이브를 생성하기 전에 배포 패키지 폴더에 대한 권한을 설정해야 할 수 있습니다.

주제

Node.js의 런타임 종속 항목

Node.js 런타임을 사용하는 Lambda 함수의 경우 종속 항목은 모든 Node.js 모듈일 수 있습니다. Node.js 런타임에는 여러 공통 라이브러리와 AWS SDK for JavaScript의 버전이 포함되어 있습니다. `nodejs16.x` Lambda 런타임에는 SDK 버전 2.x가 포함됩니다. 런타임 버전 `nodejs18.x` 및 그 이상에는 SDK 버전 3이 포함됩니다. 런타임 버전 `nodejs18.x` 및 그 이상에서 SDK 버전 2를 사용하려면 .zip 파일 배포 패키지에 SDK를 추가하십시오. 선택한 런타임에 사용 중인 SDK 버전이 포함된 경우 .zip 파일에 SDK 라이브러리를 포함할 필요가 없습니다. 사용 중인 런타임에 포함된 SDK 버전을 찾으려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

Lambda는 최신 기능 및 보안 업그레이드를 포함하도록 Node.js 런타임에서 SDK 라이브러리를 주기적으로 업데이트합니다. Lambda는 런타임에 포함된 다른 라이브러리에도 보안 패치와 업데이트를 적용합니다. 패키지의 종속 항목을 완벽하게 제어하려면 런타임에 포함된 종속 항목의 원하는 버전을 배

포 패키지에 추가하면 됩니다. 예를 들어 JavaScript용 SDK의 특정 버전을 사용하려는 경우 .zip 파일에 종속 항목으로 포함할 수 있습니다. .zip 파일에 런타임 포함 종속 항목을 추가하는 방법에 대한 자세한 내용은 종속 항목 검색 경로 및 런타임 포함 라이브러리 단원을 참조하세요.

AWS Shared Responsibility Model에서는 사용자가 함수의 배포 패키지에 있는 모든 종속 항목을 관리해야 합니다. 여기에는 업데이트 및 보안 패치 적용이 포함됩니다. 함수의 배포 패키지에서 종속 항목을 업데이트하려면 먼저 새 .zip 파일을 생성한 다음 Lambda에 업로드합니다. 자세한 내용은 종속 항목이 있는 .zip 배포 패키지 생성 및 .zip 파일을 사용하여 Node.js Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 없는 .zip 배포 패키지 생성

함수 코드에 Lambda 런타임에 포함된 라이브러리를 제외하고 종속 항목이 없는 경우 .zip 파일에는 함수의 핸들러 코드가 포함된 `index.js` 또는 `index.mjs` 파일만 포함됩니다. 선호하는 zip 유틸리티를 사용하여 루트에 `index.js` 또는 `index.mjs` 파일이 있는 .zip 파일을 생성합니다. 핸들러 코드가 포함된 파일이 .zip 파일의 루트에 없는 경우 Lambda가 코드를 실행할 수 없습니다.

.zip 파일을 배포하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트하는 방법을 알아보려면 .zip 파일을 사용하여 Node.js Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 있는 .zip 배포 패키지 생성

함수 코드가 Lambda Node.js 런타임에 포함되지 않은 패키지 또는 모듈에 종속되어 있는 경우, 함수 코드와 함께 이러한 종속 항목을 .zip 파일에 추가하거나 Lambda 계층을 사용할 수 있습니다. 이 섹션의 지침에서는 .zip 배포 패키지에 종속 항목을 포함하는 방법을 보여줍니다. 계층에 종속 항목을 포함하는 방법에 대한 지침은 the section called “종속 항목을 위한 Node.js 계층 생성” 섹션을 참조하세요.

다음 예제 CLI 명령은 함수의 핸들러 코드와 해당 종속 항목이 포함된 `index.js` 또는 `index.mjs` 파일이 포함된 `my_deployment_package.zip`이라는 이름의 .zip 파일을 생성합니다. 이 예제에서는 npm 패키지 관리자를 사용하여 종속 항목을 설치합니다.

배포 패키지를 만드는 방법

1. `index.js` 또는 `index.mjs` 소스 코드 파일이 포함된 프로젝트 디렉터리로 이동합니다. 이 예에서 디렉터리 이름은 `my_function`입니다.

   ```
   cd my_function
   ```

2. `npm install` 명령을 사용하여 함수의 필수 라이브러리를 `node_modules` 디렉터리에 설치합니다. 이 예제에서는 AWS X-Ray SDK for Node.js를 설치합니다.

```
npm install aws-xray-sdk
```

그러면 다음과 유사한 폴더 구조가 생성됩니다.

```
~/my_function
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

배포 패키지에 직접 만든 사용자 지정 모듈을 추가할 수도 있습니다. node_modules 아래에 모듈 이름으로 디렉터리를 만들고 사용자 지정 작성 패키지를 저장합니다.

3. 루트에 프로젝트 폴더의 콘텐츠가 포함된 .zip 파일을 만듭니다. r (재귀) 옵션을 사용하여 zip이 하위 폴더를 압축하는지 확인합니다.

```
zip -r my_deployment_package.zip .
```

종속 항목을 위한 Node.js 계층 생성

이 섹션의 지침은 계층에 종속 항목을 포함하는 방법을 보여줍니다. 배포 패키지에 종속 항목을 포함하는 방법에 대한 지침은 the section called “종속 항목이 있는 .zip 배포 패키지 생성” 섹션을 참조하세요.

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 /opt 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 PATH 변수에는 /opt 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. PATH 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

- nodejs/node_modules
- nodejs/node16/node_modules (NODE_PATH)
- nodejs/node18/node_modules (NODE_PATH)
- nodejs/node20/node_modules (NODE_PATH)

예를 들어, 계층.zip 파일 구조는 다음과 같을 수 있습니다.

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

또한 Lambda는 /opt/lib 디렉터리의 모든 라이브러리와 /opt/bin 디렉터리의 모든 바이너리를 자동으로 감지합니다. Lambda가 계층 콘텐츠를 제대로 찾을 수 있도록 다음 구조로 계층을 생성할 수도 있습니다.

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

계층을 패키징한 후 the section called “계층 생성 및 삭제” 및 the section called “계층 추가”을 참조하여 계층 설정을 완료합니다.

종속 항목 검색 경로 및 런타임 포함 라이브러리

Node.js 런타임에는 여러 공통 라이브러리와 AWS SDK for JavaScript의 버전이 포함되어 있습니다. 런타임에 포함된 라이브러리의 다른 버전을 사용하려면 함수와 함께 번들링하거나 배포 패키지에 종속 항목으로 추가하면 됩니다. 예를 들어 다른 버전의 SDK를 .zip 배포 패키지에 추가하여 사용할 수 있습니다. 함수의 Lambda 계층에 포함할 수도 있습니다.

코드에서 import 또는 require 문을 사용하면 Node.js 런타임은 모듈을 찾을 때까지 NODE_PATH 경로의 디렉터리를 검색합니다. 기본적으로 런타임에서 검색하는 첫 번째 위치는.zip 배포 패키지가 압축 해제되고 탑재되는 디렉터리입니다(/var/task). 배포 패키지에 런타임 포함 라이브러리 버전을 포함하는 경우 이 버전이 런타임에 포함된 버전보다 우선합니다. 배포 패키지의 종속 항목도 계층의 종속 항목보다 우선합니다.

계층에 종속 항목을 추가하면 Lambda는 이 종속 항목을 /opt/nodejs/nodexx/node_modules로 추출합니다. 여기서 nodexx는 사용 중인 런타임 버전을 나타냅니다. 검색 경로에서 이 디렉터리는 런타임 포함 라이브러리(/var/lang/lib/node_modules)가 있는 디렉터리보다 우선합니다. 따라서 함수 계층의 라이브러리는 런타임에 포함된 버전보다 우선합니다.

다음 코드 줄을 추가하면 Lambda 함수에 대한 전체 검색 경로를 확인할 수 있습니다.

```
console.log(process.env.NODE_PATH)
```

.zip 패키지 내의 별도 폴더에 종속 항목을 추가할 수도 있습니다. 예를 들어 .zip 패키지의 `common`이라는 폴더에 사용자 지정 모듈을 추가할 수 있습니다. .zip 패키지를 압축 해제하고 탑재하면 `/var/task` 디렉터리 내에 이 폴더가 배치됩니다. 코드에서 .zip 배포 패키지의 폴더에 있는 종속 항목을 사용하려면 CJS 또는 ESM 모듈 해결 방법을 사용하는지 여부에 따라 `import { } from` 또는 `const { } = require()` 문을 사용합니다. 예:

```
import { myModule } from './common'
```

코드를 `esbuild`, `rollup` 등으로 번들링하는 경우 함수에서 사용하는 종속 항목이 하나 이상의 파일에 함께 번들링됩니다. 가급적 이 방법을 사용하여 종속 항목을 제공하는 것이 좋습니다. 코드를 번들링하면 배포 패키지에 종속 항목을 추가할 때보다 I/O 작업이 줄어들어 성능이 향상됩니다.

.zip 파일을 사용하여 Node.js Lambda 함수 생성 및 업데이트

.zip 배포 패키지를 생성한 후에는 이를 사용하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트할 수 있습니다. Lambda 콘솔, AWS Command Line Interface 및 Lambda API를 사용하여 .zip 패키지를 배포할 수 있습니다. AWS Serverless Application Model(AWS SAM) 및 AWS CloudFormation을 사용하여 Lambda 함수를 생성하고 업데이트할 수도 있습니다.

Lambda용 .zip 배포 패키지의 최대 크기는 250MB(압축 해제됨)입니다. 이 제한은 Lambda 계층을 포함하여 업로드하는 모든 파일의 합산 크기에 적용됩니다.

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 `chmod` 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

콘솔을 사용하여.zip 파일로 함수 생성 및 업데이트

새 함수를 생성하려면 먼저 콘솔에서 함수를 생성한 다음.zip 아카이브를 업로드해야 합니다. 기존 함수를 업데이트하려면 함수에 대한 페이지를 연 다음 동일한 절차에 따라 업데이트된 .zip 파일을 추가합니다.

.zip 파일이 50MB 미만인 경우 로컬 컴퓨터에서 직접 파일을 업로드하여 함수를 생성하거나 업데이트할 수 있습니다. 50MB보다 큰 .zip 파일의 경우 먼저 패키지를 Amazon S3 버킷에 업로드해야 합니다. AWS Management Console을 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 Amazon S3 시작하기를 참조하세요. AWS CLI를 사용하여 파일을 업로드하려면 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

새 함수 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.
3. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름에 함수 이름을 입력합니다.
   b. 런타임에서 사용할 런타임을 선택합니다.
   c. (선택 사항) 아키텍처에서 함수에 대한 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수에 대한 .zip 배포 패키지가 선택한 명령 세트 아키텍처와 호환되는지 확인합니다.
4. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 실행 역할을 사용할 수 있습니다.
5. 함수 생성을 선택합니다. Lambda에서 선택한 런타임을 사용하여 기본 'Hello World' 함수를 생성합니다.

로컬 시스템에서 .zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서.zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. .zip 파일을 선택합니다.
5. .zip 파일을 업로드하려면 다음을 수행합니다.
   a. 업로드를 선택한 다음 파일 선택기에서.zip 파일을 선택합니다.
   b. Open을 선택합니다.
   c. Save(저장)를 선택합니다.

Amazon S3 버킷에서.zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서 새 .zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. Amazon S3 위치를 선택합니다.
5. .zip 파일의 Amazon S3 링크 URL을 붙여 넣고 저장을 선택합니다.

콘솔 코드 편집기를 사용하여.zip 파일 함수 업데이트

.zip 배포 패키지를 사용하는 일부 함수의 경우 Lambda 콘솔의 기본 제공 코드 편집기를 사용하여 함수 코드를 직접 업데이트할 수 있습니다. 이 기능을 사용하려면 함수가 다음 조건을 충족해야 합니다.

- 함수에서 해석된 언어 런타임(Python, Node.js 또는 Ruby) 중 하나를 사용해야 합니다.
- 함수의 배포 패키지가 3MB보다 작아야 합니다.

컨테이너 이미지 배포 패키지가 있는 함수의 함수 코드는 콘솔에서 직접 편집할 수 없습니다.

콘솔 코드 편집기를 사용하여 함수 코드 업데이트

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 소스 코드 파일을 선택하고 통합 코드 편집기에서 편집합니다.

4. 코드 편집을 마치면 배포를 선택하여 변경 사항을 저장하고 함수를 업데이트합니다.

AWS CLI를 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CLI를 사용하여 새 함수를 생성하거나.zip 파일로 기존 함수를 업데이트할 수 있습니다. create-function 및 update-function-code 명령을 사용하여 .zip 패키지를 배포합니다. .zip 파일이 50MB보다 작은 경우 로컬 빌드 시스템의 파일 위치에서 .zip 패키지를 업로드할 수 있습니다. 더 큰 파일의 경우 Amazon S3 버킷에서 .zip 패키지를 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> AWS CLI를 사용하여 Amazon S3 버킷에서 .zip 파일을 업로드하는 경우 버킷은 함수와 동일한 AWS 리전에 있어야 합니다.

AWS CLI에서 .zip 파일을 사용하여 새 함수를 생성하려면 다음을 지정해야 합니다.

- 함수의 이름(`--function-name`)
- 함수의 런타임(`--runtime`)
- 함수의 실행 역할(`--role`)의 Amazon 리소스 이름(ARN)
- 함수 코드에 있는 핸들러 메서드의 이름(`--handler`)

.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
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 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--code` 옵션을 사용합니다. 버전이 지정된 객체에만 `S3ObjectVersion` 파라미터를 사용해야 합니다.

```
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
```

CLI를 사용하여 기존 함수를 업데이트하려면 `--function-name` 파라미터를 사용하여 함수 이름을 지정합니다. 함수 코드를 업데이트하는 데 사용할.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--s3-bucket` 및 `--s3-key` 옵션을 사용합니다. 버전이 지정된 객체에만 `--s3-object-version` 파라미터를 사용해야 합니다.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API를 사용하여.zip 파일로 함수 생성 및 업데이트

.zip 파일 아카이브를 사용하여 함수를 생성하고 업데이트하려면 다음 API 작업을 사용합니다.

- CreateFunction
- UpdateFunctionCode

AWS SAM을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS Serverless Application Model(AWS SAM)은 AWS에서 서버리스 애플리케이션을 빌드하고 실행하는 프로세스를 간소화하는 데 도움이 되는 도구 키트입니다. YAML 또는 JSON 템플릿에서 애플리케이션의 리소스를 정의하고 AWS SAM Command Line Interface(AWS SAM CLI)를 사용하여 애플리케이션을 빌드, 패키징 및 배포합니다. AWS SAM 템플릿에서 Lambda 함수를 빌드하면 AWS SAM은 함수 코드와 사용자가 지정하는 종속 항목을 사용하여 .zip 배포 패키지 또는 컨테이너 이미지를 자동으로 생성합니다. AWS SAM을 사용하여 Lambda 함수를 빌드하고 배포하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Getting started with AWS SAM을 참조하세요.

AWS SAM을 사용하여 기존 .zip 파일 아카이브로 Lambda 함수를 생성할 수도 있습니다. AWS SAM을 사용하여 Lambda 함수를 생성하려면 Amazon S3 버킷 또는 빌드 시스템의 로컬 폴더에 .zip 파일을 저장할 수 있습니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS SAM 템플릿에서 `AWS::Serverless::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `CodeUri` - 함수 코드의 Amazon S3 URI, 로컬 폴더 경로 또는 FunctionCode 객체로 설정됨
- `Runtime` - 선택한 런타임으로 설정됨

AWS SAM을 사용하면 .zip 파일이 50MB보다 큰 경우 Amazon S3 버킷에 먼저 파일을 업로드할 필요가 없습니다. AWS SAM은 로컬 빌드 시스템의 위치에서 허용되는 최대 크기 250MB(압축 해제)까지 .zip 패키지를 업로드할 수 있습니다.

AWS SAM에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS SAM 개발자 안내서의 AWS::Serverless::Function을 참조하세요.

AWS CloudFormation을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CloudFormation을 사용하여 .zip 파일 아카이브로 Lambda 함수를 생성할 수 있습니다. .zip 파일에서 Lambda 함수를 생성하려면 먼저 Amazon S3 버킷에 파일을 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `Code` - `S3Bucket` 및 `S3Key` 필드에 Amazon S3 버킷 이름과 .zip 파일 이름을 입력합니다.
- `Runtime` - 선택한 런타임으로 설정됨

AWS CloudFormation에서 생성하는 .zip 파일은 4MB를 초과할 수 없습니다. AWS CloudFormation에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function을 참조하세요.

컨테이너 이미지를 사용하여 Node.js Lambda 함수 배포

Node.js Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- Node.js용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.

- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Node.js용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Node.js용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

주제

- AWSNode.js용 기본 이미지
- Node.js용 AWS 기본 이미지 사용
- 런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

AWSNode.js용 기본 이미지

AWS는 Node.js에 대한 다음과 같은 기본 이미지를 제공합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
20	Node.js 20	Amazon Linux 2023	GitHub의 Node.js 20용 Dockerfile	
18	Node.js 18	Amazon Linux 2	GitHub의 Node.js 18용 Dockerfile	
16	Node.js 16	Amazon Linux 2	GitHub의 Node.js 16용 Dockerfile	2024년 6월 12일

Amazon ECR 리포지토리: gallery.ecr.aws/lambda/nodejs

Node.js 20 이상의 기본 이미지는 Amazon Linux 2023 최소 컨테이너 이미지를 기반으로 합니다. 이전 기본 이미지는 Amazon Linux 2를 사용합니다. AL2023은 작은 배포 공간과 `glibc`와 같이 업데이트된 라이브러리 버전을 포함하여 Amazon Linux 2에 비해 여러 가지 이점을 제공합니다.

AL2023 기반 이미지는 `microdnf`(`dnf` 심볼릭 링크)를 Amazon Linux 2에서 기본 패키지 관리자인 `yum` 대신 패키지 관리자로 사용합니다. `microdnf`는 `dnf`의 독립 실행형 구현입니다. AL2023 기반 이미지에 포함된 패키지 목록의 경우 Comparing packages installed on Amazon Linux 2023 Container Images의 Minimal Container 열을 참조하세요. AL2023과 Amazon Linux 2의 차이점에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Introducing the Amazon Linux 2023 runtime for AWS Lambda를 참조하세요.

> **Note**
>
> AWS Serverless Application Model(AWS SAM)을 포함하여 AL2023 기반 이미지를 로컬에서 실행하려면 Docker 버전 20.10.10 이상을 사용해야 합니다.

Node.js용 AWS 기본 이미지 사용

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker(Node.js 20 이상 기본 이미지의 경우 최소 버전 20.10.10)
- Node.js

기본 이미지에서 이미지 생성

Node.js용 AWS 기본 이미지에서 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

   ```
   mkdir example
   cd example
   ```

2. npm을 사용하여 새 Node.js 프로젝트를 생성합니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 Enter을 누릅니다.

   ```
   npm init
   ```

3. index.js라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

   Example CommonJS 핸들러

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
           body: JSON.stringify('Hello from Lambda!'),
       };
       return response;
   };
   ```

4. 함수가 AWS SDK for JavaScript 외의 다른 라이브러리를 사용하는 경우 npm을 사용하여 패키지에 해당 라이브러리를 추가합니다.
5. 다음 구성으로 새 Dockerfile을 생성합니다.

   - FROM 속성을 기본 이미지의 URI로 설정합니다.
   - COPY 명령을 사용하여 함수 코드와 런타임 종속성을 Lambda 정의 환경 변수인 {LAMBDA_TASK_ROOT}에 복사합니다.
   - CMD 인수를 Lambda 함수 핸들러로 설정합니다.

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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 이 예제에서 docker-image는 이미지 이름이고 test는 태그입니다.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

이 명령은 이미지를 컨테이너로 실행하고 localhost:9000/2015-03-31/functions/function/invocations에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

2. 새 터미널 창에서 로컬 엔드포인트에 이벤트를 게시합니다.

   Linux/macOS

   Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 컨테이너 ID를 가져옵니다.

   ```
   docker ps
   ```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - `111122223333`를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API을 확장합니다.

npm 패키지 관리자를 사용하여 Node.js 런타임 인터페이스 클라이언트를 설치합니다.

```
npm install aws-lambda-ric
```

GitHub에서 Node.js 런타임 인터페이스 클라이언트를 다운로드할 수도 있습니다. 런타임 인터페이스 클라이언트는 다음 NodeJS 버전을 지원합니다.

- 14.x
- 16.x
- 18.x
- 20..x

다음 예제에서는 비 AWS 기본 이미지를 사용하여 Node.js용 컨테이너 이미지를 빌드하는 방법을 보여 줍니다. 예제 Dockerfile에서는 `buster` 기본 이미지를 사용합니다. Dockerfile에는 런타임 인터페이스 클라이언트가 포함되어 있습니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker
- Node.js

대체 기본 이미지에서 이미지 생성

비 AWS 기본 이미지에서 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir example
cd example
```

2. npm을 사용하여 새 Node.js 프로젝트를 생성합니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

```
npm init
```

3. `index.js`라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

Example CommonJS 핸들러

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 새 Dockerfile을 생성합니다. 다음 Dockerfile은 AWS 기본 이미지 대신 `buster` 기본 이미지를 사용합니다. Dockerfile에는 이미지가 Lambda와 호환되도록 하는 런타임 인터페이스 클라이언트가 포함되어 있습니다. Dockerfile은 다단계 빌드를 사용합니다. 첫 번째 단계에서는 함수의 종속 항목이 설치되는 표준 Node.js 환경인 빌드 이미지를 생성합니다. 두 번째 단계에서는 함수 코드와 해당 종속 항목을 포함하는 더 슬림한 이미지를 생성합니다. 이렇게 하면 최종 이미지 크기가 줄어듭니다.

   - `FROM` 속성을 기본 이미지 식별자로 설정합니다.
   - `COPY` 명령을 사용하여 함수 코드와 런타임 종속 항목을 복사합니다.
   - Docker 컨테이너가 시작될 때 실행할 모듈로 `ENTRYPOINT`를 설정합니다. 이 경우 모듈은 런타임 인터페이스 클라이언트입니다.
   - `CMD` 인수를 Lambda 함수 핸들러로 설정합니다.

   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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. 에뮬레이터를 이미지에 빌드하거나 다음 절차를 사용하여 로컬 시스템에 설치할 수 있습니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 설치 및 실행

1. 프로젝트 디렉터리에서 다음 명령을 실행하여 GitHub에서 런타임 인터페이스 에뮬레이터(x86-64 아키텍처)를 다운로드하고 로컬 시스템에 설치합니다.

   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 에뮬레이터를 설치하려면 이전 명령의 GitHub 리포지토리 URL을 다음과 같이 바꿉니다.

   ```
   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 에뮬레이터를 설치하려면 `$downloadLink`을(를) 다음과 같이 바꿉니다.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `/usr/local/bin/npx aws-lambda-ric index.handler`는 Docker 파일의 `CMD` 다음에 오는 `ENTRYPOINT`입니다.

   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
   ```

   이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

3. 로컬 엔드포인트에 이벤트를 게시합니다.

   Linux/macOS

   Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. 컨테이너 ID를 가져옵니다.

   ```
   docker ps
   ```

5. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - `111122223333`를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

AWS Lambda 컨텍스트 객체(Node.js)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러.에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다.

컨텍스트 메서드

- `getRemainingTimeInMillis()` – 실행 시간이 초과되기까지 남은 시간(밀리초)을 반환합니다.

컨텍스트 속성

- `functionName` – Lambda 함수의 이름입니다.
- `functionVersion` – 함수의 버전입니다.
- `invokedFunctionArn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `memoryLimitInMB` – 함수에 할당된 메모리의 양입니다.
- `awsRequestId` – 호출 요청의 식별자입니다.
- `logGroupName` – 함수에 대한 로그 그룹입니다.
- `logStreamName` – 함수 인스턴스에 대한 로그 스트림입니다.
- `identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
  - `cognitoIdentityId`— 인증된 Amazon Cognito ID입니다.
  - `cognitoIdentityPoolId` – 호출에 대한 권한을 부여한 Amazon Cognito ID 풀입니다.
- `clientContext` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.
  - `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` – 클라이언트 애플리케이션에 의해 지정되는 사용자 지정 값입니다.
- `callbackWaitsForEmptyEventLoop` – Node.js 이벤트 루프가 빌 때까지 대기하는 대신, 콜백이 실행될 때 즉시 응답을 보내려면 false로 설정합니다. 이것이 false인 경우, 대기 중인 이벤트는 다음 번 호출 중에 계속 실행됩니다.

다음 예제 함수는 컨텍스트 정보를 로깅하고 로그의 위치를 반환합니다.

Example index.js 파일

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

AWS Lambda 함수 로깅(Node.js)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그를 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 호출에 관한 세부 정보를 로그 스트림에 전송하며, 함수 코드에서 로그 및 그 외 출력을 중계합니다. 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 단원을 참조하십시오.

이 페이지에서는 AWS Command Line Interface, Lambda 콘솔 또는 CloudWatch 콘솔을 사용하여 Lambda 함수 코드의 로그 출력이나 액세스 로그를 생성하는 방법에 대해 설명합니다.

단원

- 로그를 반환하는 함수 생성
- Node.js에서 Lambda 고급 로깅 제어 사용
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제

로그를 반환하는 함수 생성

함수 코드의 로그를 출력하려면, 콘솔 객체에서 메서드를 사용하거나, `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용합니다. 다음 예제는 환경 변수의 값과 이벤트 객체를 로깅합니다.

Example index.js 파일 - 로깅

```
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 로그 형식

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 라인을 로그합니다. 함수에 의해 로그된 각 항목에는 타임스탬프, 요청 ID, 로그 수준을 추가합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

Lambda 콘솔, CloudWatch Logs 콘솔 또는 명령줄에서 로그를 볼 수 있습니다.

Node.js에서 Lambda 고급 로깅 제어 사용

함수의 로그를 캡처, 처리 및 사용하는 방법을 더 잘 제어할 수 있도록 지원되는 Node.js 런타임에 대한 다음의 로깅 옵션을 구성할 수 있습니다.

- 로그 형식 - 함수 로그의 경우 일반 텍스트와 구조화된 JSON 형식 중에서 선택
- 로그 수준 - JSON 형식의 로그의 경우, Lambda가 Amazon CloudWatch로 전송하는 로그의 세부 수준(ERROR, DEBUG 또는 INFO 등)을 선택
- 로그 그룹 - 함수가 로그를 보내는 CloudWatch 로그 그룹을 선택

이러한 로깅 옵션에 대한 자세한 내용과 이를 사용하도록 함수를 구성하는 방법에 대한 지침은 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”을 참조하세요.

Node.js Lambda 함수에서 로그 형식 및 로그 수준 옵션을 사용하려면 다음 섹션의 지침을 참조하세요.

Node.js에서 구조화된 JSON 로그 사용

함수의 로그 형식으로 JSON을 선택하면 Lambda는 `console.trace`, `console.debug`, `console.log`, `console.info`, `console.error` 및 `console.warn`의 콘솔 메서드를 사용gksms 로그 출력을 구조화된 JSON으로 CloudWatch에 전송합니다. 각 JSON 로그 객체에는 다음 키가 있는 4개의 키 값 페어가 포함되어 있습니다.

- `"timestamp"` - 로그 메시지가 생성된 시간
- `"level"` - 메시지에 할당된 로그 수준
- `"message"` - 로그 메시지의 내용
- `"requestId"` - 함수 간접 호출의 고유한 요청 ID

함수에서 사용하는 로깅 방법에 따라 이 JSON 객체에 추가 키 페어가 포함될 수도 있습니다. 예를 들어 함수가 `console` 메서드를 통해 여러 인수를 사용하여 오류 객체를 기록하는 경우 JSON 객체에는 키 `errorMessage`, `errorType`, `stackTrace`와 함께 추가 키 값 페어가 포함됩니다.

코드에서 이미 AWS Lambda용 Powertools 같은 다른 로깅 라이브러리를 사용하여 JSON 구조화된 로그를 생성하는 경우에는 변경할 필요가 없습니다. Lambda는 이미 JSON으로 인코딩된 로그를 이중 인코딩하지 않으므로 함수의 애플리케이션 로그는 이전과 같이 계속 캡처됩니다.

AWS Lambda용 Powertools 로깅 패키지를 사용하여 Node.js 런타임에서 JSON 구조의 로그를 생성하는 방법에 대한 자세한 내용은 the section called “로깅”을 참조하세요.

JSON 형식의 로그 출력 예제

다음 예제는 함수의 로그 형식을 JSON으로 설정할 때 단일 및 다중 인수를 포함한 `console` 메서드를 사용하여 생성된 다양한 로그 출력이 CloudWatch Logs에 캡처되는 방법을 보여줍니다.

첫 번째 예제에서는 `console.error` 메서드를 사용하여 간단한 문자열을 출력합니다.

Example Node.js 로깅 코드

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON 로그 레코드

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

`console` 메서드와 함께 단일 또는 다중 인수를 사용하여 더 복잡한 구조의 로그 메시지를 출력할 수도 있습니다. 다음 예제에서는 `console.log`를 사용하여 단일 인수를 사용하여 두 개의 키 값 페어를 출력합니다. Lambda가 CloudWatch Logs로 전송하는 JSON 객체의 `"message"` 필드는 문자열 필드가 아닙니다.

Example Node.js 로깅 코드

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON 로그 레코드

```
{
```

```
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

다음 예제에서는 `console.log` 메서드를 다시 사용하여 로그 출력을 생성합니다. 이번에는 메서드가 두 개의 인수, 즉 두 개의 키 값 페어를 포함하는 맵과 식별 문자열을 사용합니다. 이 경우에는 두 개의 인수를 제공했으므로 Lambda는 `"message"` 필드를 문자열 필드로 설정합니다.

Example Node.js 로깅 코드

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON 로그 레코드

```
{
    "timestamp": "2023-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda는 `console.log` 로그 수준 INFO를 사용하여 생성된 출력을 할당합니다.

마지막 예제에서는 `console` 메서드를 사용하여 오류 객체를 CloudWatch Logs에 출력하는 방법을 보여줍니다. 여러 인수를 사용하여 오류 객체를 기록하는 경우 Lambda는 `errorMessage`, `errorType`, `stackTrace` 필드를 로그 출력에 추가합니다.

Example Node.js 로깅 코드

```
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 로그 레코드

```
{
    "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)"
    ]
}
```

여러 오류 유형을 로깅하는 경우 `console` 메서드에 제공된 첫 번째 오류 유형에서 추가 필드 `errorMessage`, `errorType`, `stackTrace`가 추출됩니다.

구조화된 JSON 로그가 포함된 임베디드 메트릭 형식(EMF) 클라이언트 라이브러리 사용

AWS는 임베디드 지표 형식 로그(EMF)를 생성하는 데 사용할 수 있는 오픈 소스 클라이언트 라이브러리를 제공합니다. 이러한 라이브러리를 사용하는 기존 함수를 소유하고 있고 함수의 로그 형식을 JSON으로 변경하면 CloudWatch가 코드에서 내보내는 지표를 더 이상 인식하지 못할 수 있습니다.

현재 코드에서 `console.log`를 사용하거나 또는 AWS Lambda용 Powertools(TypeScript)를 사용하여 직접 EMF 로그를 내보내는 경우에도 함수의 로그 형식을 JSON으로 변경하면 CloudWatch에서 해당 로그를 구문분석할 수 없습니다.

> **Important**
>
> CloudWatch에서 함수의 EMF 로그를 계속해서 적절하게 구문분석하려면 EMF 및 AWS Lambda용 Powertools 라이브러리를 최신 버전으로 업데이트합니다. 또한 JSON 로그 형식으로 전환하는 경우 테스트를 수행하여 함수에 내장된 지표와의 호환성을 확인하는 것이 좋습니다. `console.log`를 사용하여 직접 EMF 로그를 내보내는 코드의 경우 다음 코드 예제에서 표시된 것처럼 해당 메트릭을 직접 `stdout`로 출력하도록 코드를 변경합니다.

Example 임베디드 메트릭을 **`stdout`**로 방출하는 코드

```
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.js에서 로그 수준 필터링 사용

AWS Lambda에서 애플리케이션 로그를 로그 수준에 따라 필터링하려면 함수에서 JSON 형식의 로그를 사용해야 합니다. 다음 두 가지 방법으로 이 작업을 달성할 수 있습니다.

- 표준 콘솔 메서드를 사용하여 로그 출력을 생성하고 JSON 로그 형식을 사용하도록 함수를 구성합니다. AWS Lambda은 그런 다음 the section called “Node.js에서 구조화된 JSON 로그 사용”에서 설명하는 JSON 객체의 “레벨” 키 값 쌍을 사용하여 로그 출력을 필터링합니다. 함수의 로그 형식을 구성하는 방법을 알아보려면 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”를 참조하세요.
- 다른 로깅 라이브러리 또는 메서드를 사용하여 로그 출력 수준을 정의하는 “레벨” 키 값 쌍이 포함된 JSON 구조화된 로그를 코드에 만들 수 있습니다. 예를 들어 AWS Lambda용 Powertools를 사용하여 코드로부터 JSON 구조화된 로그 출력을 생성할 수 있습니다. Powertools를 Node.js 런타임과 함께 사용하는 방법에 대한 자세한 내용은 the section called “로깅”을 참조하세요.

  Lambda가 함수 로그를 필터링하려면 JSON 로그 출력에 `"timestamp"` 키 값 쌍도 포함해야 합니다. 시간은 유효한 RFC 3339 타임스탬프 형식으로 지정해야 합니다. 유효한 타임스탬프를 제공하지 않으면 Lambda는 로그에 레벨 INFO를 할당하고 타임스탬프를 추가합니다.

로그 수준 필터링을 사용하도록 함수를 구성할 때는 다음 옵션 중에서 AWS Lambda이 CloudWatch 로그로 전송하기를 원하는 로그 수준을 선택합니다.

로그 수준	표준 사용량
TRACE(최대 세부 정보)	코드 실행 경로를 추적하는 데 사용되는 가장 세밀한 정보
DEBUG	시스템 디버깅에 대한 세부 정보
INFO	함수의 정상 작동을 기록하는 메시지
WARN	해결되지 않을 경우 예상치 못한 동작으로 이어질 수 있는 잠재적 오류에 대한 메시지
ERROR	코드가 예상대로 작동하지 못하게 하는 문제에 대한 메시지

로그 수준	표준 사용량
FATAL(최소 세부 정보)	응용 프로그램 작동을 중지시키는 심각한 오류에 대한 메시지

Lambda는 선택한 수준 이하의 로그만 Cloudwatch로 전송합니다. 예를 들어 로그 수준을 WARN으로 구성하면 Lambda는 WARN, ERROR 및 FATAL에 해당하는 로그를 전송합니다.

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2

- AWS CLI - aws configure을 통한 빠른 구성

AWS CLI를 사용하면 --log-type 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 LogResult 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 LogResult이라는 함수의 my-function 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 base64 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 my-function에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:


```
{
    "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"
}
```

로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

AWS Lambda에서 Node.js 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 두 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS Distro for OpenTelemetry(ADOT) - 안전하게 프로덕션 준비가 된 AWS에서 지원하는 OpenTelemetry(OTEL) SDK의 배포입니다.
- AWS X-Ray SDK for Node.js — 추적 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 어떤 솔루션을 사용하든 X-Ray에서 엔드 투 엔드 추적 기능을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

ADOT를 사용하여 Node.js 함수 계측

ADOT는 OTel SDK를 사용하여 원격 측정 데이터를 수집하는 데 필요한 모든 것을 패키징할 수 있는 완전 관리형 Lambda 계층을 제공합니다. 이 계층을 사용하면 모든 함수 코드를 수정하지 않고도 Lambda 함수를 계측할 수 있습니다. 계층을 구성하여 OTel의 사용자 지정 초기화를 수행할 수도 있습니다. 자세한 내용은 ADOT 설명서의 Lambda에서 ADOT 컬렉터에 대한 사용자 지정 구성을 참조하세요.

Node.js 런타임의 경우 ADOT Node.js용 AWS 관리형 Lambda 계층을 추가하여 함수를 자동으로 계측할 수 있습니다. 이 계층을 추가하는 방법에 대한 자세한 지침은 ADOT 설명서의 AWS Distro for OpenTelemetry Lambda Support for JavaScript를 참조하세요.

X-Ray SDK를 사용하여 Node.js 함수 계측

Lambda 함수가 애플리케이션의 다른 리소스에 대해 수행하는 호출에 대한 세부 정보를 기록하려면 AWS X-Ray SDK for Node.js를 사용할 수도 있습니다. SDK를 가져오려면 애플리케이션의 종속성에 `aws-xray-sdk-core` 패키지를 추가합니다.

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 for JavaScript v3에서 AWS SDK 클라이언트를 계측하려면 `captureAWSv3Client` 메서드를 사용하여 클라이언트 인스턴스를 래핑합니다.

Example blank-nodejs/function/index.js – AWS SDK 클라이언트 추적

```
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 런타임은 X-Ray SDK를 구성하는 몇 가지 환경 변수를 설정합니다. 예를 들어 Lambda는 X-Ray SDK에서 런타임 오류가 발생하지 않도록 AWS_XRAY_CONTEXT_MISSING을(를) LOG_ERROR로 설정합니다. 사용자 지정 컨텍스트 누락 전략을 설정하려면 값이 없도록 함수 구성에서 환경 변수를 재정의합니다. 그러면 컨텍스트 누락 전략을 프로그래밍 방식으로 설정할 수 있습니다.

Example 초기화 코드 예제

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

자세한 내용은 the section called “환경 변수 구성” 단원을 참조하십시오.

올바른 종속성을 추가하고 필요한 코드를 변경한 후 Lambda 콘솔 또는 API를 통해 함수의 구성에서 추적을 활성화합니다.

Lambda 콘솔을 사용하여 추적 활성화

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.

5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

Lambda API를 사용하여 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 Lambda 함수에 대한 추적을 구성하고 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation을 사용하여 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray 추적 해석

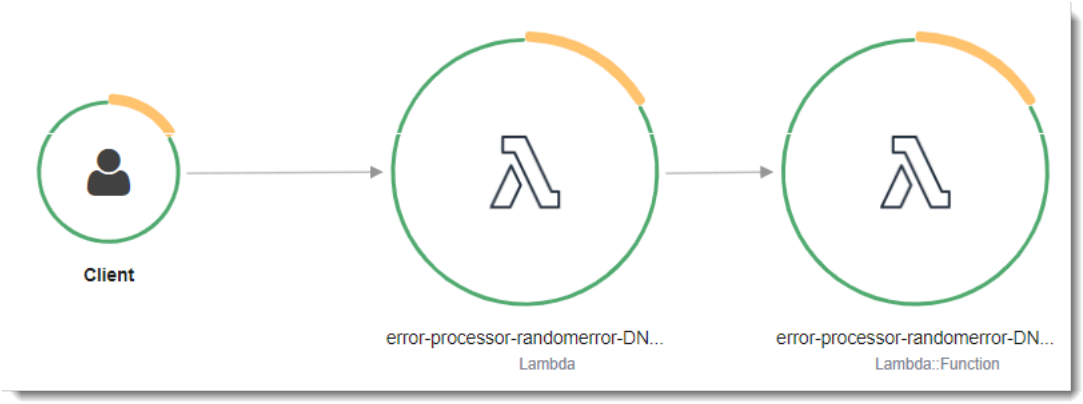
함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

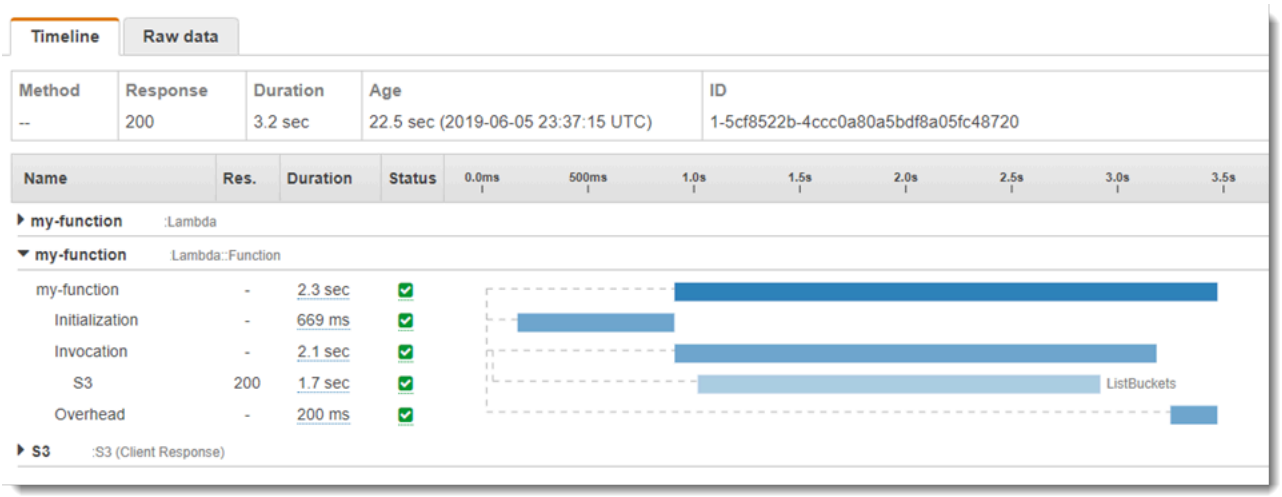
X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. `AWS::Lambda::Function` 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 `AWS::Lambda::Function` 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 AWS X-Ray SDK for Node.js를 참조하십시오.

> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

계층에 런타임 종속성 저장(X-Ray SDK)

X-Ray SDK를 사용하여 AWS SDK 클라이언트를 계측하는 경우 함수 코드와 배포 패키지가 상당히 커질 수 있습니다. 함수 코드를 업데이트할 때마다 런타임 종속성을 업로드하지 않으려면 X-Ray SDK를 Lambda 계층에 패키징합니다.

다음 예제에서는 AWS X-Ray SDK for Node.js를 저장하는 `AWS::Serverless::LayerVersion` 리소스를 보여줍니다.

Example template.yml – 종속성 계층

```
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
```

이 구성을 사용하면 런타임 종속성을 변경하는 경우 라이브러리 계층만 업데이트하면 됩니다. 함수 배포 패키지에는 코드만 포함되어 있으므로 이는 업로드 시간을 줄일 수 있습니다.

종속성 계층을 만들려면 배포 전에 계층 아카이브를 생성하기 위해 빌드를 변경해야 합니다. 사용 가능한 예제는 blank-nodejs 샘플 애플리케이션을 참조하세요.

TypeScript를 사용하여 Lambda 함수 빌드

Node.js 런타임을 사용하여 AWS Lambda에서 TypeScript 코드를 실행할 수 있습니다. Node.js 는 기본적으로 TypeScript 코드를 실행하지 않으므로 먼저 TypeScript 코드를 JavaScript로 트랜스파일해야 합니다. 그런 다음 JavaScript 파일을 사용하여 함수 코드를 Lambda에 배포합니다. 코드는 사용자가 관리하는 AWS Identity and Access Management(IAM) 역할의 자격 증명을 사용하여 JavaScript용 AWS SDK가 포함된 환경에서 실행됩니다. Node.js 런타임에 포함된 SDK 버전에 대해 자세히 알아보려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

Lambda는 다음과 같은 Node.js 런타임을 지원합니다.

Node.js

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Node.js 20	`nodejs20.x`	Amazon Linux 2023			
Node.js 18	`nodejs18.x`	Amazon Linux 2			
Node.js 16	`nodejs16.x`	Amazon Linux 2	2024년 6월 12일	2025년 2월 28일	2025년 3월 31일

주제

- TypeScript 개발 환경 설정
- TypeScript에서 Lambda 함수 핸들러 정의
- .zip 파일 아카이브를 사용하여 Lambda에 트랜스파일된 TypeScript 코드를 배포합니다.
- 컨테이너 이미지를 사용하여 Lambda에 트랜스파일된 TypeScript 코드 배포
- AWS Lambda컨텍스트 객체 입력 TypeScript
- TypeScript의 AWS Lambda 함수 로깅
- 추적 TypeScript 코드 입력 AWS Lambda

TypeScript 개발 환경 설정

로컬 IDE(통합 개발 환경), 텍스트 편집기 또는 AWS Cloud9을 사용하여 TypeScript 함수 코드를 작성합니다. Lambda 콘솔에서는 TypeScript 코드를 생성할 수 없습니다.

TypeScript 코드를 트랜스파일하려면 TypeScript 배포와 함께 번들로 제공되는 esbuild 또는 Microsoft의 TypeScript 컴파일러(`tsc`)와 같은 컴파일러를 설정하세요. AWS Serverless Application Model(AWS SAM) 또는 AWS Cloud Development Kit (AWS CDK)를 사용하여 TypeScript 코드의 빌드 및 배포를 간소화할 수 있습니다. 두 도구 모두 esbuild를 사용하여 TypeScript 코드를 JavaScript로 트랜스파일합니다.

esbuild를 사용할 경우 다음을 고려하세요.

- TypeScript 경고에는 여러 종류가 있습니다.
- 사용하려는 Node.js 런타임과 일치하도록 TypeScript 변환 설정을 구성해야 합니다. 자세한 내용은 esbuild 설명서의 Target(대상)을 참조하세요. Lambda에서 지원하는 특정 Node.js 버전을 타겟팅하는 방법을 보여주는 tsconfig.json 파일의 예는 TypeScript GitHub 저장소를 참조하세요.
- esbuild는 유형 검사를 수행하지 않습니다. 유형을 확인하려면 `tsc` 컴파일러를 사용합니다. 다음 예에 표시된 대로 `tsc -noEmit`을 실행하거나 tsconfig.json 파일에 `"noEmit"` 파라미터를 추가합니다. 이렇게 하면 `tsc`를 구성하여 JavaScript 파일을 방출하지 않습니다. 유형을 확인한 후 esbuild를 사용하여 TypeScript 파일을 JavaScript로 변환합니다.

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"]
```

```
}
```

TypeScript에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

Example TypeScript 핸들러

다음 예제 함수는 이벤트 객체의 내용을 로깅하고 로그의 위치를 반환합니다. 유의할 사항:

- Lambda 함수에서 이 코드를 사용하기 전에 @types/aws-lambda 패키지를 개발 종속 항목으로 추가해야 합니다. 이 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다. `@types/aws-lambda`가 설치되면 `import` 문(`import ... from 'aws-lambda'`)이 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 DefinitelyTyped GitHub 리포지토리의 aws-lambda를 참조하세요.
- 이 예제의 핸들러는 ES 모듈이며 `package.json` 파일에서 또는 `.mjs` 파일 확장명을 사용하여 지정해야 합니다. 자세한 내용은 함수 핸들러를 ES 모듈로 지정을 참조하십시오.

```
import { Handler } from 'aws-lambda';

export const handler: Handler = async (event, context) => {
    console.log('EVENT: \n' + JSON.stringify(event, null, 2));
    return context.logStreamName;
};
```

런타임은 인수를 핸들러 메서드에 전달합니다. 첫 번째 인수는 호출자로부터의 정보가 포함된 `event` 객체입니다. 이 정보는 Invoke를 호출할 때 호출자가 JSON 형식 문자열로 전달하고, 런타임은 이 정보를 객체로 변환합니다. AWS 서비스가 함수를 호출할 때, 이벤트 구조는 서비스별로 다릅니다. TypeScript에서는 이벤트 객체에 유형 주석을 사용하는 것이 좋습니다. 자세한 내용은 이벤트 객체에 대한 유형의 사용 단원을 참조하십시오.

두 번째 인수는 컨텍스트 객체이며, 여기에는 호출, 함수 및 실행 환경에 대한 정보가 포함되어 있습니다. 이전 예제에서는, 함수가 컨텍스트 객체로부터 로그 스트림의 이름을 가져와서 호출자에게 반환합니다.

응답을 전송하기 위해 비동기 핸들러에서 호출할 수 있는 함수인 콜백 인수를 사용할 수도 있습니다. 콜백 대신 비동기/대기를 사용하는 것이 좋습니다. 비동기/대기는 향상된 가독성, 오류 처리 및 효율성을 제공합니다. 비동기/대기 및 콜백 간의 차이에 대한 더 자세한 내용은 콜백 사용을 참조하십시오.

비동기/대기 사용

비동기식 작업을 수행하는 코드의 경우, 동기/대기 패턴을 사용하여 핸들러 실행을 완료하세요. 비동기/대기는 중첩된 콜백이나 체인 프라미스 없이 Node.js 내에서 비동기 코드를 작성할 수 있는 간결하고 읽기 쉬운 방법입니다. 비동기/대기를 사용하면 비동기 및 비차단 상태를 유지하면서 동기 코드처럼 읽는 코드를 작성할 수 있습니다.

`async` 키워드는 함수를 비동기로 표시하고 `await` 키워드는 `Promise`이 해결될 때까지 함수 실행을 일시 중지합니다.

Example TypeScript 함수 - 비동기

이 예제에서는 `nodejs18.x` 런타임에서 사용할 수 있는 `fetch`를 사용합니다. 유의할 사항:

- Lambda 함수에서 이 코드를 사용하기 전에 @types/aws-lambda 패키지를 개발 종속 항목으로 추가해야 합니다. 이 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다. `@types/aws-lambda`가 설치되면 `import` 문(`import ... from 'aws-lambda'`)이 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 DefinitelyTyped GitHub 리포지토리의 aws-lambda를 참조하세요.
- 이 예제의 핸들러는 ES 모듈이며 `package.json` 파일에서 또는 `..mjs` 파일 확장명을 사용하여 지정해야 합니다. 자세한 내용은 함수 핸들러를 ES 모듈로 지정을 참조하십시오.

```
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',
            }),
        };
    }
};
```

콜백 사용

콜백을 사용하는 대신 비동기/대기를 사용하여 함수 핸들러를 선언하는 것이 좋습니다. 비동기/대기는 다음과 같은 여러 가지 이유로 더 나은 선택입니다.

- 가독성: 비동기/대기 코드는 콜백 코드보다 읽기 쉽고 이해하기 쉬우므로 따르기가 금방 어려워지고 콜백 문제가 발생할 수 있습니다.
- 디버깅 및 오류 처리: 콜백 기반 코드를 디버깅하는 것은 어려울 수 있습니다. 콜 스택은 추적하기 어려워지고 오류는 쉽게 삼킬 수 있습니다. 비동기/대기를 사용하면 try/catch 블록을 사용하여 오류를 처리할 수 있습니다.
- 효율성: 콜백은 종종 코드의 다른 부분을 전환해야 합니다. 비동기/대기는 컨텍스트 전환 수를 줄여 코드 효율성을 높일 수 있습니다.

핸들러에서 콜백을 사용하는 경우, 이벤트 루프가 비어 있거나 함수 제한 시간을 초과할 때까지 함수가 계속 실행됩니다. 응답은 모든 이벤트 루프 작업이 완료될 때까지 호출자에게 전송되지 않습니다. 함수 제한 시간을 초과하면, 대신 오류가 반환됩니다. context.callbackWaitsForEmptyEventLoop를 false로 설정하여 즉시 응답을 전송하도록 런타임을 구성할 수 있습니다.

콜백 함수는 두 개의 인수, `Error` 및 응답을 사용합니다. 응답 객체는 `JSON.stringify`와 호환되어야 합니다.

Example 콜백이 있는 TypeScript 함수

이 샘플 함수는 Amazon API Gateway에서 이벤트를 수신하고 이벤트 및 컨텍스트 객체를 로깅한 다음 API Gateway에 응답을 반환합니다. 유의할 사항:

- Lambda 함수에서 이 코드를 사용하기 전에 @types/aws-lambda 패키지를 개발 종속 항목으로 추가해야 합니다. 이 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다. `@types/aws-lambda`가 설치되면 `import` 문(`import ... from 'aws-lambda'`)이 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 DefinitelyTyped GitHub 리포지토리의 aws-lambda를 참조하세요.
- 이 예제의 핸들러는 ES 모듈이며 `package.json` 파일에서 또는 `.mjs` 파일 확장명을 사용하여 지정해야 합니다. 자세한 내용은 함수 핸들러를 ES 모듈로 지정을 참조하십시오.

```
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',
        }),
    });
};
```

이벤트 객체에 대한 유형의 사용

유형을 확인할 수 있는 기능을 잃게 되므로 핸들러 인수 및 반환 유형에 대해 any 유형을 사용하지 않는 것이 좋습니다. 대신 sam local generate-event AWS Serverless Application Model CLI 명령을 사용하여 이벤트를 생성하거나 @types/aws-lambda 패키지에서 제공되는 오픈 소스 정의를 사용합니다.

sam local generate-event 명령을 사용하여 이벤트 생성

1. Amazon Simple Storage Service(Amazon S3) 프록시 이벤트를 생성합니다.

   ```
   sam local generate-event s3 put >> S3PutEvent.json
   ```

2. quicktype 유틸리티를 사용하여 S3PutEvent.json 파일에서 유형 정의를 생성합니다.

   ```
   npm install -g quicktype
   quicktype S3PutEvent.json -o S3PutEvent.ts
   ```

3. 생성된 유형을 코드에 사용합니다.

   ```
   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 패키지에서 오픈 소스 정의를 사용하여 이벤트 생성

1. @types/aws-lambda 패키지를 개발 종속성으로 추가합니다.

```
npm install -D @types/aws-lambda
```

2. 해당 유형을 코드에 사용합니다.

```
import { S3Event } from "aws-lambda";

export const lambdaHandler = async (event: S3Event): Promise<void> => {
  event.Records.map((record) => console.log(record.s3.object.key));
};
```

.zip 파일 아카이브를 사용하여 Lambda에 트랜스파일된 TypeScript 코드를 배포합니다.

에 코드를 배포하려면 먼저 TypeScript 코드를 AWS Lambda 트랜스파일해야 합니다. JavaScript 이 페이지에서는.zip 파일 아카이브를 사용하여 TypeScript 코드를 빌드하고 Lambda에 배포하는 세 가지 방법을 설명합니다.

- AWS Serverless Application Model(AWS SAM) 사용
- AWS Cloud Development Kit (AWS CDK) 사용
- AWS Command Line Interface(AWS CLI) 및 esbuild 사용

AWS SAM함수 구축 및 AWS CDK 배포를 간소화할 수 있습니다. TypeScript AWS SAM 템플릿 사양에서는 서버리스 애플리케이션을 구성하는 Lambda 함수, API, 권한, 구성 및 이벤트를 설명하는 간단하고 깔끔한 구문을 제공합니다. AWS CDK를 사용하면 프로그래밍 언어의 우수한 표현력을 활용하여 클라우드에서 신뢰할 수 있고 확장 가능하며 비용 효율적인 애플리케이션을 빌드할 수 있습니다. AWS CDK는 중급 또는 고급의 숙련도를 갖춘 AWS 사용자를 대상으로 합니다. AWS CDK와 모두 esbuild를 AWS SAM 사용하여 코드를 변환합니다. TypeScript JavaScript

Lambda에 TypeScript 코드를 배포하는 AWS SAM 데 사용

아래 단계에 따라 를 사용하여 샘플 Hello World TypeScript 애플리케이션을 다운로드, 구축 및 배포하십시오. AWS SAM 이 애플리케이션은 기본적인 API 백엔드를 구현합니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트에 GET 요청을 보내면 Lambda 함수가 호출됩니다. 이 함수는 `hello world` 메시지를 반환합니다.

> **Note**
>
> AWS SAMesbuild를 사용하여 TypeScript 코드에서 Node.js Lambda 함수를 생성합니다. esbuild 지원은 현재 공개 프리뷰 중에 있습니다. 공개 평가판으로 제공되는 동안에는 esbuild 지원에 이전 버전과 호환되지 않는 변경 사항이 적용될 수 있습니다.

사전 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS CLI 버전 2

- AWS SAM CLI 버전 1.75 이상
- Node.js 18.x

샘플 AWS SAM 애플리케이션 배포

1. Hello World 템플릿을 사용하여 애플리케이션을 초기화합니다. TypeScript

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. (선택 사항) 샘플 애플리케이션에는 코드 린팅을 위한 ESLint 및 단위 테스트를 위한 Jest 등의 일반적으로 사용되는 도구에 대한 구성이 포함되어 있습니다. lint 및 test 명령 실행

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. 앱을 빌드합니다.

```
cd sam-app
sam build
```

4. 앱을 배포합니다.

```
sam deploy --guided
```

5. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 Enter로 응답합니다.
6. 출력에는 REST API에 대한 엔드포인트가 표시됩니다. 함수를 테스트하려면 브라우저에서 엔드포인트를 엽니다. 다음과 같은 응답이 표시되어야 합니다.

```
{"message":"hello world"}
```

7. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

를 사용하여 Lambda에 TypeScript 코드 AWS CDK 배포하기

아래 단계에 따라 를 사용하여 샘플 TypeScript 애플리케이션을 구축하고 배포하십시오. AWS CDK 이 애플리케이션은 기본적인 API 백엔드를 구현합니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트에 GET 요청을 보내면 Lambda 함수가 호출됩니다. 이 함수는 `hello world` 메시지를 반환합니다.

사전 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS CLI 버전 2
- AWS CDK 버전 2
- Node.js 18.x
- 도커 또는 esbuild

샘플 AWS CDK 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. 앱을 초기화합니다.

   ```
   cdk init app --language typescript
   ```

3. @types/aws-lambda 패키지를 개발 종속성으로 추가합니다. 이 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다.

   ```
   npm install -D @types/aws-lambda
   ```

4. lib 디렉터리를 엽니다. hello-world-stack.ts라는 파일이 보일 것입니다. 이 디렉터리에 hello-world.function.ts와 hello-world.ts라는 2개의 새 파일을 만듭니다.
5. hello-world.function.ts를 열고 파일에 다음 코드를 추가합니다. Lambda 함수의 코드입니다.

> **Note**
>
> `import` 문은 @types/aws-lambda에서 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 리포지토리의 aws-lambda를 참조하십시오. DefinitelyTyped GitHub

```
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.ts를 열고 파일에 다음 코드를 추가합니다. 여기에는 Lambda 함수를 생성하는 NodejsFunction 구문과 LambdaRestApi REST API를 생성하는 구문이 포함됩니다.

```
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` 구문에서는 기본적으로 다음 사항을 가정합니다.

- 함수 핸들러는 `handler`라고 부릅니다.
- 함수 코드가 포함된 .ts 파일(hello-world.function.ts)은 구문을 포함하는 .ts 파일(hello-world.ts)과 동일한 디렉터리에 있습니다. 이 구문은 구문의 ID('hello-world') 및 Lambda 핸들러 파일 이름('function')을 사용하여 함수 코드를 찾습니다. 예를 들어, 함수 코드가 hello-world.my-function.ts라는 파일에 있는 경우 hello-world.ts 파일은 다음과 같은 함수 코드를 참조해야 합니다.

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

이 동작을 변경하고 다른 esbuild 파라미터를 구성할 수 있습니다. 자세한 내용은 AWS CDK API 참조의 esbuild 구성을 참조하세요.

7. .ts를 엽니다hello-world-stack. 이 코드는 AWS CDK 스택을 정의하는 코드입니다. 코드를 다음으로 바꿉니다.

```
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` 파일이 들어 있는 `hello-world` 디렉터리에서 애플리케이션을 배포합니다.

```
cdk deploy
```

9. AWS CDK는 esbuild를 사용하여 Lambda 함수를 빌드하고 패키징한 다음, 함수를 Lambda 런타임에 배포합니다. 출력에는 REST API에 대한 엔드포인트가 표시됩니다. 함수를 테스트하려면 브라우저에서 엔드포인트를 엽니다. 다음과 같은 응답이 표시되어야 합니다.

```
{"message":"hello world"}
```

이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

AWS CLI및 esbuild를 사용하여 Lambda에 TypeScript 코드 배포하기

다음 예제는 esbuild를 사용하여 코드를 트랜스파일하고 TypeScript Lambda에 배포하는 방법을 보여줍니다. esbuild는 모든 종속성이 포함된 하나의 파일을 생성합니다AWS CLI. JavaScript 이 파일이 .zip 아카이브에 추가해야 하는 유일한 파일입니다.

사전 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS CLI 버전 2
- Node.js 18.x
- Lambda 함수용 실행 역할
- Windows 사용자의 경우 7zip과 같은 zip 파일 유틸리티가 필요합니다.

샘플 함수 배포

1. 로컬 컴퓨터에서 새 함수에 대한 프로젝트 디렉터리를 생성합니다.
2. 선택한 npm 또는 패키지 관리자를 사용하여 새 Node.js 프로젝트를 생성합니다.

```
npm init
```

3. @types/aws-lambda 및 esbuild 패키지를 개발 종속성으로 추가합니다. `@types/aws-lambda` 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다.

```
npm install -D @types/aws-lambda esbuild
```

4. index.ts라는 이름의 새 파일을 만듭니다. 다음 코드를 새 파일에 추가합니다. Lambda 함수에 대한 코드입니다. 이 함수는 `hello world` 메시지를 반환합니다. 이 함수는 어떠한 API Gateway 리소스도 생성하지 않습니다.

> **Note**
>
> `import` 문은 @types/aws-lambda에서 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 리포지토리의 aws-lambda를 참조하십시오. DefinitelyTyped GitHub

```
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. 빌드 스크립트를 package.json 파일에 추가합니다. 이렇게 하면 .zip 배포 패키지를 자동으로 생성하도록 esbuild가 구성됩니다. 자세한 내용은 esbuild 설명서의 빌드 스크립트를 참조하세요.

   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

   이 예제에서 "postbuild" 명령은 7zip 유틸리티를 사용하여.zip 파일을 만듭니다. 선호하는 Windows zip 유틸리티를 사용하고 필요에 따라 명령을 수정하십시오.

   ```
   "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. 패키지를 빌드합니다.

```
npm run build
```

7. .zip 배포 패키지를 사용하여 Lambda 함수를 생성합니다. 강조 표시된 텍스트를 실행 역할의 Amazon 리소스 이름(ARN)으로 바꿉니다.

```
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. 테스트 이벤트를 실행하여 함수가 다음 응답을 반환하는지 확인합니다. API Gateway를 사용하여 이 함수를 호출하려면 REST API를 생성하고 구성합니다.

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

컨테이너 이미지를 사용하여 Lambda에 트랜스파일된 TypeScript 코드 배포

TypeScript 코드를 Node.js 컨테이너 이미지로 AWS Lambda 함수에 배포할 수 있습니다. AWS는 컨테이너 이미지를 빌드하는 데 도움이 되는 Node.js용 기본 이미지를 제공합니다. 이러한 기본 이미지는 Lambda에서 이미지를 실행하는 데 필요한 언어 런타임 및 기타 구성 요소가 미리 로드되어 있습니다. AWS는 컨테이너 이미지를 빌드하는 데 도움이 되는 각 기본 이미지의 Dockerfile을 제공합니다.

커뮤니티 또는 사기업에서 제공하는 기본 이미지를 사용하는 경우 Lambda와 호환될 수 있도록 기본 이미지에 Node.js 런타임 인터페이스 클라이언트(RIC)를 추가해야 합니다.

Lambda는 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터를 제공합니다. Node.js용 AWS 기본 이미지에는 런타임 인터페이스 에뮬레이터가 포함되어 있습니다. Alpine Linux 또는 Debian 이미지와 같은 대체 기본 이미지를 사용하는 경우 에뮬레이터를 이미지에 빌드하거나 로컬 시스템에 설치할 수 있습니다.

Node.js 기본 이미지를 사용하여 TypeScript 함수 코드 빌드 및 패키징

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker
- Node.js 18.x

기본 이미지에서 이미지 생성

Lambda용 AWS 기본 이미지에서 이미지를 생성하려면

1. 로컬 컴퓨터에서 새 함수에 대한 프로젝트 디렉터리를 생성합니다.
2. 선택한 npm 또는 패키지 관리자를 사용하여 새 Node.js 프로젝트를 생성합니다.

```
npm init
```

3. @types/aws-lambda 및 esbuild 패키지를 개발 종속성으로 추가합니다. `@types/aws-lambda` 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다.

```
npm install -D @types/aws-lambda esbuild
```

4. package.json 파일에 빌드 스크립트를 추가합니다.

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. index.ts라는 파일을 새로 생성합니다. 다음 샘플 코드를 새 파일에 추가합니다. Lambda 함수에 대한 코드입니다. 이 함수는 hello world 메시지를 반환합니다.

> **Note**
>
> import 문은 @types/aws-lambda에서 유형 정의를 가져옵니다. 관련 없는 타사 도구인 aws-lambda NPM 패키지는 가져오지 않습니다. 자세한 내용은 DefinitelyTyped GitHub 리포지토리의 aws-lambda를 참조하세요.

```
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. 다음 구성으로 새 Dockerfile을 생성합니다.

   - FROM 속성을 기본 이미지의 URI로 설정합니다.
   - CMD 인수를 설정하여 Lambda 함수 핸들러를 지정합니다.

Example Dockerfile

다음 Dockerfile은 다단계 빌드를 사용합니다. 첫 번째 단계에서는 TypeScript 코드를 JavaScript로 트랜스파일합니다. 두 번째 단계에서는 JavaScript 파일 및 프로덕션 종속성만 포함하는 컨테이너 이미지를 생성합니다.

```
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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 이 예제에서 `docker-image`는 이미지 이름이고 `test`는 태그입니다.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

2. 새 터미널 창에서 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 컨테이너 ID를 가져옵니다.

```
docker ps
```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - --region 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

AWS Lambda컨텍스트 객체 입력 TypeScript

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러.에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다.

컨텍스트 메서드

- `getRemainingTimeInMillis()` – 실행 시간이 초과되기까지 남은 시간(밀리초)을 반환합니다.

컨텍스트 속성

- `functionName` – Lambda 함수의 이름입니다.
- `functionVersion` – 함수의 버전입니다.
- `invokedFunctionArn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `memoryLimitInMB` – 함수에 할당된 메모리의 양입니다.
- `awsRequestId` – 호출 요청의 식별자입니다.
- `logGroupName` – 함수에 대한 로그 그룹입니다.
- `logStreamName` – 함수 인스턴스에 대한 로그 스트림입니다.
- `identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
  - `cognitoIdentityId`— 인증된 Amazon Cognito ID입니다.
  - `cognitoIdentityPoolId` – 호출에 대한 권한을 부여한 Amazon Cognito ID 풀입니다.
- `clientContext` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.
  - `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` – 클라이언트 애플리케이션에 의해 지정되는 사용자 지정 값입니다.
- `callbackWaitsForEmptyEventLoop` – Node.js 이벤트 루프가 빌 때까지 대기하는 대신, 콜백이 실행될 때 즉시 응답을 보내려면 false로 설정합니다. 이것이 false인 경우, 대기 중인 이벤트는 다음 번 호출 중에 계속 실행됩니다.

@types/aws-lambda npm 패키지를 사용하여 컨텍스트 객체를 사용할 수 있습니다.

Example index.ts 파일

다음 예제 함수는 컨텍스트 정보를 로깅하고 로그의 위치를 반환합니다.

> **Note**
>
> Lambda 함수에서 이 코드를 사용하기 전에 @types/aws-lambda 패키지를 개발 종속 항목으로 추가해야 합니다. 이 패키지에는 Lambda에 대한 유형 정의가 들어 있습니다. `@types/aws-lambda`가 설치되면 `import` 문(`import ... from 'aws-lambda'`)이 유형 정의를 가져옵니다. 관련 없는 타사 도구인 `aws-lambda` NPM 패키지는 가져오지 않습니다. 자세한 내용은 리포지토리의 AWS-Lambda를 참조하십시오. DefinitelyTyped GitHub

```
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;
};
```

TypeScript의 AWS Lambda 함수 로깅

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그 항목을 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 간접 호출에 대한 세부 정보와 함수 코드의 기타 출력을 로그 스트림으로 전송합니다. CloudWatch Logs에 대한 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 섹션을 참조하세요.

함수 코드의 로그를 출력하려면 콘솔 객체의 메서드를 사용합니다. 보다 자세한 로깅을 위해 `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용할 수 있습니다.

Sections

- 도구 및 라이브러리
- 구조화된 로깅에 Powertools for AWS Lambda(TypeScript) 및 AWS SAM 사용
- 구조화된 로깅에 Powertools for AWS Lambda(TypeScript) 및 AWS CDK 사용
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용

도구 및 라이브러리

Powertools for AWS Lambda(TypeScript)는 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다. Logger 유틸리티는 JSON으로 구조화된 출력과 함께 모든 함수의 함수 컨텍스트에 대한 추가 정보를 포함하는 Lambda 최적화 로거를 제공합니다. 이 유틸리티를 사용하여 다음을 수행합니다.

- Lambda 컨텍스트, 콜드 스타트 및 구조 로깅 출력에서 JSON으로 주요 필드 캡처
- 지시 시 Lambda 호출 이벤트 로깅(기본적으로 비활성화됨)
- 로그 샘플링을 통해 호출 비율에 대해서만 모든 로그 인쇄(기본적으로 비활성화됨)
- 언제든지 구조화된 로그에 추가 키 추가
- 사용자 지정 로그 포맷터(Bring Your Own Formatter)를 사용하여 조직의 로깅 RFC와 호환되는 구조로 로그 출력

구조화된 로깅에 Powertools for AWS Lambda(TypeScript) 및 AWS SAM 사용

다음 단계에 따라 AWS SAM을 사용하는 통합 Powertools for AWS Lambda(TypeScript) 모듈로 샘플 Hello World TypeScript 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Node.js 18.x 이상
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World TypeScript 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name sam-app
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

로그 보존 관리

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 경과 후 CloudWatch가 로그를 자동으로 삭제하는 보존 기간을 구성하세요. 로그 보존을 설정하려면 AWS SAM 템플릿에 다음을 추가합니다.

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

구조화된 로깅에 Powertools for AWS Lambda(TypeScript) 및 AWS CDK 사용

다음 단계에 따라 AWS CDK을 사용하는 통합 Powertools for AWS Lambda(TypeScript) 모듈로 샘플 Hello World TypeScript 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전

송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Node.js 18.x 이상
- AWS CLI 버전 2
- AWS CDK 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS CDK 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

```
mkdir hello-world
cd hello-world
```

2. 앱을 초기화합니다.

```
cdk init app --language typescript
```

3. @types/aws-lambda 패키지를 개발 종속성으로 추가합니다.

```
npm install -D @types/aws-lambda
```

4. Powertools Logger 유틸리티를 설치합니다.

```
npm install @aws-lambda-powertools/logger
```

5. lib 디렉터리를 엽니다. hello-world-stack.ts라는 이름의 파일이 있어야 합니다. 이 디렉터리에 hello-world.function.ts와 hello-world.ts하는 2개의 새 파일을 만듭니다.
6. hello-world.function.ts를 열고 파일에 다음 코드를 추가합니다. Lambda 함수의 코드입니다.

```
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.ts를 열고 파일에 다음 코드를 추가합니다. 여기에는 Lambda 함수를 생성하고, Powertools용 환경 변수를 구성하고, 로그 보존을 1주일로 설정하는 NodeJSFunction 구성체가 포함됩니다. 또한 REST API를 생성하는 LambdaRestAPI 구성체도 포함되어 있습니다.

```
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.ts를 엽니다. 이 코드는 AWS CDK 스택을 정의하는 코드입니다. 코드를 다음으로 바꿉니다.

```
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. 프로젝트 디렉터리로 돌아갑니다.

```
cd hello-world
```

10. 애플리케이션 배포

```
cdk deploy
```

11. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. API 엔드포인트 호출:

```
curl <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

13. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name HelloWorldStack
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
cdk destroy
```

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

추적 TypeScript 코드 입력 AWS Lambda

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 세 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS OpenTelemetry (ADOT) 용 배포판 — 프로덕션 환경에서 바로 사용할 수 있는 안전한 (oTel) SDK 배포판입니다. AWS OpenTelemetry
- AWS X-Ray SDK for Node.js – 트레이스 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.
- Powertools for AWS Lambda (TypeScript) — 서버리스 모범 사례를 구현하고 개발자 개발 속도를 높이기 위한 개발자 툴킷입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 두 솔루션 중 하나를 사용하여 X-Ray에서 end-to-end 추적을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

- Powertools를 사용하여 AWS Lambda (TypeScript) 및 AWS SAM 추적에 사용
- Powertools를 AWS Lambda (TypeScript) 에 사용하고 추적에는 AWS CDK Powertools를 사용합니다.
- X-Ray 추적 해석

Powertools를 사용하여 AWS Lambda (TypeScript) 및 AWS SAM 추적에 사용

아래 단계에 따라 를 사용하여 Powertools for AWS Lambda (TypeScript) 모듈이 통합된 샘플 Hello World TypeScript 응용 프로그램을 다운로드, 빌드 및 배포하십시오. AWS SAM 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트에 GET 요청을 보내면 Lambda 함수는 임베디드 메트릭 형식을 CloudWatch 사용하여 로그와 지표를 호출하고, 에 전송하고, 추적을 전송합니다. AWS X-Ray 이 함수는 `hello world` 메시지를 반환합니다.

사전 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Node.js 18.x 이상
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World 템플릿을 사용하여 애플리케이션을 초기화하십시오. TypeScript

   ```
   sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs18.x --no-tracing
   ```

2. 앱을 빌드합니다.

   ```
   cd sam-app && sam build
   ```

3. 앱을 배포합니다.

   ```
   sam deploy --guided
   ```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> 권한이 정의되어 있지 않을 HelloWorldFunction 수도 있습니다. 괜찮을까요? , 꼭 입력하세요y.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

추적 출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

Powertools를 AWS Lambda (TypeScript) 에 사용하고 추적에는 AWS CDK Powertools를 사용합니다.

아래 단계에 따라 를 사용하여 Powertools for AWS Lambda (TypeScript) 모듈이 통합된 샘플 Hello World TypeScript 응용 프로그램을 다운로드, 빌드 및 배포하십시오. AWS CDK 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트에 GET 요청을 보내면 Lambda 함수는 임베디드 메트릭 형식을 CloudWatch 사용하여 로그와 지표를 호출하고, 에 전송하고, 추적을 전송합니다. AWS X-Ray 이 함수는 `hello world` 메시지를 반환합니다.

사전 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Node.js 18.x 이상
- AWS CLI 버전 2
- AWS CDK 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS Cloud Development Kit (AWS CDK) 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

```
mkdir hello-world
```

```
cd hello-world
```

2. 앱을 초기화합니다.

```
cdk init app --language typescript
```

3. @types/aws-lambda 패키지를 개발 종속성으로 추가합니다.

```
npm install -D @types/aws-lambda
```

4. Powertools Tracer 유틸리티를 설치합니다.

```
npm install @aws-lambda-powertools/tracer
```

5. lib 디렉터리를 엽니다. .ts라는 파일이 보일 것입니다. hello-world-stack 이 디렉터리에 hello-world.function.ts와 hello-world.ts하는 2개의 새 파일을 만듭니다.

6. hello-world.function.ts를 열고 파일에 다음 코드를 추가합니다. Lambda 함수의 코드입니다.

```
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.ts를 열고 파일에 다음 코드를 추가합니다. 여기에는 Lambda 함수를 생성하고, Powertools의 환경 변수를 NodejsFunction 구성하고, 로그 보존을 1주로 설정하는 구조가 포함되어 있습니다. 또한 LambdaRestApi REST API를 생성하는 구조도 포함됩니다.

```
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.ts를 엽니다. 이 코드는 AWS CDK 스택을 정의하는 코드입니다. 코드를 다음으로 바꿉니다.

```
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. 애플리케이션 배포

```
cd ..
cdk deploy
```

10. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. API 엔드포인트 호출:

```
curl <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

12. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

추적 출력은 다음과 같습니다.

```
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]
```

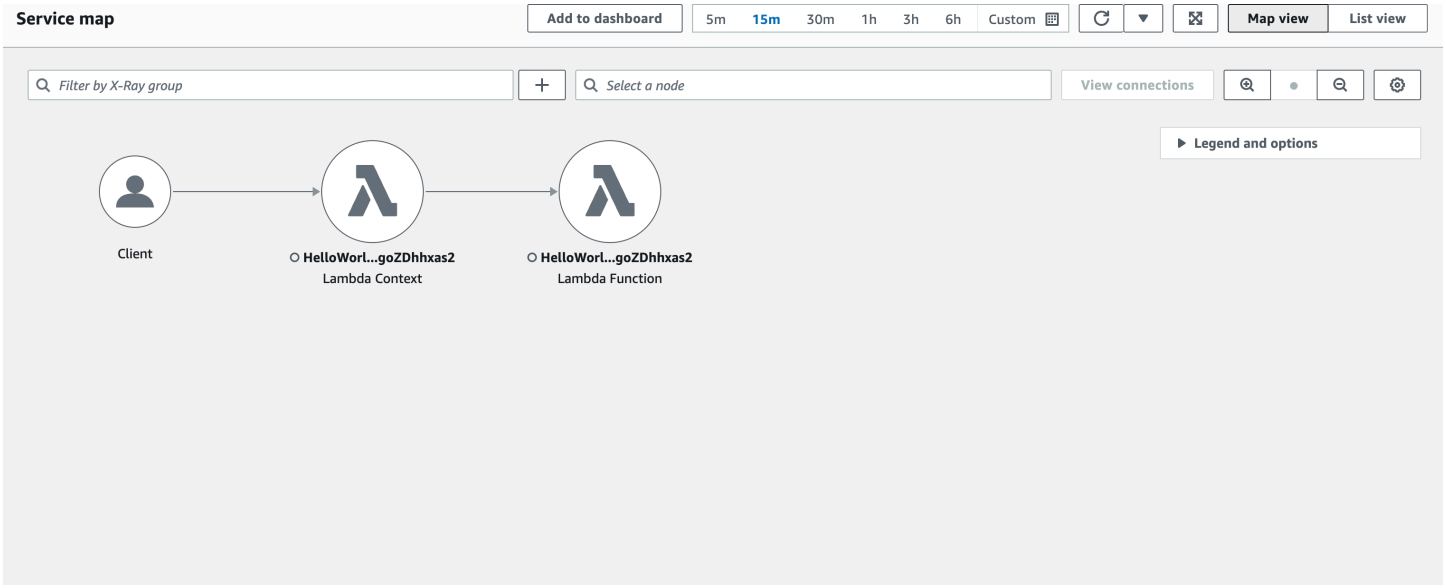
```
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
cdk destroy
```

X-Ray 추적 해석

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 트레이스 맵에서는 애플리케이션 및 모든 구성 요소에 대한 정보를 제공합니다. 다음 예제에서는 샘플 애플리케이션의 추적을 보여줍니다.



Python을 사용하여 Lambda 함수 빌드

AWS Lambda에서 Python 코드를 실행할 수 있습니다. Lambda는 이벤트 처리를 위해 코드를 실행하는 Python을 위한 런타임을 제공합니다. 코드는 사용자가 관리하는 AWS Identity and Access Management(IAM) 역할의 자격 증명을 사용하여 SDK for Python(Boto3)이 포함된 환경에서 실행됩니다. Python 런타임에 포함된 SDK 버전에 대해 자세히 알아보려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

Lambda는 다음과 같은 Python 런타임을 지원합니다.

Python

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
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	2024년 10월 14일	2025년 2월 28일	2025년 3월 31일

> **Note**
>
> 이 표의 런타임 정보는 지속적으로 업데이트됩니다. Lambda의 AWS SDK 사용에 대한 자세한 내용은 Serverless Land의 Managing AWS SDKs in Lambda functions를 참조하세요.

Python 함수를 만들려면

1. Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. 다음 설정을 구성합니다:
   - 함수 이름: 함수의 이름을 입력합니다.
   - 런타임: Python 3.12를 선택합니다.
4. 함수 생성을 선택합니다.
5. 테스트 이벤트를 구성하려면 테스트를 선택합니다.
6. 이벤트 이름에 **test**를 입력합니다.
7. Save changes(변경 사항 저장)를 선택합니다.
8. 함수를 호출하려면 테스트를 선택합니다.

콘솔은 `lambda_function`(이)라는 단일 소스 파일로 Lambda 함수를 생성합니다. 이 파일을 편집하고 기본 제공 코드 편집기에서 더 많은 파일을 추가할 수 있습니다. 변경 사항을 저장하려면 [Save]를 선택합니다. 그런 다음 코드를 실행하려면 테스트를 선택합니다.

> **Note**
>
> Lambda 콘솔은 AWS Cloud9를 사용하여 브라우저에서 통합 개발 환경(IDE)을 제공합니다. AWS Cloud9을 사용하면 자신의 환경에서 Lambda 함수를 개발할 수도 있습니다. 자세한 내용은 AWS Cloud9 사용 설명서의 Working with AWS Lambda functions using the AWS Toolkit을 참조하세요.

> **Note**
>
> 로컬 환경에서 애플리케이션 개발을 시작하려면 이 가이드의 GitHub 리포지토리에서 사용할 수 있는 샘플 애플리케이션 중 하나를 배포하십시오.
>
> Python의 샘플 Lambda 애플리케이션
>
> - blank-python – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Python 함수입니다.

Lambda 함수는 CloudWatch Logs 로그 그룹을 함께 제공됩니다. 함수 런타임은 각 호출에 대한 세부 정보를 CloudWatch Logs에 보냅니다. 호출 중 함수가 출력하는 로그를 전달합니다. 함수가 오류를 반환하면 Lambda은 오류에 서식을 지정한 후 이를 호출자에게 반환합니다.

주제

런타임에 포함된 SDK 버전

Python 런타임에 포함된 AWS SDK 버전은 런타임 버전 및 사용자의 AWS 리전에 따라 달라집니다. 사용 중인 런타임에 포함된 SDK 버전을 찾으려면 다음 코드를 사용하여 Lambda 함수를 생성합니다.

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

응답 형식

Python 3.12 이상 버전의 Python 런타임에서 함수는 JSON 응답의 일부로 유니코드 문자를 반환합니다. 이전 Python 런타임에서는 응답에 유니코드 문자의 이스케이프된 시퀀스를 반환했습니다. 예를 들어, Python 3.11에서 'こんにちは'와 같은 유니코드 문자열을 반환하는 경우 유니코드 문자를 이스케이

프하고 '\u3053\u3093\u306b\u3061\u306f'를 반환합니다. Python 3.12 런타임은 원래 'こんにちは'를 반환합니다.

유니코드 응답을 사용하면 Lambda 응답 크기가 줄어들어 동기 함수에서 더 큰 응답을 최대 6MB의 페이로드 크기에 더 쉽게 맞출 수 있습니다. 이전 예제에서 이스케이프된 버전은 32바이트이며, 유니코드 문자열의 경우 17바이트입니다.

Python 3.12로 업그레이드할 때 새 응답 형식을 고려하여 코드를 조정해야 할 수 있습니다. 직접 호출자에서 이스케이프된 유니코드를 예상하는 경우 반환 함수에 코드를 추가하여 유니코드를 수동으로 이스케이프하거나 직접 호출자가 유니코드 반환을 처리하도록 조정해야 합니다.

확장의 정상 종료

Python 3.12 이상 버전의 Python 런타임에서는 외부 확장을 포함하는 함수에 대해 향상된 정상 종료 기능을 제공합니다. Lambda는 실행 환경을 종료할 때 `SIGTERM` 신호를 런타임에 전송한 다음, `SHUTDOWN` 이벤트를 등록된 각 외부 확장에 전송합니다. Lambda 함수에서 `SIGTERM` 신호를 포착하여 함수에서 생성된 데이터베이스 연결과 같은 리소스를 정리할 수 있습니다.

실행 환경 수명 주기에 대한 자세한 내용은 Lambda 실행 환경 섹션을 참조하세요. 확장과 함께 정상 종료를 사용하는 방법의 예제는 AWS Samples GitHub repository를 참조하세요.

Python에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

Python에서 함수 핸들러를 생성할 때 다음과 같은 일반적인 구문을 사용합니다.

```
def handler_name(event, context):
    ...
    return some_value
```

이름 지정

Lambda 함수를 생성할 때 지정된 Lambda 함수 핸들러 이름은 다음에서 파생됩니다.

- Lambda 핸들러 함수가 있는 파일의 이름.
- Python 핸들러 함수의 이름.

함수 핸들러는 임의의 이름일 수 있지만 Lambda 콘솔의 기본 이름은 `lambda_function.lambda_handler`입니다. 이 함수 핸들러 이름은 함수 이름(`lambda_handler`)과, 핸들러 코드가 저장된 파일(`lambda_function.py`)을 나타냅니다.

콘솔에서 다른 파일 이름 또는 함수 핸들러 이름을 사용하여 함수를 생성하는 경우 기본 핸들러 이름을 편집해야 합니다.

함수 핸들러 이름 변경(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. Code(코드) 탭을 선택합니다.
3. 아래로 스크롤하여 런타임 설정 창으로 이동한 다음 편집을 선택합니다.
4. 핸들러에서 함수 핸들러의 새 이름을 입력합니다.
5. Save(저장)를 선택합니다.

작동 방식

Lambda가 함수 핸들러를 호출할 때 Lambda 런타임은 함수 핸들러에 다음 두 개의 인수를 전달합니다.

- 첫 번째 인수는 이벤트 객체입니다. 이벤트는 처리할 Lambda 함수에 대한 데이터가 포함된 JSON 형식 문서입니다. Lambda 런타임은 이벤트를 객체로 변환한 후 함수 코드에 전달합니다. 이 객체는 일반적으로 Python `dict` 유형입니다. 또한 `list`, `str`, `int`, `float` 또는 `NoneType` 유형이 될 수 있습니다.

  이벤트 객체에는 호출 서비스의 정보가 포함됩니다. 함수를 호출할 때, 이벤트의 구조와 내용을 결정합니다. AWS 서비스는 함수를 호출할 때 이벤트 구조를 정의합니다. AWS 서비스의 이벤트에 대한 자세한 내용은 다른 AWS 서비스의 이벤트로 Lambda 간접 호출 단원을 참조하세요.
- 두 번째 인수는 컨텍스트 객체입니다. 컨텍스트 객체는 런타임에 Lambda에 의해 함수로 전달됩니다. 이 객체는 호출, 함수 및 런타임 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다.

값 반환

선택적으로, 핸들러는 값을 반환할 수 있습니다. 반환된 값은 함수를 호출한 호출 유형 및 서비스에 따라 달라집니다. 예:

- 동기식 호출 같은 `RequestResponse` 호출 유형을 사용하는 경우에는 AWS Lambda가 Python 함수 호출의 결과를 클라이언트에 반환하여 Lambda 함수를 호출합니다(호출 요청에 대한 HTTP 응답이 JSON에 직렬화). 예를 들어 AWS Lambda 콘솔은 `RequestResponse` 호출 유형을 사용하기 때문에 콘솔에서 함수를 호출할 때 콘솔에 반환 값이 표시됩니다.
- 핸들러가 `json.dumps`로 직렬화가 불가능한 객체를 반환하는 경우 런타임에서 오류를 반환하게 됩니다.
- None 문이 포함되지 않은 Python 함수가 묵시적으로 하는 것처럼 핸들러가 `return`을 반환하는 경우, 런타임은 `null`을 반환하게 됩니다.
- `Event` 호출 유형(비동기식 호출)을 사용하는 경우에는 해당 값이 폐기됩니다.

> **Note**
>
> Python 3.9 이상 릴리스에서 Lambda는 호출의 requestId를 오류 응답에 포함합니다.

예제

다음 섹션에서는 Lambda와 함께 사용할 수 있는 Python 함수의 예를 보여줍니다. Lambda 콘솔을 사용하여 함수를 작성하는 경우 이 섹션의 함수를 실행하기 위해 .zip 아카이브 파일을 연결할 필요가 없

습니다. 이러한 함수는 선택한 Lambda 런타임에 포함된 표준 Python 라이브러리를 사용합니다. 자세한 내용은 Lambda 배포 패키지 섹션을 참조하세요.

메시지 반환

다음 예제에서는 `lambda_handler`라는 함수를 보여줍니다. 이 함수는 사용자 입력으로 이름과 성을 받고 입력으로 받은 이벤트의 데이터가 포함된 메시지를 반환합니다.

```
def lambda_handler(event, context):
    message = 'Hello {} {}!'.format(event['first_name'], event['last_name'])
    return {
        'message' : message
    }
```

다음 이벤트 데이터를 사용하여 함수를 호출할 수 있습니다.

```
{
    "first_name": "John",
    "last_name": "Smith"
}
```

응답은 입력으로 전달된 이벤트 데이터를 보여줍니다.

```
{
    "message": "Hello John Smith!"
}
```

응답 구문 분석

다음 예제에서는 `lambda_handler`라는 함수를 보여줍니다. 이 함수는 런타임에 Lambda에 의해 전달된 이벤트 데이터를 사용합니다. JSON 응답에 반환된 `AWS_REGION`에서 환경 변수를 구문 분석합니다.

```
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
        })
    }
```

이벤트 데이터를 사용하여 함수를 호출할 수 있습니다.

```
{
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
```

Lambda 런타임은 초기화 중에 여러 환경 변수를 설정합니다. 런타임 시 응답에 반환되는 환경 변수에 대한 자세한 내용은 단원을 참조하세요Lambda 환경 변수를 사용하여 코드의 값 구성

이 예제의 함수는 Invoke API의 성공적인 응답(200)에 따라 달라집니다. Invoke API 상태에 대한 자세한 내용은 Invoke Response Syntax를 참조하세요.

계산 반환

다음 예제에서는 lambda_handler라는 함수를 보여줍니다. 이 함수는 사용자 입력을 받아들이고 사용자에게 계산을 반환합니다. 이 예제에 대한 자세한 내용은 aws-doc-sdk-examples GitHub 리포지토리를 참조하세요.

```
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
```

다음 이벤트 데이터를 사용하여 함수를 호출할 수 있습니다.

```
{
    "action": "increment",
    "number": 3
}
```

Python Lambda 함수에 대한 .zip 파일 아카이브 작업

AWS Lambda 함수의 코드는 함수의 핸들러 코드가 포함된 .py 파일과 코드가 의존하는 추가 패키지 및 모듈로 구성됩니다. Lambda에 이 함수 코드를 배포하려면 배포 패키지를 사용합니다. 이 패키지는 .zip 파일 아카이브 또는 컨테이너 이미지일 수 있습니다. Python에서 컨테이너 이미지를 사용하는 방법에 대한 자세한 내용은 컨테이너 이미지로 Python Lambda 함수 배포를 참조하세요.

배포 패키지를 .zip 파일 아카이브로 생성하려면 명령줄 도구의 기본 제공 .zip 파일 아카이브 유틸리티 또는 7zip과 같은 기타 .zip 파일 유틸리티를 사용합니다. 다음 섹션에 표시된 예제에서는 Linux 또는 MacOS 환경에서 명령줄 `zip` 도구를 사용한다고 가정합니다. Windows에서 동일한 명령을 사용하려면 Windows Subsystem for Linux를 설치하여 Ubuntu 및 Bash의 Windows 통합 버전을 가져옵니다.

Lambda는 POSIX 파일 권한을 사용하므로 .zip 파일 아카이브를 생성하기 전에 배포 패키지 폴더에 대한 권한을 설정해야 할 수 있습니다.

주제

Python의 런타임 종속 항목

Python 런타임을 사용하는 Lambda 함수의 경우 종속 항목은 모든 Python 패키지 또는 모듈일 수 있습니다. .zip 아카이브를 사용하여 함수를 배포할 때 이러한 종속 항목을 함수 코드와 함께 .zip 파일에 추가하거나 Lambda 계층을 사용할 수 있습니다. 계층은 추가 코드 또는 기타 콘텐츠를 포함할 수 있는 별도의 .zip 파일입니다. Python에서의 Lambda 계층 사용에 대해 자세히 알아보려면 the section called “계층” 섹션을 참조하세요.

Lambda Python 런타임에는 AWS SDK for Python (Boto3)과 해당 종속 항목이 포함됩니다. Lambda는 자체 종속 항목을 추가할 수 없는 배포 시나리오를 위해 런타임에 SDK를 제공합니다. 이러한 시나리오에는 기본 제공 코드 편집기를 사용하여 콘솔에서 함수를 생성하거나 AWS Serverless Application Model(AWS SAM) 또는 AWS CloudFormation 템플릿에서 인라인 함수를 사용하는 것이 포함됩니다.

Lambda는 최신 업데이트 및 보안 패치를 포함하도록 Python 런타임의 라이브러리를 주기적으로 업데이트합니다. 함수가 런타임에 포함된 Boto3 SDK 버전을 사용하지만 배포 패키지에 SDK 종속 항목이 포함된 경우 버전 불일치 문제가 발생할 수 있습니다. 예를 들어, 배포 패키지에 SDK 종속 항목 urllib3이 포함될 수 있습니다. Lambda가 런타임에 SDK를 업데이트할 때 런타임의 새 버전과 배포 패키지의 urllib3 버전 간 호환성 문제로 인해 함수가 실패할 수 있습니다.

> **Important**
>
> 종속 항목을 완전히 제어하고 버전 불일치 문제를 방지하려면 Lambda 런타임에 종속 항목의 버전이 포함되어 있더라도 배포 패키지에 함수의 모든 종속 항목을 추가하는 것이 좋습니다. 여기에는 Boto3 SDK도 포함됩니다.

사용 중인 런타임에 포함된 SDK for Python(Boto3) 버전을 찾으려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

AWS Shared Responsibility Model에서는 사용자가 함수의 배포 패키지에 있는 모든 종속 항목을 관리해야 합니다. 여기에는 업데이트 및 보안 패치 적용이 포함됩니다. 함수의 배포 패키지에서 종속 항목을 업데이트하려면 먼저 새 .zip 파일을 생성한 다음 Lambda에 업로드합니다. 자세한 내용은 종속 항목이 있는 .zip 배포 패키지 생성 및 .zip 파일을 사용하여 Python Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 없는 .zip 배포 패키지 생성

함수 코드에 종속 항목이 없는 경우 .zip 파일에는 함수의 핸들러 코드가 있는 .py 파일만 포함됩니다. 선호하는 zip 유틸리티를 사용하여 루트에 .py 파일이 있는 .zip 파일을 생성합니다. .py 파일이.zip 파일의 루트에 없는 경우 Lambda는 코드를 실행할 수 없습니다.

.zip 파일을 배포하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트하는 방법을 알아보려면 .zip 파일을 사용하여 Python Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 있는 .zip 배포 패키지 생성

함수 코드가 추가 패키지 또는 모듈에 의존하는 경우 이러한 종속 항목을 함수 코드와 함께 .zip 파일에 추가하거나 Lambda 계층을 사용할 수 있습니다. 이 섹션의 지침에서는 .zip 배포 패키지에 종속 항목을 포함하는 방법을 보여줍니다. Lambda에서 코드를 실행하려면 핸들러 코드와 함수의 모든 종속 항목을 포함하는 .py 파일을 .zip 파일의 루트에 설치해야 합니다.

함수 코드가 `lambda_function.py`라는 파일에 저장되어 있다고 가정해 보겠습니다. 다음 예제 CLI 명령은 함수 코드와 해당 종속 항목을 포함하는 `my_deployment_package.zip`라는 .zip 파일을 생

성합니다. 프로젝트 디렉터리의 폴더에 직접 종속 항목을 설치하거나 Python 가상 환경을 사용할 수 있습니다.

배포 패키지 생성(프로젝트 디렉터리)

1. `lambda_function.py` 소스 코드 파일이 들어 있는 프로젝트 디렉터리로 이동합니다. 이 예에서 디렉터리 이름은 `my_function`입니다.

   ```
   cd my_function
   ```

2. 종속 항목을 설치할 package라는 새 디렉터리를 생성합니다.

   ```
   mkdir package
   ```

   .zip 배포 패키지의 경우 Lambda는 소스 코드와 해당 종속 항목이 모두 .zip 파일의 루트에 있을 것으로 예상합니다. 그러나 프로젝트 디렉터리에 종속 항목을 직접 설치하면 많은 수의 새 파일과 폴더가 생겨서 IDE 탐색이 어려워질 수 있습니다. 여기에 별도의 `package` 디렉터리를 생성하여 종속 항목을 소스 코드와 별도로 유지합니다.

3. `package` 디렉터리에 종속 항목을 설치합니다. 아래 예제에서는 pip를 사용하여 Python 패키지 인덱스에서 Boto3 SDK를 설치합니다. 함수 코드에서 사용자가 직접 생성한 Python 패키지를 사용하는 경우 `package` 디렉터리에 해당 Python 패키지를 저장합니다.

   ```
   pip install --target ./package boto3
   ```

4. 설치된 라이브러리가 포함된 .zip 파일을 루트에 생성합니다.

   ```
   cd package
   zip -r ../my_deployment_package.zip .
   ```

   그러면 프로젝트 디렉터리에 `my_deployment_package.zip` 파일이 생성됩니다.

5. .zip 파일의 루트에 lambda_function.py 파일을 추가합니다.

   ```
   cd ..
   zip my_deployment_package.zip lambda_function.py
   ```

   .zip 파일은 다음과 같이 함수의 핸들러 코드와 모든 종속 항목 폴더가 루트에 설치된 플랫 디렉터리 구조여야 합니다.

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

함수의 핸들러 코드를 포함하는 .py 파일이.zip 파일의 루트에 없는 경우 Lambda는 코드를 실행할 수 없습니다.

배포 패키지 생성(가상 환경)

1. 프로젝트 디렉터리에 가상 환경을 생성하고 활성화합니다. 이 예제에서 프로젝트 디렉터리 이름은 `my_function`입니다.

```
~$ cd my_function
~/my_function$ python3.12 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. pip를 사용하여 필요한 라이브러리를 설치합니다. 다음 예제에서는 Boto3 SDK를 설치합니다.

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. `pip show`를 사용하여 가상 환경에서 pip가 종속 항목을 설치한 위치를 찾습니다.

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

pip가 라이브러리를 설치하는 폴더의 이름은 `site-packages` 또는 `dist-packages`일 수 있습니다. 이 폴더는 `lib/python3.x` 또는 `lib64/python3.x` 디렉터리에 있을 수 있습니다(여기서 python3.x는 사용 중인 Python 버전을 나타냄).

4. 가상 환경 비활성화

```
(my_virtual_env) ~/my_function$ deactivate
```

5. pip를 사용하여 설치한 종속 항목이 포함된 디렉터리로 이동하고 루트에 설치된 종속 항목이 있는 프로젝트 디렉터리에 .zip 파일을 생성합니다. 이 예제에서 pip는 `my_virtual_env/lib/python3.12/site-packages` 디렉터리에 종속 항목을 설치했습니다.

```
~/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. 핸들러 코드가 포함된 .py 파일이 있는 프로젝트 디렉터리의 루트로 이동하고 해당 파일을 .zip 패키지의 루트에 추가합니다. 이 예제에서 함수 코드 파일의 이름은 `lambda_function.py`입니다.

```
~/my_function/my_virtual_env/lib/python3.12/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

종속 항목 검색 경로 및 런타임 포함 라이브러리

코드에서 `import` 문을 사용하면 Python 런타임은 모듈 또는 패키지를 찾을 때까지 검색 경로의 디렉터리를 검색합니다. 기본적으로 런타임에서 검색하는 첫 번째 위치는.zip 배포 패키지가 압축 해제되고 탑재되는 디렉터리입니다(`/var/task`). 배포 패키지에 런타임 포함 라이브러리 버전을 포함하는 경우 이 버전이 런타임에 포함된 버전보다 우선합니다. 배포 패키지의 종속 항목도 계층의 종속 항목보다 우선합니다.

계층에 종속 항목을 추가하면 Lambda는 이 종속 항목을 `/opt/python/lib/python3.x/site-packages`로 추출합니다. 여기서 `python3.x`는 사용 중인 런타임의 버전 또는 `/opt/python`을 나타냅니다. 검색 경로에서 이러한 디렉터리는 런타임 포함 라이브러리와 pip 설치 라이브러리(`/var/runtime` 및 `/var/lang/lib/python3.x/site-packages`)가 포함된 디렉터리보다 우선합니다. 따라서 함수 계층의 라이브러리는 런타임에 포함된 버전보다 우선합니다.

> **Note**
>
> Python 3.11 관리형 런타임 및 기본 이미지에서 AWS SDK와 해당 종속 항목은 `/var/lang/lib/python3.11/site-packages` 디렉터리에 설치됩니다.

다음 코드 조각을 추가하여 Lambda 함수에 대한 전체 검색 경로를 볼 수 있습니다.

```
import sys
```

```
search_path = sys.path
print(search_path)
```

> **Note**
>
> 배포 패키지 또는 계층의 종속 항목이 런타임 포함 라이브러리보다 우선하므로 SDK도 포함하지 않고 패키지에 urllib3과 같은 SDK 종속 항목을 포함하면 버전 불일치 문제가 발생할 수 있습니다. 자체 버전의 Boto3 종속 항목을 배포하는 경우 Boto3도 배포 패키지에 종속 항목으로 배포해야 합니다. 런타임에 해당 버전이 포함되어 있더라도 함수의 모든 종속 항목을 패키징하는 것이 좋습니다.

.zip 패키지 내의 별도 폴더에 종속 항목을 추가할 수도 있습니다. 예를 들어 `common`이라는 .zip 패키지의 폴더에 Boto3 SDK 버전을 추가할 수 있습니다. .zip 패키지를 압축 해제하고 탑재하면 `/var/task` 디렉터리 내에 이 폴더가 배치됩니다. 코드에서 .zip 배포 패키지의 폴더에 있는 종속 항목을 사용하려면 `import from` 문을 사용합니다. 예를 들어, .zip 패키지의 `common` 폴더에 있는 Boto3 버전을 사용하려면 다음 문을 사용합니다.

```
from common import boto3
```

__pycache__ 폴더 사용

함수의 배포 패키지에 `__pycache__` 폴더를 포함하지 않는 것이 좋습니다. 다른 아키텍처나 운영 체제의 빌드 시스템에서 컴파일된 Python 바이트 코드는 Lambda 실행 환경과 호환되지 않을 수 있습니다.

네이티브 라이브러리로 .zip 배포 패키지 생성

함수가 순수 Python 패키지 및 모듈만 사용하는 경우 `pip install` 명령을 사용하여 로컬 빌드 시스템에 종속 항목을 설치하고 .zip 파일을 생성할 수 있습니다. NumPy와 Pandas를 비롯한 많은 인기 Python 라이브러리는 순수 Python이 아니며 C 또는 C++로 작성된 코드를 포함합니다. 배포 패키지에 C/C++ 코드가 포함된 라이브러리를 추가할 때 패키지가 Lambda 실행 환경과 호환되도록 패키지를 올바르게 빌드해야 합니다.

Python 패키지 인덱스(PyPI)에서 사용할 수 있는 대부분의 패키지는 '휠'(.whl 파일)로 사용할 수 있습니다. .whl 파일은 특정 운영 체제와 명령 세트 아키텍처용으로 미리 컴파일된 바이너리가 포함된 빌드된 배포판을 포함하는 일종의 ZIP 파일입니다. 배포 패키지가 Lambda와 호환되도록 하려면 Linux 운영 체제용 휠과 함수의 명령 세트 아키텍처를 설치합니다.

일부 패키지는 소스 배포판으로만 사용할 수 있습니다. 이러한 패키지의 경우 C/C++ 구성 요소를 직접 컴파일하고 빌드해야 합니다.

필요한 패키지에 사용할 수 있는 배포판을 확인하려면 다음을 수행하세요.

1. Python Package Index 기본 페이지에서 패키지 이름을 검색합니다.
2. 사용할 패키지의 버전을 선택합니다.
3. Download files를 선택합니다.

빌드된 배포판 작업(휠)

Lambda와 호환되는 휠을 다운로드하려면 pip `--platform` 옵션을 사용합니다.

Lambda 함수가 x86_64 명령 세트 아키텍처를 사용하는 경우 다음 `pip install` 명령을 실행하여 `package` 디렉터리에 호환되는 휠을 설치합니다. `--python 3.x`를 사용 중인 Python 런타임 버전으로 바꿉니다.

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

함수에서 arm64 명령 세트 아키텍처를 사용하는 경우 다음 명령을 실행합니다. `--python 3.x`를 사용 중인 Python 런타임 버전으로 바꿉니다.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

소스 배포판 작업

패키지를 소스 배포판으로만 사용할 수 있는 경우 C/C++ 라이브러리를 직접 빌드해야 합니다. 패키지가 Lambda 실행 환경과 호환되도록 하려면 동일한 Amazon Linux 2 운영 체제를 사용하는 환경에서

패키지를 빌드해야 합니다. Amazon EC2 Linux 인스턴스에서 패키지를 빌드하여 이 작업을 수행할 수 있습니다.

Amazon EC2 Linux 인스턴스를 시작하고 연결하는 방법을 알아보려면 Amazon EC2 - Linux 인스턴스용 사용 설명서의 자습서: Amazon EC2 Linux 인스턴스 시작을 참조하세요.

.zip 파일을 사용하여 Python Lambda 함수 생성 및 업데이트

.zip 배포 패키지를 생성한 후에는 이를 사용하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트할 수 있습니다. Lambda 콘솔, AWS Command Line Interface 및 Lambda API를 사용하여 .zip 패키지를 배포할 수 있습니다. AWS Serverless Application Model(AWS SAM) 및 AWS CloudFormation을 사용하여 Lambda 함수를 생성하고 업데이트할 수도 있습니다.

Lambda용 .zip 배포 패키지의 최대 크기는 250MB(압축 해제됨)입니다. 이 제한은 Lambda 계층을 포함하여 업로드하는 모든 파일의 합산 크기에 적용됩니다.

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 chmod 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

콘솔을 사용하여.zip 파일로 함수 생성 및 업데이트

새 함수를 생성하려면 먼저 콘솔에서 함수를 생성한 다음.zip 아카이브를 업로드해야 합니다. 기존 함수를 업데이트하려면 함수에 대한 페이지를 연 다음 동일한 절차에 따라 업데이트된 .zip 파일을 추가합니다.

.zip 파일이 50MB 미만인 경우 로컬 컴퓨터에서 직접 파일을 업로드하여 함수를 생성하거나 업데이트할 수 있습니다. 50MB보다 큰 .zip 파일의 경우 먼저 패키지를 Amazon S3 버킷에 업로드해야 합니다. AWS Management Console을 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 Amazon S3 시작하기를 참조하세요. AWS CLI를 사용하여 파일을 업로드하려면 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

새 함수 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.
3. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름에 함수 이름을 입력합니다.
   b. 런타임에서 사용할 런타임을 선택합니다.
   c. (선택 사항) 아키텍처에서 함수에 대한 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수에 대한 .zip 배포 패키지가 선택한 명령 세트 아키텍처와 호환되는지 확인합니다.
4. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 실행 역할을 사용할 수 있습니다.
5. 함수 생성을 선택합니다. Lambda에서 선택한 런타임을 사용하여 기본 'Hello World' 함수를 생성합니다.

로컬 시스템에서 .zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서.zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. .zip 파일을 선택합니다.
5. .zip 파일을 업로드하려면 다음을 수행합니다.
   a. 업로드를 선택한 다음 파일 선택기에서.zip 파일을 선택합니다.
   b. Open을 선택합니다.
   c. Save(저장)를 선택합니다.

Amazon S3 버킷에서.zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서 새 .zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. Amazon S3 위치를 선택합니다.
5. .zip 파일의 Amazon S3 링크 URL을 붙여 넣고 저장을 선택합니다.

콘솔 코드 편집기를 사용하여.zip 파일 함수 업데이트

.zip 배포 패키지를 사용하는 일부 함수의 경우 Lambda 콘솔의 기본 제공 코드 편집기를 사용하여 함수 코드를 직접 업데이트할 수 있습니다. 이 기능을 사용하려면 함수가 다음 조건을 충족해야 합니다.

- 함수에서 해석된 언어 런타임(Python, Node.js 또는 Ruby) 중 하나를 사용해야 합니다.
- 함수의 배포 패키지가 3MB보다 작아야 합니다.

컨테이너 이미지 배포 패키지가 있는 함수의 함수 코드는 콘솔에서 직접 편집할 수 없습니다.

콘솔 코드 편집기를 사용하여 함수 코드 업데이트

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 소스 코드 파일을 선택하고 통합 코드 편집기에서 편집합니다.
4. 코드 편집을 마치면 배포를 선택하여 변경 사항을 저장하고 함수를 업데이트합니다.

AWS CLI를 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CLI를 사용하여 새 함수를 생성하거나.zip 파일로 기존 함수를 업데이트할 수 있습니다. create-function 및 update-function-code 명령을 사용하여 .zip 패키지를 배포합니다. .zip 파일이 50MB보다 작은 경우 로컬 빌드 시스템의 파일 위치에서 .zip 패키지를 업로드할 수 있습니다. 더 큰 파일의 경우 Amazon S3 버킷에서 .zip 패키지를 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> AWS CLI를 사용하여 Amazon S3 버킷에서 .zip 파일을 업로드하는 경우 버킷은 함수와 동일한 AWS 리전에 있어야 합니다.

AWS CLI에서 .zip 파일을 사용하여 새 함수를 생성하려면 다음을 지정해야 합니다.

- 함수의 이름(`--function-name`)
- 함수의 런타임(`--runtime`)
- 함수의 실행 역할(`--role`)의 Amazon 리소스 이름(ARN)
- 함수 코드에 있는 핸들러 메서드의 이름(`--handler`)

.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
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 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--code` 옵션을 사용합니다. 버전이 지정된 객체에만 `S3ObjectVersion` 파라미터를 사용해야 합니다.

```
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
```

CLI를 사용하여 기존 함수를 업데이트하려면 `--function-name` 파라미터를 사용하여 함수 이름을 지정합니다. 함수 코드를 업데이트하는 데 사용할.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--s3-bucket` 및 `--s3-key` 옵션을 사용합니다. 버전이 지정된 객체에만 `--s3-object-version` 파라미터를 사용해야 합니다.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API를 사용하여.zip 파일로 함수 생성 및 업데이트

.zip 파일 아카이브를 사용하여 함수를 생성하고 업데이트하려면 다음 API 작업을 사용합니다.

- CreateFunction
- UpdateFunctionCode

AWS SAM을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS Serverless Application Model(AWS SAM)은 AWS에서 서버리스 애플리케이션을 빌드하고 실행하는 프로세스를 간소화하는 데 도움이 되는 도구 키트입니다. YAML 또는 JSON 템플릿에서 애플리케이션의 리소스를 정의하고 AWS SAM Command Line Interface(AWS SAM CLI)를 사용하여 애플리케이션을 빌드, 패키징 및 배포합니다. AWS SAM 템플릿에서 Lambda 함수를 빌드하면 AWS SAM은 함수 코드와 사용자가 지정하는 종속 항목을 사용하여 .zip 배포 패키지 또는 컨테이너 이미지를 자동으로 생성합니다. AWS SAM을 사용하여 Lambda 함수를 빌드하고 배포하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Getting started with AWS SAM을 참조하세요.

AWS SAM을 사용하여 기존 .zip 파일 아카이브로 Lambda 함수를 생성할 수도 있습니다. AWS SAM을 사용하여 Lambda 함수를 생성하려면 Amazon S3 버킷 또는 빌드 시스템의 로컬 폴더에 .zip 파일을 저장할 수 있습니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS SAM 템플릿에서 `AWS::Serverless::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `CodeUri` - 함수 코드의 Amazon S3 URI, 로컬 폴더 경로 또는 FunctionCode 객체로 설정됨
- `Runtime` - 선택한 런타임으로 설정됨

AWS SAM을 사용하면 .zip 파일이 50MB보다 큰 경우 Amazon S3 버킷에 먼저 파일을 업로드할 필요가 없습니다. AWS SAM은 로컬 빌드 시스템의 위치에서 허용되는 최대 크기 250MB(압축 해제)까지 .zip 패키지를 업로드할 수 있습니다.

AWS SAM에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS SAM 개발자 안내서의 AWS::Serverless::Function을 참조하세요.

AWS CloudFormation을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CloudFormation을 사용하여 .zip 파일 아카이브로 Lambda 함수를 생성할 수 있습니다. .zip 파일에서 Lambda 함수를 생성하려면 먼저 Amazon S3 버킷에 파일을 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

Node.js 및 Python 런타임의 경우 AWS CloudFormation 템플릿에서 인라인 소스 코드를 제공할 수도 있습니다. 그러면 함수를 빌드할 때 AWS CloudFormation에서 코드가 포함된 .zip 파일을 생성합니다.

기존.zip 파일 사용

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `Code` - `S3Bucket` 및 `S3Key` 필드에 Amazon S3 버킷 이름과 .zip 파일 이름을 입력합니다.
- `Runtime` - 선택한 런타임으로 설정됨

인라인 코드에서.zip 파일 생성

AWS CloudFormation 템플릿에서 Python 또는 Node.js 인라인으로 작성된 단순 함수를 선언할 수 있습니다. 코드가 YAML 또는 JSON에 포함되어 있으므로 배포 패키지에 외부 종속 항목을 추가할 수 없습니다. 즉, 함수는 런타임에 포함된 AWS SDK 버전을 사용해야 합니다. 또한 특정 문자를 이스케이프해야 하는 것과 같은 템플릿의 요구 사항으로 인해 IDE의 구문 검사 및 코드 완료 기능을 사용하기가 더 어려워집니다. 이는 템플릿에 추가 테스트가 필요할 수 있음을 의미합니다. 이러한 제한 때문에 자주 변경되지 않는 매우 단순한 코드에는 인라인으로 함수를 선언하는 것이 가장 적합합니다.

Node.js 및 Python 런타임에 대한 인라인 코드에서 .zip 파일을 생성하려면 템플릿의 `AWS::Lambda::Function` 리소스에 다음 속성을 설정합니다.

- `PackageType` - `Zip`으로 설정됨
- `Code` - `ZipFile` 필드에 함수 코드 입력

- `Runtime` - 선택한 런타임으로 설정됨

AWS CloudFormation에서 생성하는 .zip 파일은 4MB를 초과할 수 없습니다. AWS CloudFormation에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function을 참조하세요.

컨테이너 이미지로 Python Lambda 함수 배포

Python Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- Python용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.

- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Python용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Python용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

주제

- AWSPython용 기본 이미지
- Python용 AWS 기본 이미지 사용
- 런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

AWSPython용 기본 이미지

AWS는 Python에 대한 다음과 같은 기본 이미지를 제공합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
3.12	Python 3.12	Amazon Linux 2023	GitHub의 Python 3.12용 Dockerfile	
3.11	Python 3.11	Amazon Linux 2	GitHub의 Python 3.11용 Dockerfile	
3.10	Python 3.10	Amazon Linux 2	GitHub의 Python 3.10용 Dockerfile	
3.9	Python 3.9	Amazon Linux 2	GitHub의 Python 3.9용 Docker 파일	
3.8	Python 3.8	Amazon Linux 2	GitHub의 Python 3.8용 Dockerfile	2024년 10월 14일

Amazon ECR 리포지토리: gallery.ecr.aws/lambda/python

Python 3.12 이상의 기본 이미지는 Amazon Linux 2023 최소 컨테이너 이미지를 기반으로 합니다. Python 3.8-3.11 기본 이미지는 Amazon Linux 2 이미지를 기반으로 합니다. AL2023 기반 이미지는 작은 배포 공간과 `glibc`와 같이 업데이트된 라이브러리 버전을 포함하여 Amazon Linux 2에 비해 여러 가지 이점을 제공합니다.

AL2023 기반 이미지는 `microdnf`(`dnf` 심볼릭 링크)를 Amazon Linux 2에서 기본 패키지 관리자인 `yum` 대신 패키지 관리자로 사용합니다. `microdnf`는 `dnf`의 독립 실행형 구현입니다. AL2023 기반 이미지에 포함된 패키지 목록의 경우 Comparing packages installed on Amazon Linux 2023 Container Images의 Minimal Container 열을 참조하세요. AL2023과 Amazon Linux 2의 차이점에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Introducing the Amazon Linux 2023 runtime for AWS Lambda를 참조하세요.

> **Note**
>
> AWS Serverless Application Model(AWS SAM)을 포함하여 AL2023 기반 이미지를 로컬에서 실행하려면 Docker 버전 20.10.10 이상을 사용해야 합니다.

기본 이미지의 종속 항목 검색 경로

코드에서 `import` 문을 사용하면 Python 런타임은 모듈 또는 패키지를 찾을 때까지 검색 경로의 디렉터리를 검색합니다. 기본적으로 런타임은 `{LAMBDA_TASK_ROOT}` 디렉터리를 먼저 검색합니다. 이미지에 런타임 포함 라이브러리의 버전을 포함하는 경우 이 버전이 런타임에 포함된 버전보다 우선합니다.

검색 경로의 다른 단계는 사용 중인 Python용 Lambda 기본 이미지의 버전에 따라 다릅니다.

- Python 3.11 이상: 런타임 포함 라이브러리와 pip 설치 라이브러리는 `/var/lang/lib/python3.11/site-packages` 디렉터리에 설치됩니다. 이 디렉터리는 검색 경로에서 `/var/runtime`보다 우선합니다. pip를 사용하여 최신 버전을 설치하여 SDK를 재정의할 수 있습니다. pip를 사용하여 런타임 포함 SDK와 해당 종속 항목이 설치하는 패키지와 호환되는지 확인할 수 있습니다.
- Python 3.8-3.10: 런타임 포함 라이브러리는 `/var/runtime` 디렉터리에 설치됩니다. Pip 설치 라이브러리는 `/var/lang/lib/python3.x/site-packages` 디렉터리에 설치됩니다. `/var/runtime` 디렉터리는 검색 경로에서 `/var/lang/lib/python3.x/site-packages`보다 우선합니다.

다음 코드 조각을 추가하여 Lambda 함수에 대한 전체 검색 경로를 볼 수 있습니다.

```
import sys

search_path = sys.path
print(search_path)
```

Python용 AWS 기본 이미지 사용

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2

- Docker(Python 3.12 이상 기본 이미지의 경우 최소 버전 20.10.10)
- Python

기본 이미지에서 이미지 생성

Python용 AWS 기본 이미지에서 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir example
cd example
```

2. `lambda_function.py`라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

   Example Python 함수

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. `requirements.txt`라는 파일을 새로 생성합니다. 이전 단계의 샘플 함수 코드를 사용하는 경우 종속 항목이 없으므로 파일을 비워 둘 수 있습니다. 그렇지 않으면 필요한 각 라이브러리를 나열합니다. 예를 들어, 함수가 AWS SDK for Python (Boto3)을 사용하는 경우 `requirements.txt`는 다음과 같아야 합니다.

   Example requirements.txt

```
boto3
```

4. 다음 구성으로 새 Dockerfile을 생성합니다.

   - `FROM` 속성을 기본 이미지의 URI로 설정합니다.
   - COPY 명령을 사용하여 함수 코드와 런타임 종속성을 Lambda 정의 환경 변수인 `{LAMBDA_TASK_ROOT}`에 복사합니다.
   - `CMD` 인수를 Lambda 함수 핸들러로 설정합니다.

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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 이 예제에서 `docker-image`는 이미지 이름이고 `test`는 태그입니다.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

2. 새 터미널 창에서 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 컨테이너 ID를 가져옵니다.

```
docker ps
```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API을 확장합니다.

pip 패키지 관리자를 사용하여 Python용 런타임 인터페이스 클라이언트를 설치합니다.

```
pip install awslambdaric
```

GitHub에서 Python 런타임 인터페이스 클라이언트를 다운로드할 수도 있습니다.

다음 예제에서는 비 AWS 기본 이미지를 사용하여 Python용 컨테이너 이미지를 빌드하는 방법을 보여 줍니다. 예제 Dockerfile에서는 공식 Python 기본 이미지를 사용합니다. Dockerfile에는 Python용 런타임 인터페이스 클라이언트가 포함되어 있습니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker
- Python

대체 기본 이미지에서 이미지 생성

비 AWS 기본 이미지에서 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

   ```
   mkdir example
   cd example
   ```

2. `lambda_function.py`라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

   Example Python 함수

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. `requirements.txt`라는 파일을 새로 생성합니다. 이전 단계의 샘플 함수 코드를 사용하는 경우 종속 항목이 없으므로 파일을 비워 둘 수 있습니다. 그렇지 않으면 필요한 각 라이브러리를 나열합니다. 예를 들어, 함수가 AWS SDK for Python (Boto3)을 사용하는 경우 `requirements.txt`는 다음과 같아야 합니다.

Example requirements.txt

```
boto3
```

4. 새 Dockerfile을 생성합니다. 다음 Dockerfile은 AWS 기본 이미지 대신 공식 Python 기본 이미지를 사용합니다. Dockerfile에는 이미지가 Lambda와 호환되도록 하는 런타임 인터페이스 클라이언트가 포함되어 있습니다. 다음 예제 Dockerfile은 다단계 빌드를 사용합니다.

   - FROM 속성을 기본 이미지로 설정합니다.
   - Docker 컨테이너가 시작될 때 실행할 모듈로 ENTRYPOINT를 설정합니다. 이 경우 모듈은 런타임 인터페이스 클라이언트입니다.
   - CMD를 Lambda 함수 핸들러로 설정합니다.

   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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. 에뮬레이터를 이미지에 빌드하거나 다음 절차를 사용하여 로컬 시스템에 설치할 수 있습니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 설치 및 실행

1. 프로젝트 디렉터리에서 다음 명령을 실행하여 GitHub에서 런타임 인터페이스 에뮬레이터(x86-64 아키텍처)를 다운로드하고 로컬 시스템에 설치합니다.

   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 에뮬레이터를 설치하려면 이전 명령의 GitHub 리포지토리 URL을 다음과 같이 바꿉니다.

```
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 에뮬레이터를 설치하려면 `$downloadLink`을(를) 다음과 같이 바꿉니다.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `/usr/local/bin/python -m awslambdaric lambda_function.handler`는 Docker 파일의 `CMD` 다음에 오는 `ENTRYPOINT`입니다.

   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
```

이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

3. 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 컨테이너 ID를 가져옵니다.

```
docker ps
```

5. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

   예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

Alpine 기본 이미지에서 Python 이미지를 생성하는 방법에 대한 예는 AWS 블로그에서 Container image support for Lambda를 참조하세요.

Python Lambda 함수를 위한 계층 작업

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다. 계층을 생성하려면 세 가지 일반적인 단계를 거칩니다.

1. 계층 콘텐츠를 패키징합니다. 즉, 함수에 사용하려는 종속성이 포함된 .zip 파일 아카이브를 생성합니다.
2. Lambda에서 계층을 생성합니다.
3. 계층을 함수에 추가합니다.

이 주제에는 외부 라이브러리 종속성이 있는 Python Lambda 계층을 올바르게 패키징하고 생성하는 방법에 대한 단계와 지침이 포함되어 있습니다.

주제

- 필수 조건
- Amazon Linux와의 Python 계층 호환성
- Python 런타임의 계층 경로
- 계층 콘텐츠의 패키징
- 계층의 생성
- 함수에 계층 추가
- manylinux 휠 배포를 사용하여 작업

필수 조건

이 섹션의 단계를 수행하려면 다음이 필요합니다.

- Python 3.11 및 pip 패키지 설치 관리자
- AWS Command Line Interface(AWS CLI) 버전 2

이 주제에서는 awsdocs GitHub 리포지토리에 있는 `layer-python` 샘플 애플리케이션을 참조합니다. 이 애플리케이션은 종속성을 다운로드하고 계층을 생성하는 스크립트를 포함합니다. 애플리케이션은 계층의 종속성을 사용하는 해당 함수도 포함합니다. 계층을 생성한 후 해당 함수를 배포하고 호출하여 모든 것이 제대로 작동하는지 확인할 수 있습니다. 함수에 Python 3.11 런타임을 사용하므로 계층도 Python 3.11과 호환되어야 합니다.

layer-python 샘플 애플리케이션에는 두 가지 예제가 있습니다.

- 첫 번째 예제는 requests 라이브러리를 Lambda 계층으로 패키징하는 것입니다. layer/ 디렉터리는 계층을 생성하는 스크립트를 포함합니다. function/ 디렉터리는 계층이 작동하는지 테스트하는 데 도움이 되는 샘플 함수를 포함합니다. 이 자습서의 대부분은 이 계층을 생성하고 패키징하는 방법을 안내합니다.
- 두 번째 예제는 numpy 라이브러리를 Lambda 계층으로 패키징하는 것입니다. layer-numpy/ 디렉터리는 계층을 생성하는 스크립트를 포함합니다. function-numpy/ 디렉터리는 계층이 작동하는지 테스트하는 데 도움이 되는 샘플 함수를 포함합니다. 이 계층을 생성하고 패키징하는 방법에 대한 예제는 the section called “manylinux 휠 배포를 사용하여 작업” 섹션을 참조하세요.

Amazon Linux와의 Python 계층 호환성

계층을 생성하는 첫 번째 단계는 모든 계층 콘텐츠를.zip 파일 아카이브로 번들링하는 것입니다. Lambda 함수는 Amazon Linux에서 실행되기 때문에 계층 콘텐츠는 Linux 환경에서 컴파일하고 빌드할 수 있어야 합니다.

Python에서는 대부분의 패키지를 소스 배포 외에 휠(.whl 파일)로도 사용할 수 있습니다. 각 휠은 Python 버전, 운영 체제, 머신 명령어 세트의 특정 조합을 지원하는 구축된 배포판의 일종입니다.

휠은 계층이 Amazon Linux와 호환되는지 확인하는 데 유용합니다. 종속 항목을 다운로드할 때 가능하면 유니버설 휠을 다운로드하세요. (기본적으로 pip은 유니버설 휠이 있는 경우 이를 설치합니다.) 유니버설 휠에는 플랫폼 태그로 any가 포함되어 있으며 이는 Amazon Linux를 포함한 모든 플랫폼과 호환됨을 나타냅니다.

다음 예제에서는 requests 라이브러리를 Lambda 계층으로 패키징합니다. requests 라이브러리는 유니버설 휠로 사용할 수 있는 패키지의 예입니다.

모든 Python 패키지가 유니버설 휠로 배포되는 것은 아닙니다. 예를 들어 numpy에는 각각 다른 플랫폼 세트를 지원하는 여러 휠 배포가 있습니다. 이러한 패키지의 경우 manylinux 배포판을 다운로드하여 Amazon Linux와의 호환성을 확인하세요. 이러한 계층을 패키징하는 방법에 대한 자세한 지침은 the section called “manylinux 휠 배포를 사용하여 작업” 섹션을 참조하세요.

드물지만 Python 패키지를 휠로 사용할 수 없는 경우도 있습니다. 소스 배포(sdist)만 존재하는 경우 Amazon Linux 2023 기본 컨테이너 이미지를 기반으로 Docker 환경에 종속성을 설치하고 패키징하는 것이 좋습니다. C/C++와 같은 다른 언어로 작성된 사용자 지정 라이브러리를 포함하려는 경우에도 이 접근 방식을 권장합니다. 이 접근 방식은 Docker의 Lambda 실행 환경을 모방하며 Python이 아닌 패키지 종속성이 Amazon Linux와 호환되도록 보장합니다.

Python 런타임의 계층 경로

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 /opt 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 PATH 변수에는 /opt 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. PATH 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

- python
- python/lib/python3.*x*/site-packages

예를 들어 이 자습서에서 생성하는 결과 레이어 .zip 파일의 디렉터리 구조는 다음과 같습니다.

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

requests 라이브러리는 python/lib/python3.11/site-packages 디렉터리에 올바르게 위치합니다. 이렇게 하면 함수 호출 중에 Lambda가 라이브러리를 찾을 수 있습니다.

계층 콘텐츠의 패키징

이 예에서는 Python requests 라이브러리를 계층 .zip 파일로 패키징합니다. 다음 단계를 완료하여 계층 콘텐츠를 설치하고 패키징합니다.

계층 콘텐츠를 설치하고 패키징하려면 다음을 수행합니다.

1. sample-apps/layer-python 디렉터리에 필요한 샘플 코드가 포함된 aws-lambda-developer-guide GitHub 리포지토리를 복제합니다.

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. layer-python 샘플 앱의 layer 디렉터리로 이동합니다. 이 디렉터리에는 계층을 올바르게 생성하고 패키징하는 데 사용하는 스크립트가 포함되어 있습니다.

```
cd aws-lambda-developer-guide/sample-apps/layer-python/layer
```

3. requirements.txt 파일을 검사합니다. 이 파일은 계층, 즉 requests 라이브러리에 포함하려는 종속성을 정의합니다. 이 파일을 업데이트하여 자체 계층에 포함하려는 종속성을 포함할 수 있습니다.

   Example requirements.txt

   ```
   requests==2.31.0
   ```

4. 두 스크립트를 모두 실행할 수 있는 권한이 있는지 확인하세요.

   ```
   chmod 744 1-install.sh && chmod 744 2-package.sh
   ```

5. 다음 명령을 사용하여 1-install.sh 스크립트를 실행하세요.

   ```
   ./1-install.sh
   ```

   이 스크립트는 venv를 사용하여 create_layer라는 이름의 Python 가상 환경을 생성합니다. 그런 다음 create_layer/lib/python3.11/site-packages 디렉터리에 필요한 모든 종속성을 설치합니다.

   Example 1-install.sh

   ```
   python3.11 -m venv create_layer
   source create_layer/bin/activate
   pip install -r requirements.txt
   ```

6. 다음 명령을 사용하여 2-package.sh 스크립트를 실행하세요.

   ```
   ./2-package.sh
   ```

   이 스크립트는 create_layer/lib 디렉터리의 내용을 python이라는 새 디렉터리에 복사합니다. 그런 다음 python 디렉터리의 내용을 layer_content.zip라는 파일로 압축합니다. 이는 계층의 .zip 파일입니다. the section called “Python 런타임의 계층 경로” 섹션에 표시된 것처럼 파일의 압축을 풀고 올바른 파일 구조가 포함되어 있는지 확인할 수 있습니다.

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

계층의 생성

이 섹션에서는 이전 섹션에서 생성한 `layer_content.zip` 파일을 가져와 Lambda 계층으로 업로드합니다. AWS Command Line Interface(AWS CLI)를 통해 AWS Management Console 또는 Lambda API를 사용하여 계층을 업로드할 수 있습니다. 계층 .zip 파일을 업로드할 때 다음 PublishLayerVersion AWS CLI 명령에서 `python3.11`을 호환 런타임으로, `arm64`를 호환 아키텍처로 지정합니다.

```
aws lambda publish-layer-version --layer-name python-requests-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "arm64"
```

응답에서 `arn:aws:lambda:us-east-1:123456789012:layer:python-requests-layer:1`처럼 보이는 `LayerVersionArn`에 유의하세요. 이 자습서의 다음 단계인 함수에 계층을 추가할 때 이 Amazon 리소스 이름(ARN)이 필요합니다.

함수에 계층 추가

이 섹션에서는 함수 코드에 `requests` 라이브러리를 사용하는 샘플 Lambda 함수를 배포한 다음 계층을 연결합니다. 함수를 배포하려면 the section called “실행 역할(함수가 다른 리소스에 액세스할 수 있는 권한)”이 필요합니다. 기존 실행 역할이 없으면 접을 수 있는 섹션의 단계를 따르세요. 그렇지 않은 경우 다음 섹션으로 건너뛰어 기능을 배포하세요.

(선택 사항) 실행 역할 생성

실행 역할을 만들려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 다음 속성을 사용하여 역할을 만듭니다.

   - 신뢰할 수 있는 엔터티 – Lambda.

- 권한 – AWSLambdaBasicExecutionRole.
- 역할 이름 – **lambda-role**.

AWSLambdaBasicExecutionRole 정책은 함수가 CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다.

Lambda 함수를 배포하려면 다음을 수행합니다.

1. function/ 디렉터리로 이동합니다. 현재 layer/ 디렉터리에 있는 경우 다음 명령을 실행하세요.

```
cd ../function
```

2. 함수 코드를 검토합니다. 이 함수는 requests 라이브러리를 가져오고 간단한 HTTP GET 요청을 한 다음 상태 코드와 본문을 반환합니다.

```
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. 다음 명령을 사용하여 .zip 파일 배포 패키지를 생성합니다.

```
zip my_deployment_package.zip lambda_function.py
```

4. 함수를 배포합니다. 다음 AWS CLI 명령에서 --role 파라미터를 실행 역할 ARN으로 바꿉니다.

```
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
```

(선택 사항) 계층을 연결하지 않고 함수를 호출합니다.

이 시점에서 계층을 연결하기 전에 함수를 호출해 볼 수 있습니다(선택 사항). 이 방법을 시도하면 함수가 `requests` 패키지를 참조할 수 없으므로 가져오기 오류가 발생합니다. 함수를 호출하려면 다음 AWS CLI 명령을 사용합니다.

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

특정 오류를 보려면 출력 `response.json` 파일을 엽니다. 다음 오류 메시지와 함께 `ImportModuleError`가 표시됩니다.

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'requests'"
```

그런 다음 계층을 함수에 연결합니다. 다음 AWS CLI 명령에서 `--layers` 파라미터를 이전에 기록해 둔 계층 버전 ARN으로 바꿉니다.

```
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"
```

마지막으로 다음 AWS CLI 명령을 사용하여 함수를 호출합니다.

```
aws lambda invoke --function-name python_function_with_layer \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
```

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

출력 `response.json` 파일은 응답에 대한 세부 정보를 포함합니다.

(선택 사항) 리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 계층을 삭제하려면 다음을 수행합니다.

1. Lambda 콘솔의 계층 페이지를 엽니다.
2. 생성한 계층을 선택합니다.
3. 삭제를 선택한 다음 삭제를 다시 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

manylinux 휠 배포를 사용하여 작업

종속성으로 포함하려는 패키지에 유니버설 휠이 없는 경우가 있습니다(특히 플랫폼 태그에 any가 없는 경우). 이러한 경우에는 대신 manylinux를 지원하는 휠을 다운로드하세요. 이렇게 하면 계층 라이브러리가 Amazon Linux와 호환됩니다.

numpy는 유니버설 휠이 없는 패키지 중 하나입니다. 계층에 numpy 패키지를 포함하려면 다음 예제 단계를 완료하여 계층을 올바르게 설치하고 패키징할 수 있습니다.

계층 콘텐츠를 설치하고 패키징하려면 다음을 수행합니다.

1. `sample-apps/layer-python` 디렉터리에 필요한 샘플 코드가 포함된 aws-lambda-developer-guide GitHub 리포지토리를 복제합니다.

```
git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
```

2. layer-python 샘플 앱의 layer-numpy 디렉터리로 이동합니다. 이 디렉터리에는 계층을 올바르게 생성하고 패키징하는 데 사용하는 스크립트가 포함되어 있습니다.

```
cd aws-lambda-developer-guide/sample-apps/layer-python/layer-numpy
```

3. requirements.txt 파일을 검사합니다. 이 파일은 계층, 즉 numpy 라이브러리에 포함하려는 종속성을 정의합니다. 여기에서는 Python 3.11, Amazon Linux, x86_64 명령어 세트와 호환되는 manylinux 휠 배포의 URL을 지정합니다.

   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. 두 스크립트를 모두 실행할 수 있는 권한이 있는지 확인하세요.

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 다음 명령을 사용하여 1-install.sh 스크립트를 실행하세요.

```
./1-install.sh
```

   이 스크립트는 venv를 사용하여 create_layer라는 이름의 Python 가상 환경을 생성합니다. 그런 다음 create_layer/lib/python3.11/site-packages 디렉터리에 필요한 모든 종속성을 설치합니다. 이 경우에는 --platform 태그를 manylinux2014_x86_64로 지정해야 하므로 pip 명령이 다릅니다. 이렇게 하면 로컬 컴퓨터가 macOS 또는 Windows를 사용하는 경우에도 pip에 올바른 manylinux 휠을 설치하도록 지시합니다.

   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. 다음 명령을 사용하여 2-package.sh 스크립트를 실행하세요.

```
./2-package.sh
```

이 스크립트는 `create_layer/lib` 디렉터리의 내용을 `python`이라는 새 디렉터리에 복사합니다. 그런 다음 `python` 디렉터리의 내용을 `layer_content.zip`라는 파일로 압축합니다. 이는 계층의 .zip 파일입니다. the section called “Python 런타임의 계층 경로” 섹션에 표시된 것처럼 파일의 압축을 풀고 올바른 파일 구조가 포함되어 있는지 확인할 수 있습니다.

Example 2-package.sh

```
mkdir python
cp -r create_layer/lib python/
zip -r layer_content.zip python
```

이 계층을 Lambda에 업로드하려면 다음 PublishLayerVersion AWS CLI 명령을 사용합니다.

```
aws lambda publish-layer-version --layer-name python-numpy-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes python3.11 \
    --compatible-architectures "x86_64"
```

응답에서 `arn:aws:lambda:us-east-1:123456789012:layer:python-numpy-layer:1`처럼 보이는 `LayerVersionArn`에 유의하세요. 계층이 예상대로 작동하는지 확인하려면 `function-numpy` 디렉터리에 Lambda 함수를 배포합니다.

Lambda 함수를 배포하려면 다음을 수행합니다.

1. `function-numpy/` 디렉터리로 이동합니다. 현재 `layer-numpy/` 디렉터리에 있는 경우 다음 명령을 실행하세요.

   ```
   cd ../function-numpy
   ```

2. 함수 코드를 검토합니다. 이 함수는 numpy 라이브러리를 가져와서 간단한 numpy 배열을 생성한 다음 더미 상태 코드와 본문을 반환합니다.

   ```
   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. 다음 명령을 사용하여 .zip 파일 배포 패키지를 생성합니다.

```
zip my_deployment_package.zip lambda_function.py
```

4. 함수를 배포합니다. 다음 AWS CLI 명령에서 `--role` 파라미터를 실행 역할 ARN으로 바꿉니다.

```
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
```

(선택 사항) 계층을 연결하지 않고 함수를 호출합니다.

계층을 연결하기 전에 함수를 호출해 볼 수 있습니다(선택 사항). 이 방법을 시도하면 함수가 numpy 패키지를 참조할 수 없으므로 가져오기 오류가 발생합니다. 함수를 호출하려면 다음 AWS CLI 명령을 사용합니다.

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

특정 오류를 보려면 출력 `response.json` 파일을 엽니다. 다음 오류 메시지와 함께 `ImportModuleError`가 표시됩니다.

```
"errorMessage": "Unable to import module 'lambda_function': No module named 'numpy'"
```

그런 다음 계층을 함수에 연결합니다. 다음 AWS CLI 명령에서 `--layers` 파라미터를 계층 버전 ARN으로 바꿉니다.

```
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"
```

마지막으로 다음 AWS CLI 명령을 사용하여 함수를 호출합니다.

```
aws lambda invoke --function-name python_function_with_numpy \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

함수 로그를 검사하여 코드가 numpy 배열을 표준 출력으로 출력하는지 확인할 수 있습니다.

AWS Lambda 컨텍스트 객체(Python)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러.에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다. 컨텍스트 객체가 함수 핸들러에 전달되는 방법에 대한 자세한 내용은 Python에서 Lambda 함수 핸들러 정의 섹션을 참조하십시오.

컨텍스트 메서드

- `get_remaining_time_in_millis` – 실행 시간이 초과되기까지 남은 시간(밀리초)을 반환합니다.

컨텍스트 속성

- `function_name` – Lambda 함수의 이름입니다.
- `function_version` – 함수의 버전입니다.
- `invoked_function_arn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `memory_limit_in_mb` – 함수에 할당된 메모리의 양입니다.
- `aws_request_id` – 호출 요청의 식별자입니다.
- `log_group_name` – 함수에 대한 로그 그룹입니다.
- `log_stream_name` – 함수 인스턴스에 대한 로그 스트림입니다.
- `identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
  - `cognito_identity_id`— 인증된 Amazon Cognito ID입니다.
  - `cognito_identity_pool_id` – 호출에 대한 권한을 부여한 Amazon Cognito ID 풀입니다.
- `client_context` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `custom` – 모바일 클라이언트 애플리케이션에서 설정된 사용자 지정 값의 `dict`입니다.
  - `env` – AWS SDK가 제공하는 환경 정보의 `dict`입니다.

다음 예제는 컨텍스트 정보를 기록하는 핸들러 함수를 보여줍니다.

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())
```

위에 열거한 옵션들 외에도 AWS용 AWS Lambda에서 Python 코드 계측 X-Ray SDK를 사용하면 중요한 코드 경로를 식별하고 그 성능을 추적하며 분석용 데이터를 수집할 수도 있습니다.

AWS Lambda 함수 로깅(Python)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그 항목을 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 간접 호출에 대한 세부 정보와 함수 코드의 기타 출력을 로그 스트림으로 전송합니다. CloudWatch Logs에 대한 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 섹션을 참조하세요.

함수 코드에서 로그를 출력하려면 기본 제공 `logging` 모듈을 사용합니다. 더 자세한 항목을 보기 위해 `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용할 수 있습니다.

로그에 인쇄

기본 출력을 로그로 보내려면 함수에서 `print` 메서드를 사용합니다. 다음 예제에서는 CloudWatch Logs 로그 그룹 및 스트림과 이벤트 객체의 값을 로깅합니다.

참고로 함수가 Python `print` 문을 사용하여 로그를 출력하는 경우 Lambda는 일반 텍스트 형식으로만 로그 출력을 CloudWatch Logs에 보낼 수 있습니다. 구조화된 JSON으로 로그를 캡처하려면 지원되는 로깅 라이브러리를 사용해야 합니다. 자세한 정보는 the section called “Python에서 Lambda 고급 로깅 제어 사용”을 참조하세요.

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 로그 출력

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 줄을 로깅합니다. REPORT 행에는 다음 데이터가 포함됩니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

로깅 라이브러리 사용

더 자세한 로그를 보려면 표준 라이브러리의 logging 모듈을 사용하거나 `stdout` 또는 `stderr`에 쓰는 타사 로깅 라이브러리를 사용하세요.

지원되는 Python 런타임의 경우 표준 `logging` 모듈을 사용하여 생성한 로그를 일반 텍스트로 캡처할지 JSON으로 캡처할지 선택할 수 있습니다. 자세한 내용은 the section called “Python에서 Lambda 고급 로깅 제어 사용”을 참조하십시오.

현재 모든 Python 런타임의 기본 로그 형식은 일반 텍스트입니다. 다음 예제는 표준 `logging` 모듈을 사용하여 생성된 로그 출력이 CloudWatch Logs에서 일반 텍스트로 캡처되는 방법을 보여줍니다.

```
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`의 출력에는 로그 레벨, 타임스탬프와 요청 ID가 포함됩니다.

```
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**
>
> 함수의 로그 형식이 일반 텍스트로 설정된 경우 Python 런타임의 기본 로그 수준 설정은 경고입니다. 즉, Lambda는 경고 수준 이하의 로그 출력만 CloudWatch Logs로 전송합니다. 기본 로그 수준을 변경하려면 이 예제 코드에 표시된 대로 Python `logging` `setLevel()` 메서드를 사용합니다. 함수의 로그 형식을 JSON으로 설정하는 경우 코드에서 로그 수준을 설정하는 대신 Lambda 고급 로깅 제어를 사용하여 함수의 로그 수준을 구성하는 것이 좋습니다. 자세한 내용은 the section called “Python에서 로그 수준 필터링 사용” 단원을 참조하십시오.

Python에서 Lambda 고급 로깅 제어 사용

함수의 로그를 캡처, 처리 및 사용하는 방법을 더 잘 제어할 수 있도록 지원되는 Lambda Python 런타임에 대한 다음의 로깅 옵션을 구성할 수 있습니다.

- 로그 형식 - 함수 로그의 경우 일반 텍스트와 구조화된 JSON 형식 중에서 선택
- 로그 수준 - JSON 형식의 로그의 경우, Lambda가 Amazon CloudWatch로 전송하는 로그의 세부 수준(ERROR, DEBUG 또는 INFO 등)을 선택
- 로그 그룹 - 함수가 로그를 보내는 CloudWatch 로그 그룹을 선택

이러한 로깅 옵션에 대한 자세한 내용과 이를 사용하도록 함수를 구성하는 방법에 대한 지침은 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”을 참조하세요.

Python Lambda 함수에서 로그 형식 및 로그 수준 옵션을 사용하는 방법에 대해 자세히 알아보려면 다음 섹션의 지침을 참조하세요.

Python에서 구조화된 JSON 로그 사용

함수의 로그 형식으로 JSON을 선택하면 Lambda는 Python 표준 로깅 라이브러리의 로그 출력을 구조화된 JSON으로 CloudWatch에 전송합니다. 각 JSON 로그 객체에는 다음 키가 있는 4개의 키 값 페어가 포함되어 있습니다.

- `"timestamp"` - 로그 메시지가 생성된 시간
- `"level"` - 메시지에 할당된 로그 수준
- `"message"` - 로그 메시지의 내용
- `"requestId"` - 함수 간접 호출의 고유한 요청 ID

Python `logging` 라이브러리는 이 JSON 객체에 추가 키 값 페어(예: `"logger"`)를 추가할 수도 있습니다.

다음 섹션의 예는 함수의 로그 형식을 JSON으로 구성할 때 Python `logging` 라이브러리를 사용하여 생성된 로그 출력이 CloudWatch Logs에 캡처되는 방법을 보여줍니다.

참고로 `print` 메서드를 사용하여 the section called “로그에 인쇄”에 설명된 대로 기본 로그 출력을 생성하는 경우 함수의 로깅 형식을 JSON으로 구성하더라도 Lambda는 이러한 출력을 일반 텍스트로 캡처합니다.

Python 로깅 라이브러리를 사용하는 표준 JSON 로그 출력

다음 섹션의 코드 조각 및 로그 출력 예는 함수의 로그 형식을 JSON으로 구성할 때 Python `logging` 라이브러리를 사용하여 생성된 표준 로그 출력이 CloudWatch Logs에 캡처되는 방법을 보여줍니다.

Example Python 로깅 코드

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 로그 레코드

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

JSON에 추가 파라미터 로깅

함수의 로그 형식이 JSON으로 설정된 경우 `extra` 키워드를 사용하여 Python 사전을 로그 출력에 전달함으로써 표준 Python `logging` 라이브러리로 추가 파라미터를 로깅할 수도 있습니다.

Example Python 로깅 코드

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON 로그 레코드

```
{
  "timestamp": "2023-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
```

```
  "b": [
    3
  ]
}
```

JSON에서 예외 로깅

다음 코드 조각은 로그 형식을 JSON으로 구성할 때 함수의 로그 출력에서 Python 예외가 캡처되는 방법을 보여줍니다. 참고로 `logging.exception`을 사용하여 생성된 로그 출력에는 로그 수준 ERROR가 할당된다는 점에 유의하세요.

Example Python 로깅 코드

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON 로그 레코드

```
{
  "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"
}
```

다른 로깅 도구를 사용한 JSON 구조화된 로그

코드에서 이미 AWS Lambda용 Powertools 같은 다른 로깅 라이브러리를 사용하여 JSON 구조화된 로그를 생성하는 경우에는 변경할 필요가 없습니다. AWS Lambda는 이미 JSON으로 인코딩된 로그는

이중 인코딩하지 않습니다. JSON 로그 형식을 사용하도록 함수를 구성하더라도 로깅 출력은 사용자가 정의한 JSON 구조로 CloudWatch에 표시됩니다.

다음 예는 AWS Lambda 패키지용 Powertools를 사용하여 생성된 로그 출력이 CloudWatch Logs에 캡처되는 방법을 보여줍니다. 이 로그 출력의 형식은 함수의 로깅 구성이 JSON으로 설정되든 TEXT로 설정되었든 동일합니다. AWS Lambda용 Powertools 사용에 대한 자세한 내용은 the section called “구조화된 로깅에 Powertools for AWS Lambda(Python) 및 AWS SAM 사용” 및 the section called “구조화된 로깅에 Powertools for AWS Lambda(Python) 및 AWS CDK 사용”을 참조하세요.

Example Python 로깅 코드 조각(AWS Lambda용 Powertools 사용)

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 로그 레코드(AWS Lambda용 Powertools 사용)

```
{
    "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"
}
```

Python에서 로그 수준 필터링 사용

로그 수준 필터링을 구성하면 특정 로깅 수준 이하의 로그만 CloudWatch Logs로 전송하도록 선택할 수 있습니다. 함수의 로그 수준 필터링을 구성하는 방법을 알아보려면 the section called “로그 수준 필터링”을 참조하세요.

AWS Lambda에서 애플리케이션 로그를 로그 수준에 따라 필터링하려면 함수에서 JSON 형식의 로그를 사용해야 합니다. 다음 두 가지 방법으로 이 작업을 달성할 수 있습니다.

- 표준 Python `logging` 라이브러리를 사용하여 로그 출력을 생성하고 JSON 로그 형식을 사용하도록 함수를 구성합니다. AWS Lambda은 그런 다음 the section called “Python에서 구조화된 JSON 로그 사용”에서 설명하는 JSON 객체의 “레벨” 키 값 쌍을 사용하여 로그 출력을 필터링합니다. 함수

의 로그 형식을 구성하는 방법을 알아보려면 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”를 참조하세요.

- 다른 로깅 라이브러리 또는 메서드를 사용하여 로그 출력 수준을 정의하는 “레벨” 키 값 쌍이 포함된 JSON 구조화된 로그를 코드에 만들 수 있습니다. 예를 들어 AWS Lambda용 Powertools를 사용하여 코드로부터 JSON 구조화된 로그 출력을 생성할 수 있습니다.

  또한 인쇄 문을 사용하여 로그 수준 식별자를 포함하는 JSON 객체를 출력할 수도 있습니다. 다음 인쇄 문은 로그 수준이 INFO로 설정된 JSON 형식의 출력을 생성합니다. AWS Lambda는 함수의 로깅 수준이 INFO, DEBUG, 또는 TRACE로 설정된 경우 JSON 객체를 CloudWatch Logs로 전송합니다.

```
print('{"msg":"My log message", "level":"info"}')
```

Lambda가 함수 로그를 필터링하려면 JSON 로그 출력에 `"timestamp"` 키 값 쌍도 포함해야 합니다. 시간은 유효한 RFC 3339 타임스탬프 형식으로 지정해야 합니다. 유효한 타임스탬프를 제공하지 않으면 Lambda는 로그에 레벨 INFO를 할당하고 타임스탬프를 추가합니다.

Lambda 콘솔에서 로그 보기

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔에서 로그 보기

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS CLI를 사용하여 로그 보기

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - aws configure을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 `base64` 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 `my-function`에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```

로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

도구 및 라이브러리

Powertools for AWS Lambda(Python)는 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다. Logger 유틸리티는 JSON으로 구조화된 출력과 함께 모든 함수의 함수 컨텍스트에 대한 추가 정보를 포함하는 Lambda 최적화 로거를 제공합니다. 이 유틸리티를 사용하여 다음을 수행합니다.

- Lambda 컨텍스트, 콜드 스타트 및 구조 로깅 출력에서 JSON으로 주요 필드 캡처
- 지시 시 Lambda 호출 이벤트 로깅(기본적으로 비활성화됨)
- 로그 샘플링을 통해 호출 비율에 대해서만 모든 로그 인쇄(기본적으로 비활성화됨)
- 언제든지 구조화된 로그에 추가 키 추가
- 사용자 지정 로그 포맷터(Bring Your Own Formatter)를 사용하여 조직의 로깅 RFC와 호환되는 구조로 로그 출력

구조화된 로깅에 Powertools for AWS Lambda(Python) 및 AWS SAM 사용

다음 단계를 따라 AWS SAM을 사용하는 통합 Powertools for Python 모듈을 사용하여 샘플 Hello World Python 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Python 3.9
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World Python 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name sam-app
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

로그 보존 관리

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 경과 후 CloudWatch가 로그를 자동으로 삭제하는 보존 기간을 구성하세요. 로그 보존을 설정하려면 AWS SAM 템플릿에 다음을 추가합니다.

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

구조화된 로깅에 Powertools for AWS Lambda(Python) 및 AWS CDK 사용

다음 단계에 따라 AWS CDK를 사용하여 통합 Powertools for AWS Lambda(Python) 모듈이 포함된 샘플 Hello World Python 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 함수가 hello world 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Python 3.9
- AWS CLI 버전 2
- AWS CDK 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS CDK 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

```
mkdir hello-world
cd hello-world
```

2. 앱을 초기화합니다.

```
cdk init app --language python
```

3. Python 종속 구성 요소를 설치합니다.

```
pip install -r requirements.txt
```

4. 루트 폴더 아래에 lambda_function 디렉터리를 생성합니다.

```
mkdir lambda_function
cd lambda_function
```

5. 파일 app.py를 생성하고 파일에 다음 코드를 추가합니다. Lambda 함수에 대한 코드입니다.

```
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 디렉터리를 엽니다. hello_world_stack.py라는 파일이 있어야 합니다.

```
cd ..
cd hello_world
```

7. hello_world_stack.py를 열고 파일에 다음 코드를 추가합니다. 여기에는 Lambda 함수를 생성하고, Powertools에 대한 환경 변수를 구성하고, 로그 보존을 1주일로 설정하는 Lambda Constructor와 REST API를 생성하는 ApiGatewayv1 Constructor가 포함됩니다.

```
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. 애플리케이션 배포

```
cd ..
cdk deploy
```

9. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. API 엔드포인트 호출:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

11. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name HelloWorldStack
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
cdk destroy
```

Python에서 AWS Lambda 함수 테스트

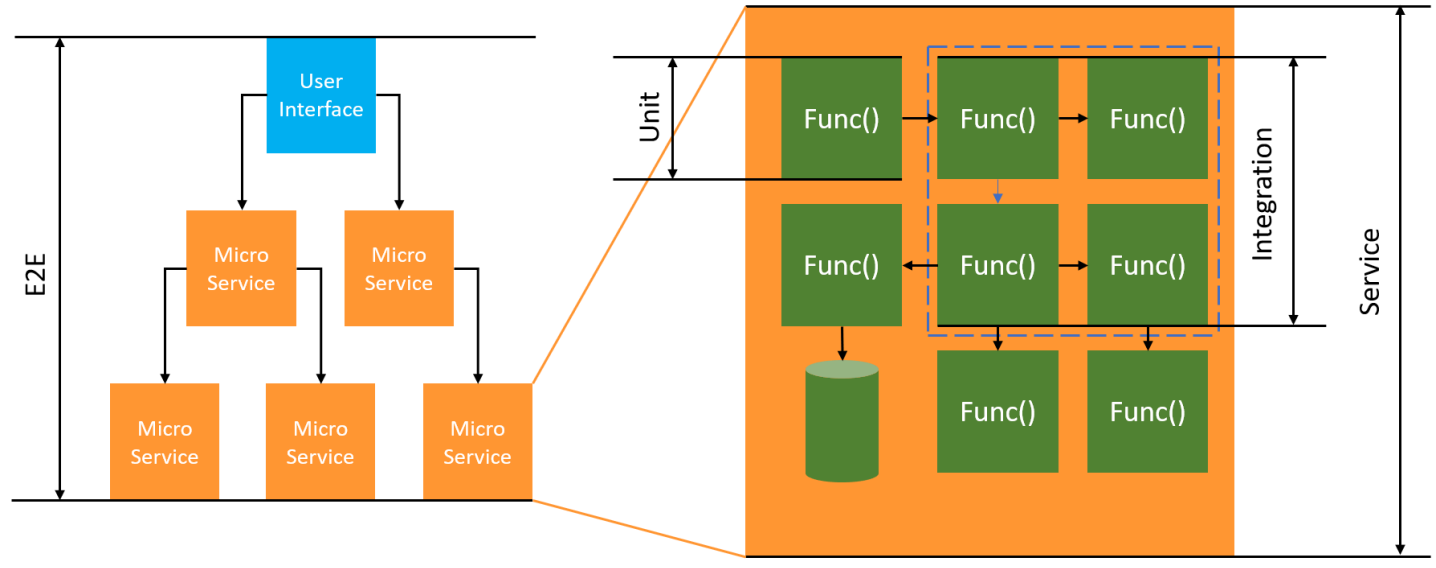
> **Note**
>
> 서버리스 솔루션 테스트를 위한 기술 및 모범 사례에 대한 전체 소개는 함수 테스트 장을 참조하세요.

서버리스 함수 테스트는 기존 테스트 유형과 기법을 사용하지만, 서버리스 애플리케이션을 전체적으로 테스트하는 것도 고려해야 합니다. 클라우드 기반 테스트는 함수와 서버리스 애플리케이션 모두의 품질을 가장 정확하게 측정합니다.

서버리스 애플리케이션 아키텍처에는 API 호출을 통해 중요한 애플리케이션 기능을 제공하는 관리형 서비스가 포함됩니다. 따라서 개발 주기에는 함수와 서비스가 상호 작용할 때 기능을 확인하는 자동화된 테스트가 포함되어야 합니다.

클라우드 기반 테스트를 생성하지 않으면 로컬 환경과 배포된 환경 간의 차이로 인해 문제가 발생할 수 있습니다. 지속적 통합 프로세스는 QA, 스테이징 또는 프로덕션과 같은 다음 배포 환경으로 코드를 승격하기 전에 클라우드에서 프로비저닝되는 리소스 제품군을 대상으로 테스트를 실행해야 합니다.

이 짧은 안내서를 계속 읽고 서버리스 애플리케이션의 테스트 전략에 대해 알아보거나 Serverless Test Samples 리포지토리를 방문하여 선택한 언어 및 런타임과 관련된 실제 예제를 자세히 살펴보세요.



서버리스 테스트의 경우 여전히 단위, 통합 및 end-to-end테스트를 작성해야 합니다.

- 단위 테스트 - 격리된 코드 블록에 대해 실행되는 테스트입니다. 예를 들어, 특정 항목과 대상에 대한 배송료를 계산하는 비즈니스 로직을 확인합니다.
- 통합 테스트 - 일반적으로 클라우드 환경에서 상호 작용하는 둘 이상의 구성 요소 또는 서비스를 포함하는 테스트입니다. 예를 들어, 함수가 대기열에서 이벤트를 처리하는지 확인합니다.
- E nd-to-end 테스트 - 전체 애플리케이션의 동작을 확인하는 테스트입니다. 예를 들어, 인프라가 올바르게 설정되어 있고 고객의 주문 기록을 위해 예상대로 서비스 간에 이벤트가 흐르는지 확인합니다.

서버리스 애플리케이션 테스트

일반적으로 다양한 접근 방식을 사용하여 클라우드에서 테스트, 모의 객체로 테스트, 에뮬레이터로 테스트 등의 서버리스 애플리케이션 코드 테스트를 수행합니다.

클라우드에서 테스트

클라우드에서의 테스트는 단위 테스트, 통합 테스트, 테스트를 포함한 모든 테스트 단계에서 유용합니다. end-to-end 클라우드에 배포되고 클라우드 기반 서비스와 상호 작용하는 코드에 대해 테스트를 실행합니다. 이 접근 방식은 코드 품질을 가장 정확하게 측정합니다.

클라우드에서 Lambda 함수를 디버깅하는 편리한 방법은 테스트 이벤트와 콘솔을 이용하는 것입니다. 테스트 이벤트는 함수에 대한 JSON 입력입니다. 함수에 입력이 필요하지 않은 경우 이벤트는 빈 JSON 문서(({}))가 될 수 있습니다. 콘솔은 다양한 서비스 통합을 위한 샘플 이벤트를 제공합니다. 콘솔에서 이벤트를 생성한 후 팀과 공유하여 테스트를 더 쉽고 일관성 있게 만들 수 있습니다.

> **Note**
>
> 콘솔에서 함수를 테스트하는 것이 빠르게 시작할 수 있는 방법이지만 테스트 주기를 자동화하면 애플리케이션 품질과 개발 속도가 보장됩니다.

테스트 도구

개발 피드백 루프를 가속화하기 위한 도구와 기법이 있습니다. 예를 들어, AWS SAM Accelerate와 AWS CDK 감시 모드 모두 클라우드 환경을 업데이트하는 데 필요한 시간을 줄입니다.

Moto는 데코레이터로 응답을 가로채고 시뮬레이션하는 함수를 거의 또는 전혀 수정하지 않고 테스트할 수 있도록 AWS 서비스 및 리소스를 모의하는 데 사용하는 Python 라이브러리입니다.

Powertools for AWS Lambda(Python)의 검증 기능은 데코레이터를 제공하므로 Python 함수의 입력 이벤트와 출력 응답을 검증할 수 있습니다.

자세한 내용은 블로그 게시물 Unit Testing Lambda with Python and Mock AWS Services를 참조하세요.

클라우드 배포 반복 관련 대기 시간을 줄이려면 AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) watch mode를 참조하세요. 이러한 도구는 인프라와 코드의 변경 사항을 모니터링합니다. 클라우드 환경에 증분적 업데이트를 자동으로 생성하고 배포하여 이러한 변화에 대응합니다.

이러한 도구를 사용하는 예제는 Python Test Samples 코드 리포지토리에서 사용할 수 있습니다.

AWS Lambda에서 Python 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 세 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS Distro for OpenTelemetry(ADOT) - 안전하게 프로덕션 준비가 된 AWS에서 지원하는 OpenTelemetry(OTEL) SDK의 배포입니다.
- AWS X-Ray SDK for Python — 추적 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.
- Powertools for AWS Lambda(Python) - 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 어떤 솔루션을 사용하든 X-Ray에서 엔드 투 엔드 추적 기능을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

- X-Ray 추적 해석
- 계층에 런타임 종속성 저장(X-Ray SDK)

추적에 Powertools for AWS Lambda(Python) 및 AWS SAM 사용

다음 단계에 따라 AWS SAM를 사용하여 통합 Powertools for AWS Lambda(Python) 모듈이 포함된 샘플 Hello World Python 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 함수가 hello world 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Python 3.9
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World Python 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

추적 출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

추적에 Powertools for AWS Lambda(Python) 및 AWS CDK 사용

다음 단계에 따라 AWS CDK를 사용하여 통합 Powertools for AWS Lambda(Python) 모듈이 포함된 샘플 Hello World Python 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API

Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 함수가 hello world 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Python 3.9
- AWS CLI 버전 2
- AWS CDK 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS CDK 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

```
mkdir hello-world
cd hello-world
```

2. 앱을 초기화합니다.

```
cdk init app --language python
```

3. Python 종속 구성 요소를 설치합니다.

```
pip install -r requirements.txt
```

4. 루트 폴더 아래에 lambda_function 디렉터리를 생성합니다.

```
mkdir lambda_function
cd lambda_function
```

5. 파일 app.py를 생성하고 파일에 다음 코드를 추가합니다. Lambda 함수에 대한 코드입니다.

```
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 디렉터리를 엽니다. hello_world_stack.py라는 파일이 있어야 합니다.

```
cd ..
cd hello_world
```

7. hello_world_stack.py를 열고 파일에 다음 코드를 추가합니다. 여기에는 Lambda 함수를 생성하고, Powertools에 대한 환경 변수를 구성하고, 로그 보존을 1주일로 설정하는 Lambda Constructor와 REST API를 생성하는 ApiGatewayv1 Constructor가 포함됩니다.

```
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. 애플리케이션 배포

```
cd ..
cdk deploy
```

9. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. API 엔드포인트 호출:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

11. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

기록 출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
cdk destroy
```

ADOT를 사용하여 Python 함수 계측

ADOT는 OTel SDK를 사용하여 원격 측정 데이터를 수집하는 데 필요한 모든 것을 패키징할 수 있는 완전 관리형 Lambda 계층을 제공합니다. 이 계층을 사용하면 모든 함수 코드를 수정하지 않고도 Lambda 함수를 계측할 수 있습니다. 계층을 구성하여 OTel의 사용자 지정 초기화를 수행할 수도 있습니다. 자세한 내용은 ADOT 설명서의 Lambda에서 ADOT 컬렉터에 대한 사용자 지정 구성을 참조하세요.

Python 런타임의 경우 ADOT Python용 AWS 관리형 Lambda 계층을 추가하여 함수를 자동으로 계측할 수 있습니다. 이 계층은 arm64 및 x86_64 아키텍처 모두에서 작동합니다. 이 계층을 추가하는 방법에 대한 자세한 지침은 ADOT 설명서의 AWS Distro for OpenTelemetry Lambda Support for Python을 참조하세요.

X-Ray SDK를 사용하여 Python 함수 계측

Lambda 함수가 애플리케이션의 다른 리소스에 대해 수행하는 호출에 대한 세부 정보를 기록하려면 AWS X-Ray SDK for Python를 사용할 수도 있습니다. SDK를 가져오려면 애플리케이션의 종속성에 `aws-xray-sdk` 패키지를 추가합니다.

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

함수 코드에서 `boto3` 라이브러리를 `aws_xray_sdk.core` 모듈로 패치하여 AWS SDK 클라이언트를 계측할 수 있습니다.

Example 함수 — AWS SDK 클라이언트 추적

```
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)))
  ...
```

올바른 종속성을 추가하고 필요한 코드를 변경한 후 Lambda 콘솔 또는 API를 통해 함수의 구성에서 추적을 활성화합니다.

Lambda 콘솔을 사용하여 추적 활성화

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

Lambda API를 사용하여 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 Lambda 함수에 대한 추적을 구성하고 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation을 사용하여 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray 추적 해석

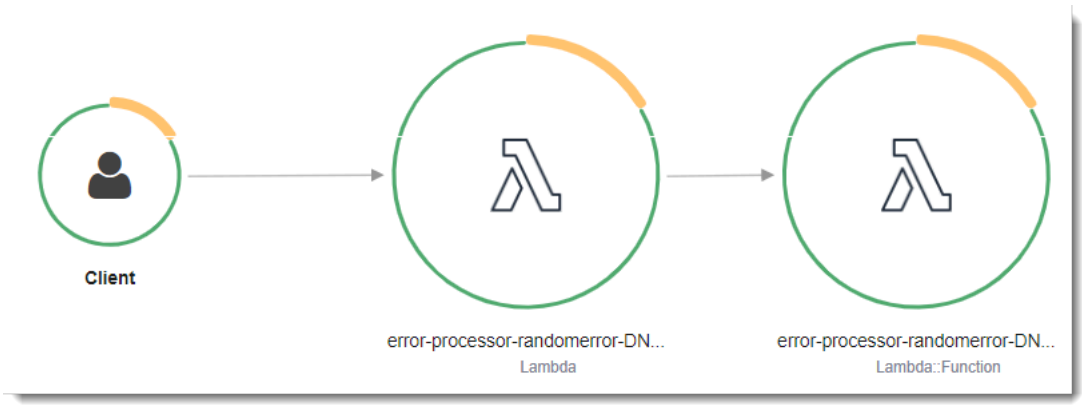
함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

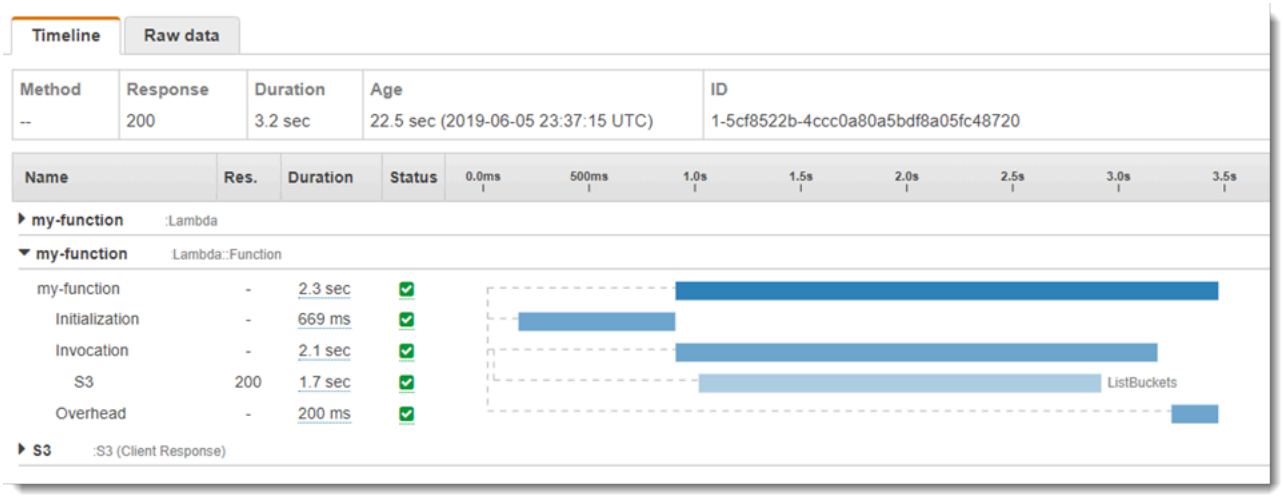
> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이

AWS::Lambda::Function입니다. AWS::Lambda 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. AWS::Lambda::Function 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 AWS::Lambda::Function 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 AWS X-Ray SDK for Python를 참조하십시오.

> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

계층에 런타임 종속성 저장(X-Ray SDK)

X-Ray SDK를 사용하여 AWS SDK 클라이언트를 계측하는 경우 함수 코드와 배포 패키지가 상당히 커질 수 있습니다. 함수 코드를 업데이트할 때마다 런타임 종속성을 업로드하지 않으려면 X-Ray SDK를 Lambda 계층에 패키징합니다.

다음 예제에서는 AWS X-Ray SDK for Python를 저장하는 `AWS::Serverless::LayerVersion` 리소스를 보여줍니다.

Example template.yml – 종속성 계층

```
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
```

이 구성을 사용하면 런타임 종속성을 변경하는 경우 라이브러리 계층만 업데이트하면 됩니다. 함수 배포 패키지에는 코드만 포함되어 있으므로 이는 업로드 시간을 줄일 수 있습니다.

종속성 계층을 만들려면 배포 전에 계층 아카이브를 생성하기 위해 빌드를 변경해야 합니다. 사용 가능한 예제는 blank-python 샘플 애플리케이션을 참조하십시오.

Ruby를 사용하여 Lambda 함수 빌드

AWS Lambda에서 Ruby 코드를 실행할 수 있습니다. Lambda는 이벤트 처리를 위해 코드를 실행하는 Ruby를 위한 런타임을 제공합니다. 코드는 사용자가 관리하는 AWS Identity and Access Management(IAM) 역할의 자격 증명을 사용하여 AWS SDK for Ruby가 포함된 환경에서 실행됩니다. Ruby 런타임에 포함된 SDK 버전에 대해 자세히 알아보려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

Lambda는 다음과 같은 Ruby 런타임을 지원합니다.

Ruby

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
Ruby 3.3	`ruby3.3`	Amazon Linux 2023			
Ruby 3.2	`ruby3.2`	Amazon Linux 2			

Ruby 함수를 만들려면

1. Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. 다음 설정을 구성합니다:
   - 함수 이름: 함수의 이름을 입력합니다.
   - 런타임: Ruby 3.2를 선택합니다.
4. 함수 생성을 선택합니다.
5. 테스트 이벤트를 구성하려면 테스트를 선택합니다.
6. 이벤트 이름에 **`test`**를 입력합니다.
7. Save changes(변경 사항 저장)를 선택합니다.
8. 함수를 호출하려면 테스트를 선택합니다.

콘솔은 `lambda_function.rb`(이)라는 단일 소스 파일로 Lambda 함수를 생성합니다. 이 파일을 편집하고 기본 제공 코드 편집기에서 더 많은 파일을 추가할 수 있습니다. 변경 사항을 저장하려면 [Save]를 선택합니다. 그런 다음 코드를 실행하려면 테스트를 선택합니다.

> **Note**
>
> Lambda 콘솔은 AWS Cloud9를 사용하여 브라우저에서 통합 개발 환경(IDE)을 제공합니다. AWS Cloud9을 사용하면 자신의 환경에서 Lambda 함수를 개발할 수도 있습니다. 자세한 내용은 AWS Cloud9 사용 설명서의 Working with AWS Lambda functions using the AWS Toolkit을 참조하세요.

`lambda_function.rb` 파일은 이벤트 객체와 컨텍스트 객체를 취하는 `lambda_handler`라는 이름의 함수를 내보냅니다. 이는 함수가 호출될 때 Lambda가 호출하는 핸들러 함수입니다. Ruby 함수 런타임은 Lambda에서 호출 이벤트를 가져와 핸들러로 전달합니다. 함수 구성에서 핸들러 값은 `lambda_function.lambda_handler`입니다.

함수 코드를 저장하면 Lambda 콘솔에서 .zip 파일 아카이브 배포 패키지를 만듭니다. 콘솔 외부에서(IDE를 사용해) 함수 코드를 개발하는 경우 Lambda 함수에 코드를 업로드하려면 배포 패키지를 생성해야 합니다.

> **Note**
>
> 로컬 환경에서 애플리케이션 개발을 시작하려면 이 가이드의 GitHub 리포지토리에서 사용할 수 있는 샘플 애플리케이션 중 하나를 배포하세요.
>
> Ruby의 샘플 Lambda 애플리케이션
>
> - blank-ruby – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Ruby 함수입니다.
> - AWS Lambda용 Ruby 코드 샘플 – AWS Lambda와 상호 작용하는 방법을 보여주는 Ruby로 작성된 코드 샘플입니다.

함수 런타임은 호출 이벤트 외에도 컨텍스트 객체를 핸들러에 전달합니다. 컨텍스트 객체에는 호출, 함수 및 실행 환경에 관한 추가 정보가 포함되어 있습니다. 자세한 내용은 환경 변수에서 확인할 수 있습니다.

Lambda 함수는 CloudWatch Logs 로그 그룹을 함께 제공됩니다. 함수 런타임은 각 호출에 대한 세부 정보를 CloudWatch Logs에 보냅니다. 호출 중 함수가 출력하는 로그를 전달합니다. 함수가 오류를 반환하면 Lambda은 오류에 서식을 지정한 후 이를 호출자에게 반환합니다.

주제

런타임에 포함된 SDK 버전

Ruby 런타임에 포함된 AWS SDK 버전은 런타임 버전 및 사용자의 AWS 리전에 따라 달라집니다. AWS SDK for Ruby는 모듈식으로 설계되었으며 AWS 서비스와 구분됩니다. 사용 중인 런타임에 포함된 특정 서비스 gem의 버전 번호를 찾으려면 다음 형식의 코드를 사용하여 Lambda 함수를 생성합니다. `aws-sdk-s3` 및 `Aws::S3`을 코드에서 사용하는 서비스 gem의 이름으로 바꾸십시오.

```
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
```

또 다른 Ruby JIT(YJIT) 활성화

Ruby 3.2 런타임은 가볍고 최소한의 Ruby JIT 컴파일러인 YJIT를 지원합니다. YJIT는 Ruby 인터프리터보다 훨씬 더 높은 성능을 제공하지만 메모리 사용량도 더 많습니다. YJIT는 Ruby on Rails 워크로드에 권장됩니다.

YJIT는 기본적으로 활성화되어 있지 않습니다. Ruby 3.2 함수에 대해 YJIT를 활성화하려면 RUBY_YJIT_ENABLE 환경 변수를 1로 설정합니다. YJIT가 활성화되었는지 확인하려면 RubyVM::YJIT.enabled? 메서드의 결과를 인쇄합니다.

Example - YJIT가 활성화되어 있는지 확인

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Ruby에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

다음 예제에서 파일 `function.rb`는 `handler`라는 이름의 핸들러 메서드를 정의합니다. 핸들러 함수는 2개의 객체를 입력으로 사용하며 JSON 문서를 반환합니다.

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

함수 구성에서 `handler` 설정은 핸들러를 찾을 위치를 Lambda에 알려줍니다. 앞의 예제에서 이 설정의 올바른 값은 **`function.handler`**입니다. 여기에는 점으로 구분된 2개의 이름 즉, 파일 이름과 핸들러 메서드의 이름이 포함됩니다.

하나의 클래스에서 핸들러 메서드를 정의할 수도 있습니다. 다음 예제에서는 `process`라는 이름의 모듈에서 `Handler`라는 이름의 클래스에 `LambdaFunctions`라는 이름의 핸들러 메서드를 정의합니다.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

이 경우, 핸들러 설정은 **`source.LambdaFunctions::Handler.process`**입니다.

핸들러가 허용하는 2개의 객체로는 호출 이벤트 및 컨텍스트가 있습니다. 해당 이벤트는 호출자가 제공하는 페이로드가 포함된 Ruby 객체입니다. 페이로드가 JSON 문서인 경우, 이벤트 객체는 Ruby 해시입니다. 그렇지 않은 경우, 이 객체는 문자열입니다. 컨텍스트 객체에는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드와 속성이 있습니다.

함수 핸들러는 Lambda 함수가 호출될 때마다 실행됩니다. 핸들러 외부의 정적 코드는 함수의 인스턴스당 한 번씩 실행됩니다. 핸들러가 SDK 클라이언트 및 데이터베이스 연결과 같은 리소스를 사용하는 경우, 핸들러 메서드 외부에서 그러한 리소스를 생성하면 다중 호출 시 이 리소스를 다시 사용할 수 있습니다.

함수의 각 인스턴스는 다중 호출 이벤트를 처리할 수 있으며 다만 이벤트를 한 번에 하나씩만 처리합니다. 주어진 시간에 하나의 이벤트를 처리하는 인스턴스의 수는 함수의 동시성을 나타냅니다. Lambda 실행 환경에 대한 자세한 내용은 Lambda 실행 환경 단원을 참조하세요.

Ruby Lambda 함수에 대한 .zip 파일 아카이브 작업

AWS Lambda 함수의 코드는 함수의 핸들러 코드와 코드가 의존하는 추가 종속 항목(젬)을 포함하는 .rb 파일로 구성됩니다. Lambda에 이 함수 코드를 배포하려면 배포 패키지를 사용합니다. 이 패키지는 .zip 파일 아카이브 또는 컨테이너 이미지일 수 있습니다. Ruby에서 컨테이너 이미지를 사용하는 방법에 대한 자세한 내용은 컨테이너 이미지로 Ruby Lambda 함수 배포를 참조하세요.

배포 패키지를 .zip 파일 아카이브로 생성하려면 명령줄 도구의 기본 제공 .zip 파일 아카이브 유틸리티 또는 7zip과 같은 기타 .zip 파일 유틸리티를 사용합니다. 다음 섹션에 표시된 예제에서는 Linux 또는 MacOS 환경에서 명령줄 `zip` 도구를 사용한다고 가정합니다. Windows에서 동일한 명령을 사용하려면 Windows Subsystem for Linux를 설치하여 Ubuntu 및 Bash의 Windows 통합 버전을 가져옵니다.

Lambda는 POSIX 파일 권한을 사용하므로 .zip 파일 아카이브를 생성하기 전에 배포 패키지 폴더에 대한 권한을 설정해야 할 수 있습니다.

다음 섹션의 예제 명령은 Bundler 유틸리티를 사용하여 배포 패키지에 종속 항목을 추가합니다. 번들러를 설치하려면 다음 명령을 실행하세요.

```
gem install bundler
```

Sections

Ruby의 종속 항목

Ruby 런타임을 사용하는 Lambda 함수의 경우 종속 항목은 Ruby 젬일 수 있습니다. .zip 아카이브를 사용하여 함수를 배포할 때 이러한 종속 항목을 함수 코드와 함께 .zip 파일에 추가하거나 Lambda 계층을 사용할 수 있습니다. 계층은 추가 코드 또는 기타 콘텐츠를 포함할 수 있는 별도의 .zip 파일입니다. Lambda 계층 사용에 대해 자세히 알아보려면 Lambda 계층 섹션을 참조하세요.

Ruby 런타임에는 AWS SDK for Ruby가 포함되어 있습니다. 함수에서 SDK를 사용하는 경우 SDK를 코드와 번들링할 필요가 없습니다. 그러나 종속 항목을 완전히 제어하거나 특정 버전의 SDK를 사용하

기 위해 함수의 배포 패키지에 SDK를 추가할 수 있습니다. SDK를 .zip 파일에 포함하거나 Lambda 계층을 사용하여 추가할 수 있습니다. .zip 파일 또는 Lambda 계층의 종속 항목은 런타임에 포함된 버전보다 우선합니다. 런타임 버전에 포함된 SDK for Ruby 버전을 확인하려면 the section called “런타임에 포함된 SDK 버전” 섹션을 참조하세요.

AWS Shared Responsibility Model에서는 사용자가 함수의 배포 패키지에 있는 모든 종속 항목을 관리해야 합니다. 여기에는 업데이트 및 보안 패치 적용이 포함됩니다. 함수의 배포 패키지에서 종속 항목을 업데이트하려면 먼저 새 .zip 파일을 생성한 다음 Lambda에 업로드합니다. 자세한 내용은 종속 항목이 포함된 .zip 배포 패키지 생성 및 .zip 파일을 사용하여 Ruby Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 없는 .zip 배포 패키지 생성

함수 코드에 종속 항목이 없는 경우 .zip 파일에는 함수의 핸들러 코드가 있는 .rb 파일만 포함됩니다. 선호하는 zip 유틸리티를 사용하여 루트에 .rb 파일이 있는 .zip 파일을 생성합니다. .rb 파일이.zip 파일의 루트에 없는 경우 Lambda는 코드를 실행할 수 없습니다.

.zip 파일을 배포하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트하는 방법을 알아보려면 .zip 파일을 사용하여 Ruby Lambda 함수 생성 및 업데이트 섹션을 참조하세요.

종속 항목이 포함된 .zip 배포 패키지 생성

함수 코드가 추가 Ruby 젬에 의존하는 경우 이러한 종속 항목을 함수 코드와 함께 .zip 파일에 추가하거나 Lambda 계층을 사용할 수 있습니다. 이 섹션의 지침에서는 .zip 배포 패키지에 종속 항목을 포함하는 방법을 보여줍니다. 계층에 종속 항목을 포함하는 방법에 대한 지침은 the section called “종속 항목을 위한 Ruby 계층 생성” 섹션을 참조하세요.

함수 코드가 프로젝트 디렉터리의 `lambda_function.rb`라는 파일에 저장되어 있다고 가정해 보겠습니다. 다음 예제 CLI 명령은 함수 코드와 해당 종속 항목을 포함하는 `my_deployment_package.zip`라는 .zip 파일을 생성합니다.

배포 패키지를 만드는 방법

1. 프로젝트 디렉터리에서 `Gemfile`을 생성하여 종속 항목을 지정합니다.

   ```
   bundle init
   ```

2. 선호하는 텍스트 편집기를 사용해 `Gemfile`을 편집하여 함수의 종속 항목을 지정합니다. 예를 들어 TZInfo 젬을 사용하려면 `Gemfile`을 다음과 같이 편집하세요.

```
source "https://rubygems.org"
gem "tzinfo"
```

3. 다음 명령을 실행하여 Gemfile에 지정된 젬을 프로젝트 디렉터리에 설치합니다. 이 명령은 vendor/bundle을 젬 설치를 위한 기본 경로로 설정합니다.

```
bundle config set --local path 'vendor/bundle' && bundle install
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
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**
>
> 나중에 젬을 전역적으로 다시 설치하려면 다음 명령을 실행합니다.
>
> ```
> bundle config set --local system 'true'
> ```

4. 함수의 핸들러 코드와 이전 단계에서 설치한 종속 항목과 함께 lambda_function.rb 파일이 포함된 .zip 파일 아카이브를 생성합니다.

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
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%)
```

```
...
```

종속 항목을 위한 Ruby 계층 생성

이 섹션의 지침은 계층에 종속 항목을 포함하는 방법을 보여줍니다. 배포 패키지에 종속 항목을 포함하는 방법에 대한 지침은 the section called “종속 항목이 포함된 .zip 배포 패키지 생성” 섹션을 참조하세요.

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 `/opt` 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 `PATH` 변수에는 `/opt` 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. `PATH` 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

- `ruby/gems/2.7.0` (`GEM_PATH`)
- `ruby/lib` (`RUBYLIB`)

예를 들어, 계층.zip 파일 구조는 다음과 같을 수 있습니다.

```
json.zip
# ruby/gems/2.7.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

또한 Lambda는 `/opt/lib` 디렉터리의 모든 라이브러리와 `/opt/bin` 디렉터리의 모든 바이너리를 자동으로 감지합니다. Lambda가 계층 콘텐츠를 제대로 찾을 수 있도록 다음 구조로 계층을 생성할 수도 있습니다.

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
```

```
    | bin_2
```

계층을 패키징한 후 the section called “계층 생성 및 삭제” 및 the section called “계층 추가”을 참조하여 계층 설정을 완료합니다.

네이티브 라이브러리로 .zip 배포 패키지 생성

`nokogiri`, `nio4r`, `mysql`과 같은 많은 일반적인 Ruby 젬은 C로 작성된 네이티브 확장을 포함합니다. C 코드가 포함된 라이브러리를 배포 패키지에 추가할 때 패키지가 Lambda 실행 환경과 호환되도록 올바르게 빌드해야 합니다.

프로덕션 애플리케이션의 경우 AWS Serverless Application Model(AWS SAM)을 사용하여 코드를 빌드하고 배포하는 것이 좋습니다. AWS SAM에서 `sam build --use-container` 옵션을 사용하여 Lambda와 유사한 Docker 컨테이너 내에 함수를 빌드합니다. AWS SAM을 사용하여 함수 코드를 배포하는 방법에 대한 자세한 내용은 AWS SAM 개발자 안내서의 Building applications를 참조하세요.

AWS SAM을 사용하지 않고 기본 확장이 있는 젬이 포함된 .zip 배포 패키지를 생성하려면 대신 컨테이너를 사용하여 Lambda Ruby 런타임 환경과 동일한 환경에서 종속 항목을 번들링할 수 있습니다. 이 단계를 완료하려면 빌드 머신에 Docker가 설치되어 있어야 합니다. Docker 설치에 대해 자세히 알아보려면 Install Docker Engine을 참조하세요.

Docker 컨테이너에서.zip 배포 패키지를 생성하려면

1. 컨테이너를 저장할 로컬 빌드 머신에 폴더를 만듭니다. 해당 폴더 안에 `dockerfile`이라는 파일을 생성하고 다음 코드를 붙여넣습니다.

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. `dockerfile`을 생성한 폴더 내에서 다음 명령을 실행하여 Docker 컨테이너를 생성합니다.

   ```
   docker build -t awsruby32 .
   ```

3. 함수의 핸들러 코드와 함수의 종속 항목을 지정하는 `Gemfile`이 있는 `.rb` 파일이 포함된 프로젝트 디렉터리로 이동합니다. 해당 디렉터리 내에서 다음 명령을 실행하여 Lambda Ruby 컨테이너를 시작합니다.

Linux/MacOS

```
docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
```

> **Note**
>
> macOS에서는 요청한 이미지의 플랫폼이 감지된 호스트 플랫폼과 일치하지 않는다는 경고가 표시될 수 있습니다. 이 경고는 무시하세요.

Windows PowerShell

```
docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
```

컨테이너가 시작되면 bash 프롬프트가 표시됩니다.

```
bash-4.2#
```

4. `Gemfile`에 지정된 젬을 로컬 `vendor/bundle` 디렉터리에 설치하고 종속 항목을 설치하도록 번들 유틸리티를 구성합니다.

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. 함수 코드 및 해당 종속 항목을 사용하여 .zip 배포 패키지를 생성합니다. 이 예제에서 함수의 핸들러 코드가 포함된 파일의 이름은 `lambda_function.rb`입니다.

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. 컨테이너를 종료하고 로컬 프로젝트 디렉터리로 돌아갑니다.

```
bash-4.2# exit
```

이제 .zip 파일 배포 패키지를 사용하여 Lambda 함수를 생성하거나 업데이트할 수 있습니다. .zip 파일을 사용하여 Ruby Lambda 함수 생성 및 업데이트 섹션 참조

.zip 파일을 사용하여 Ruby Lambda 함수 생성 및 업데이트

.zip 배포 패키지를 생성한 후에는 이를 사용하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트할 수 있습니다. Lambda 콘솔, AWS Command Line Interface 및 Lambda API를 사용하여 .zip 패키지를 배포할 수 있습니다. AWS Serverless Application Model(AWS SAM) 및 AWS CloudFormation을 사용하여 Lambda 함수를 생성하고 업데이트할 수도 있습니다.

Lambda용 .zip 배포 패키지의 최대 크기는 250MB(압축 해제됨)입니다. 이 제한은 Lambda 계층을 포함하여 업로드하는 모든 파일의 합산 크기에 적용됩니다.

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 chmod 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

콘솔을 사용하여.zip 파일로 함수 생성 및 업데이트

새 함수를 생성하려면 먼저 콘솔에서 함수를 생성한 다음.zip 아카이브를 업로드해야 합니다. 기존 함수를 업데이트하려면 함수에 대한 페이지를 연 다음 동일한 절차에 따라 업데이트된 .zip 파일을 추가합니다.

.zip 파일이 50MB 미만인 경우 로컬 컴퓨터에서 직접 파일을 업로드하여 함수를 생성하거나 업데이트할 수 있습니다. 50MB보다 큰 .zip 파일의 경우 먼저 패키지를 Amazon S3 버킷에 업로드해야 합니다. AWS Management Console을 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 Amazon S3 시작하기를 참조하세요. AWS CLI를 사용하여 파일을 업로드하려면 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

새 함수 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.
3. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름에 함수 이름을 입력합니다.
   b. 런타임에서 사용할 런타임을 선택합니다.
   c. (선택 사항) 아키텍처에서 함수에 대한 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수에 대한 .zip 배포 패키지가 선택한 명령 세트 아키텍처와 호환되는지 확인합니다.
4. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 실행 역할을 사용할 수 있습니다.
5. 함수 생성을 선택합니다. Lambda에서 선택한 런타임을 사용하여 기본 'Hello World' 함수를 생성합니다.

로컬 시스템에서 .zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서.zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. .zip 파일을 선택합니다.
5. .zip 파일을 업로드하려면 다음을 수행합니다.
   a. 업로드를 선택한 다음 파일 선택기에서.zip 파일을 선택합니다.
   b. Open을 선택합니다.
   c. Save(저장)를 선택합니다.

Amazon S3 버킷에서.zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서 새 .zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. Amazon S3 위치를 선택합니다.

5. .zip 파일의 Amazon S3 링크 URL을 붙여 넣고 저장을 선택합니다.

콘솔 코드 편집기를 사용하여.zip 파일 함수 업데이트

.zip 배포 패키지를 사용하는 일부 함수의 경우 Lambda 콘솔의 기본 제공 코드 편집기를 사용하여 함수 코드를 직접 업데이트할 수 있습니다. 이 기능을 사용하려면 함수가 다음 조건을 충족해야 합니다.

- 함수에서 해석된 언어 런타임(Python, Node.js 또는 Ruby) 중 하나를 사용해야 합니다.
- 함수의 배포 패키지가 3MB보다 작아야 합니다.

컨테이너 이미지 배포 패키지가 있는 함수의 함수 코드는 콘솔에서 직접 편집할 수 없습니다.

콘솔 코드 편집기를 사용하여 함수 코드 업데이트

1. Lambda 콘솔의 함수 페이지를 열고 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 소스 코드 파일을 선택하고 통합 코드 편집기에서 편집합니다.
4. 코드 편집을 마치면 배포를 선택하여 변경 사항을 저장하고 함수를 업데이트합니다.

AWS CLI를 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CLI를 사용하여 새 함수를 생성하거나.zip 파일로 기존 함수를 업데이트할 수 있습니다. create-function 및 update-function-code 명령을 사용하여 .zip 패키지를 배포합니다. .zip 파일이 50MB보다 작은 경우 로컬 빌드 시스템의 파일 위치에서 .zip 패키지를 업로드할 수 있습니다. 더 큰 파일의 경우 Amazon S3 버킷에서 .zip 패키지를 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> AWS CLI를 사용하여 Amazon S3 버킷에서 .zip 파일을 업로드하는 경우 버킷은 함수와 동일한 AWS 리전에 있어야 합니다.

AWS CLI에서 .zip 파일을 사용하여 새 함수를 생성하려면 다음을 지정해야 합니다.

- 함수의 이름(`--function-name`)
- 함수의 런타임(`--runtime`)

- 함수의 실행 역할(--role)의 Amazon 리소스 이름(ARN)
- 함수 코드에 있는 핸들러 메서드의 이름(--handler)

.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 --zip-file 옵션을 사용하여 파일 경로를 지정합니다.

```
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 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 --code 옵션을 사용합니다. 버전이 지정된 객체에만 S3ObjectVersion 파라미터를 사용해야 합니다.

```
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
```

CLI를 사용하여 기존 함수를 업데이트하려면 --function-name 파라미터를 사용하여 함수 이름을 지정합니다. 함수 코드를 업데이트하는 데 사용할.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 --zip-file 옵션을 사용하여 파일 경로를 지정합니다.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 --s3-bucket 및 --s3-key 옵션을 사용합니다. 버전이 지정된 객체에만 --s3-object-version 파라미터를 사용해야 합니다.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API를 사용하여.zip 파일로 함수 생성 및 업데이트

.zip 파일 아카이브를 사용하여 함수를 생성하고 업데이트하려면 다음 API 작업을 사용합니다.

- CreateFunction
- UpdateFunctionCode

AWS SAM을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS Serverless Application Model(AWS SAM)은 AWS에서 서버리스 애플리케이션을 빌드하고 실행하는 프로세스를 간소화하는 데 도움이 되는 도구 키트입니다. YAML 또는 JSON 템플릿에서 애플리케이션의 리소스를 정의하고 AWS SAM Command Line Interface(AWS SAM CLI)를 사용하여 애플리케이션을 빌드, 패키징 및 배포합니다. AWS SAM 템플릿에서 Lambda 함수를 빌드하면 AWS SAM은 함수 코드와 사용자가 지정하는 종속 항목을 사용하여 .zip 배포 패키지 또는 컨테이너 이미지를 자동으로 생성합니다. AWS SAM을 사용하여 Lambda 함수를 빌드하고 배포하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Getting started with AWS SAM을 참조하세요.

AWS SAM을 사용하여 기존 .zip 파일 아카이브로 Lambda 함수를 생성할 수도 있습니다. AWS SAM을 사용하여 Lambda 함수를 생성하려면 Amazon S3 버킷 또는 빌드 시스템의 로컬 폴더에 .zip 파일을 저장할 수 있습니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS SAM 템플릿에서 `AWS::Serverless::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `CodeUri` - 함수 코드의 Amazon S3 URI, 로컬 폴더 경로 또는 FunctionCode 객체로 설정됨
- `Runtime` - 선택한 런타임으로 설정됨

AWS SAM을 사용하면 .zip 파일이 50MB보다 큰 경우 Amazon S3 버킷에 먼저 파일을 업로드할 필요가 없습니다. AWS SAM은 로컬 빌드 시스템의 위치에서 허용되는 최대 크기 250MB(압축 해제)까지 .zip 패키지를 업로드할 수 있습니다.

AWS SAM에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS SAM 개발자 안내서의 AWS::Serverless::Function을 참조하세요.

AWS CloudFormation을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CloudFormation을 사용하여 .zip 파일 아카이브로 Lambda 함수를 생성할 수 있습니다. .zip 파일에서 Lambda 함수를 생성하려면 먼저 Amazon S3 버킷에 파일을 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `Code` - `S3Bucket` 및 `S3Key` 필드에 Amazon S3 버킷 이름과 .zip 파일 이름을 입력합니다.
- `Runtime` - 선택한 런타임으로 설정됨

AWS CloudFormation에서 생성하는 .zip 파일은 4MB를 초과할 수 없습니다. AWS CloudFormation에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function을 참조하세요.

컨테이너 이미지로 Ruby Lambda 함수 배포

Ruby Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- Ruby용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.

- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Ruby용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Ruby용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

주제

- AWSRuby용 기본 이미지
- Ruby용 AWS 기본 이미지 사용
- 런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

AWSRuby용 기본 이미지

AWS는 Ruby에 대한 다음과 같은 기본 이미지를 제공합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
3.3	Ruby 3.3	Amazon Linux 2023	GitHub의 Ruby 3.3용 Dockerfile	
3.2	Ruby 3.2	Amazon Linux 2	GitHub의 Ruby 3.2용 Dockerfile	

Amazon ECR 리포지토리: gallery.ecr.aws/lambda/ruby

Ruby용 AWS 기본 이미지 사용

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker
- Ruby

기본 이미지에서 이미지 생성

Ruby용 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir example
cd example
```

2. Gemfile라는 파일을 새로 생성합니다. 여기에 애플리케이션의 필수 RubyGems 패키지를 나열합니다. AWS SDK for Ruby는 RubyGems에서 사용할 수 있습니다. 설치할 특정 AWS 서비스 gem을 선택해야 합니다. 예를 들어, Lambda용 Ruby gem을 사용하려면 Gemfile이 다음과 같아야 합니다.

```
source 'https://rubygems.org'
```

```
gem 'aws-sdk-lambda'
```

또는 aws-sdk gem에는 사용 가능한 모든 AWS 서비스 gem이 포함되어 있습니다. 이 gem은 매우 크므로 많은 AWS 서비스에 의존하는 경우에만 사용하는 것이 좋습니다.

3. 번들 설치를 사용하여 Gemfile에 지정된 종속 항목을 설치합니다.

```
bundle install
```

4. `lambda_function.rb`라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

Example Ruby 함수

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 새 Dockerfile을 생성합니다. 다음은 AWS 기본 이미지를 사용하는 예제 Dockerfile입니다. 이 Dockerfile에서는 다음 구성을 사용합니다.

- `FROM` 속성을 기본 이미지의 URI로 설정합니다.
- `COPY` 명령을 사용하여 함수 코드와 런타임 종속성을 Lambda 정의 환경 변수인 `{LAMBDA_TASK_ROOT}`에 복사합니다.
- `CMD` 인수를 Lambda 함수 핸들러로 설정합니다.

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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 이 예제에서 docker-image는 이미지 이름이고 test는 태그입니다.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

이 명령은 이미지를 컨테이너로 실행하고 localhost:9000/2015-03-31/functions/function/invocations에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

2. 새 터미널 창에서 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 컨테이너 ID를 가져옵니다.

```
docker ps
```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - `111122223333`를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API을 확장합니다.

RubyGems.org 패키지 관리자를 사용하여 Ruby용 런타임 인터페이스 클라이언트를 설치합니다.

```
gem install aws_lambda_ric
```

GitHub에서 Ruby 런타임 인터페이스 클라이언트를 다운로드할 수도 있습니다. 런타임 인터페이스 클라이언트는 Ruby 버전 2.5.x~2.7.x를 지원합니다.

다음 예제에서는 비AWS 기본 이미지를 사용하여 Ruby용 컨테이너 이미지를 빌드하는 방법을 보여줍니다. 예제 Dockerfile에서는 공식 Ruby 기본 이미지를 사용합니다. Dockerfile에는 런타임 인터페이스 클라이언트가 포함되어 있습니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- Docker
- Ruby

대체 기본 이미지에서 이미지 생성

대체 기본 이미지를 사용하여 컨테이너 이미지 생성

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

   ```
   mkdir example
   cd example
   ```

2. `Gemfile`라는 파일을 새로 생성합니다. 여기에 애플리케이션의 필수 RubyGems 패키지를 나열합니다. AWS SDK for Ruby는 RubyGems에서 사용할 수 있습니다. 설치할 특정 AWS 서비스 gem을 선택해야 합니다. 예를 들어, Lambda용 Ruby gem을 사용하려면 Gemfile이 다음과 같아야 합니다.

   ```
   source 'https://rubygems.org'

   gem 'aws-sdk-lambda'
   ```

   또는 aws-sdk gem에는 사용 가능한 모든 AWS 서비스 gem이 포함되어 있습니다. 이 gem은 매우 크므로 많은 AWS 서비스에 의존하는 경우에만 사용하는 것이 좋습니다.

3. 번들 설치를 사용하여 Gemfile에 지정된 종속 항목을 설치합니다.

   ```
   bundle install
   ```

4. `lambda_function.rb`라는 파일을 새로 생성합니다. 테스트를 위해 다음 샘플 함수 코드를 파일에 추가하거나 자체 샘플 함수 코드를 사용할 수 있습니다.

   Example Ruby 함수

   ```
   module LambdaFunction
     class Handler
       def self.process(event:,context:)
         "Hello from Lambda!"
       end
     end
   end
   ```

5. 새 Dockerfile을 생성합니다. 다음 Dockerfile은 AWS 기본 이미지 대신 Ruby 기본 이미지를 사용합니다. Dockerfile에는 이미지가 Lambda와 호환되도록 하는 Ruby용 런타임 인터페이스 클라이언트가 포함되어 있습니다. 또는 애플리케이션의 Gemfile에 런타임 인터페이스 클라이언트를 추가할 수 있습니다.

   - `FROM` 속성을 Ruby 기본 이미지로 설정합니다.
   - 함수 코드의 디렉터리와 해당 디렉터리를 가리키는 환경 변수를 생성합니다. 이 예제에서 디렉토리는 `/var/task`로, Lambda 실행 환경을 미러링합니다. 그러나 Docker파일은 AWS 기본 이미지를 사용하지 않으므로 함수 코드의 디렉터리를 선택할 수 있습니다.
   - Docker 컨테이너가 시작될 때 실행할 모듈로 `ENTRYPOINT`를 설정합니다. 이 경우 모듈은 런타임 인터페이스 클라이언트입니다.
   - `CMD` 인수를 Lambda 함수 핸들러로 설정합니다.

   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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. 에뮬레이터를 이미지에 빌드하거나 다음 절차를 사용하여 로컬 시스템에 설치할 수 있습니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 설치 및 실행

1. 프로젝트 디렉터리에서 다음 명령을 실행하여 GitHub에서 런타임 인터페이스 에뮬레이터(x86-64 아키텍처)를 다운로드하고 로컬 시스템에 설치합니다.

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 에뮬레이터를 설치하려면 이전 명령의 GitHub 리포지토리 URL을 다음과 같이 바꿉니다.

```
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 에뮬레이터를 설치하려면 `$downloadLink`을(를) 다음과 같이 바꿉니다.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process`는 Docker 파일의 `CMD` 다음에 오는 `ENTRYPOINT`입니다.

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
```

이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

3. 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 컨테이너 ID를 가져옵니다.

```
docker ps
```

5. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

AWS Lambda 컨텍스트 객체(Ruby)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러.에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다.

컨텍스트 메서드

- `get_remaining_time_in_millis` – 실행 시간이 초과되기까지 남은 시간(밀리초)을 반환합니다.

컨텍스트 속성

- `function_name` – Lambda 함수의 이름입니다.
- `function_version` – 함수의 버전입니다.
- `invoked_function_arn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `memory_limit_in_mb` – 함수에 할당된 메모리의 양입니다.
- `aws_request_id` – 호출 요청의 식별자입니다.
- `log_group_name` – 함수에 대한 로그 그룹입니다.
- `log_stream_name` – 함수 인스턴스에 대한 로그 스트림입니다.
- `deadline_ms` – 실행 시간이 초과된 날짜를 Unix 시간 형식에 따른 밀리초 단위로 나타낸 값입니다.
- `identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
- `client_context` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.

AWS Lambda 함수 로깅(Ruby)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그를 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 호출에 관한 세부 정보를 로그 스트림에 전송하며, 함수 코드에서 로그 및 그 외 출력을 중계합니다. 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 단원을 참조하십시오.

이 페이지에서는 AWS Command Line Interface, Lambda 콘솔 또는 CloudWatch 콘솔을 사용하여 Lambda 함수 코드의 로그 출력이나 액세스 로그를 생성하는 방법에 대해 설명합니다.

단원

- 로그를 반환하는 함수 생성
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제
- 로거 라이브러리

로그를 반환하는 함수 생성

함수 코드에서 로그를 출력하려는 경우, `puts` 명령문을 사용하거나 `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용할 수 있습니다. 다음 예제는 환경 변수의 값과 이벤트 객체를 로깅합니다.

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 로그 형식

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 줄을 로깅합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

자세한 로그를 보려면 the section called “로거 라이브러리”를 사용합니다.

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - `aws configure`을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
```

```
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 base64 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 my-function에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한

내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```

로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

로거 라이브러리

Ruby 로거 라이브러리는 쉽게 읽을 수 있는 간소화된 로그를 반환합니다. 로거 유틸리티를 사용하여 함수와 관련된 세부 정보, 메시지 및 오류 코드를 출력합니다.

```
# 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
```

logger의 출력에는 로그 레벨, 타임스탬프와 요청 ID가 포함됩니다.

```
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 Lambda에서 Ruby 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. 프런트엔드 API에서 백엔드의 스토리지 및 데이터베이스에 이르기까지 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다. 빌드 구성에 X-Ray SDK 라이브러리를 추가하기만 하면 함수가 AWS 서비스에 대해 수행하는 모든 호출에 대한 오류 및 지연 시간을 기록할 수 있습니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

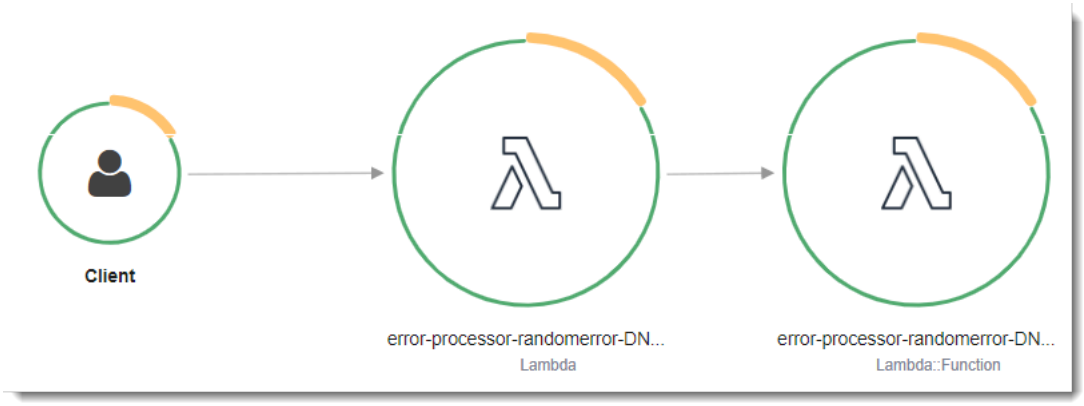
> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

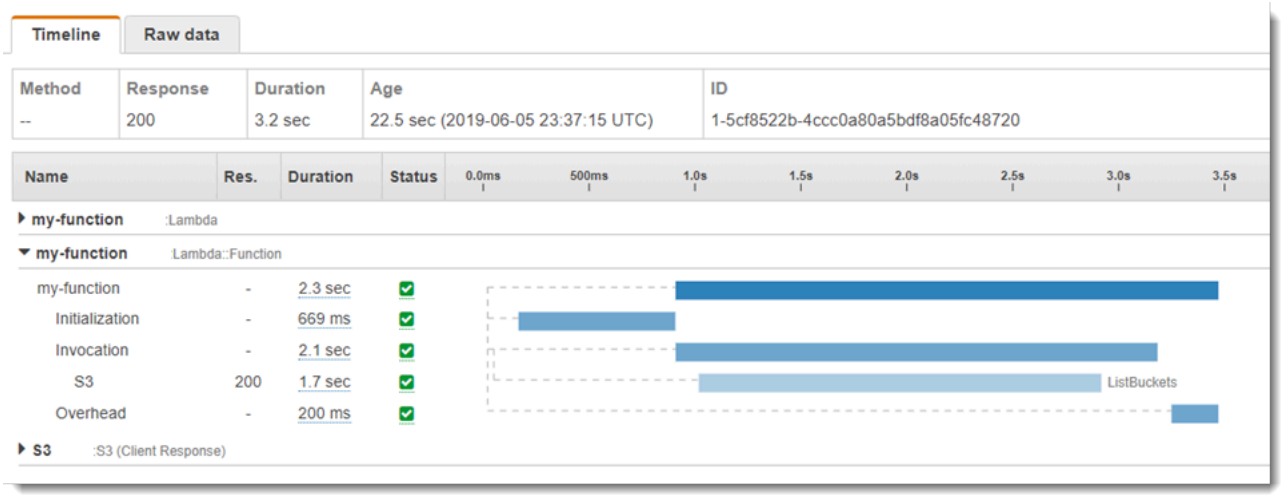
> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에

문제가 있는 것입니다. `AWS::Lambda::Function` 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 `AWS::Lambda::Function` 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

핸들러 코드를 계측하여 메타데이터를 기록하고 다운스트림 호출을 추적할 수 있습니다. 핸들러가 다른 리소스 및 서비스에 대해 수행하는 호출과 관련된 세부 정보를 기록하려면 X-Ray SDK for Ruby를 사용합니다. SDK를 가져오려면 애플리케이션의 종속성에 `aws-xray-sdk` 패키지를 추가합니다.

Example blank-ruby/function/Gemfile

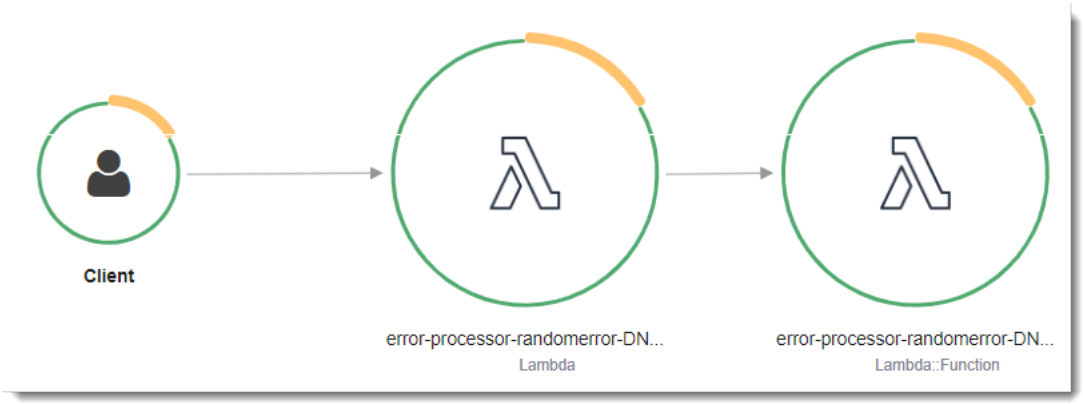
```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

AWS SDK 클라이언트를 계측하려면 초기화 코드에 클라이언트를 생성한 후 `aws-xray-sdk/lambda` 모듈이 필요합니다.

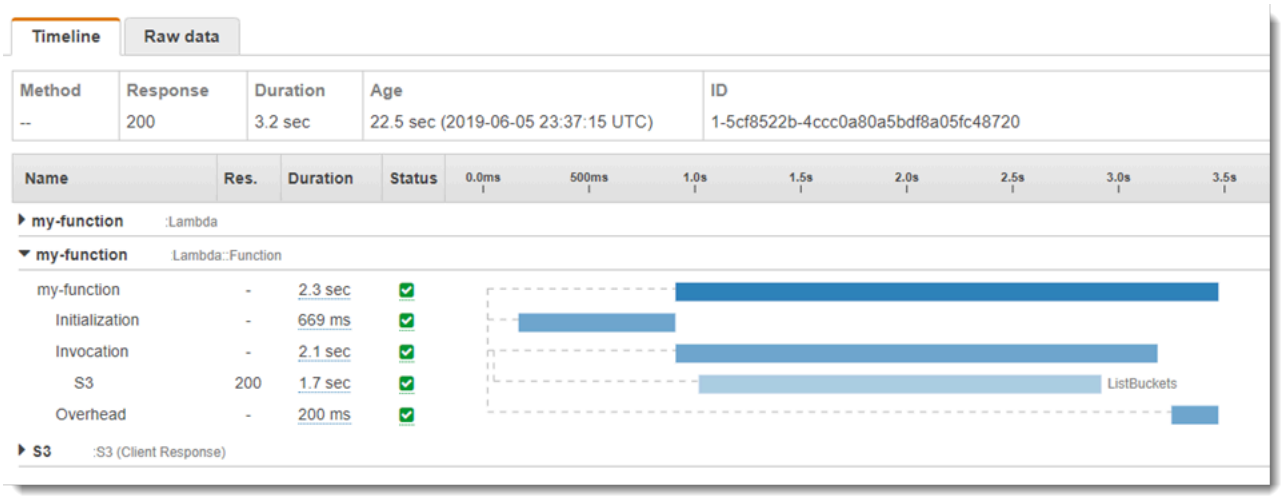
Example [blank-ruby/function/lambda_function.rb](#) – AWS SDK 클라이언트 추적

```
# 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-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. `AWS::Lambda::Function` 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 `AWS::Lambda::Function` 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 X-Ray SDK for Ruby를 참조하세요.

단원

- Lambda API로 활성 추적 사용
- AWS CloudFormation으로 활성 추적 활성화
- 계층에 런타임 종속성 저장

Lambda API로 활성 추적 사용

AWS CLI 또는 AWS SDK를 사용하여 추적 구성을 관리하려면 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation으로 활성 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

계층에 런타임 종속성 저장

X-Ray SDK를 사용하여 AWS SDK 클라이언트를 계측하는 경우 함수 코드와 배포 패키지가 상당히 커질 수 있습니다. 함수 코드를 업데이트할 때마다 런타임 종속성을 업로드하지 않으려면 X-Ray SDK를 Lambda 계층에 패키징합니다.

다음 예제에서는 X-Ray SDK for Ruby를 저장하는 `AWS::Serverless::LayerVersion` 리소스를 보여줍니다.

Example template.yml – 종속성 계층

```
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
```

이 구성을 사용하면 런타임 종속성을 변경하는 경우 라이브러리 계층만 업데이트하면 됩니다. 함수 배포 패키지에는 코드만 포함되어 있으므로 이는 업로드 시간을 줄일 수 있습니다.

종속성 계층을 만들려면 배포 전에 계층 아카이브를 생성하기 위해 빌드를 변경해야 합니다. 사용 가능한 예제는 blank-ruby 샘플 애플리케이션을 참조하세요.

Java를 사용하여 Lambda 함수 빌드

AWS Lambda에서 Java 코드를 실행할 수 있습니다. Lambda는 이벤트 처리를 위해 코드를 실행하는 Java를 위한 런타임을 제공합니다. 코드는 사용자가 관리하는 AWS Identity and Access Management(IAM) 역할의 AWS 자격 증명이 포함된 Amazon Linux 환경에서 실행됩니다.

Lambda는 다음과 같은 Java 런타임을 지원합니다.

Java

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
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			

Lambda는 Java 함수를 위한 다음 라이브러리를 제공합니다.

- com.amazonaws:aws-lambda-java-core(필수) – 핸들러 메서드 인터페이스와 런타임이 핸들러에 전달하는 컨텍스트 객체를 정의합니다. 고유한 입력 유형을 정의하는 경우 이 라이브러리만 필요합니다.
- com.amazonaws:aws-lambda-java-events – Lambda 함수를 호출하는 서비스의 이벤트 입력 유형입니다.
- com.amazonaws:aws-lambda-java-log4j2 – 현재 호출의 요청 ID를 함수 로그에 추가하는 데 사용할 수 있는 Apache Log4j 2용 appender 라이브러리입니다.
- AWS SDK for Java 2.0: Java 프로그래밍 언어용 공식 AWS SDK입니다.

> **Important**
>
> 프라이빗 필드, 메서드 또는 클래스와 같은 JDK API의 프라이빗 구성 요소를 사용하지 않습니다. 퍼블릭 API가 아닌 API 구성 요소는 변경되거나 업데이트 시 제거될 수 있으며, 이로 인해 애플리케이션이 중단될 수 있습니다.

Java 함수를 만들려면

1. Lambda 콘솔을 엽니다.
2. 함수 생성을 선택합니다.
3. 다음 설정을 구성합니다:
   - 함수 이름: 함수의 이름을 입력합니다.
   - 런타임: Java 21을 선택합니다.
4. 함수 생성을 선택합니다.
5. 테스트 이벤트를 구성하려면 테스트를 선택합니다.
6. 이벤트 이름에 **`test`**를 입력합니다.
7. Save changes(변경 사항 저장)를 선택합니다.
8. 함수를 호출하려면 테스트를 선택합니다.

콘솔은 `Hello`라는 핸들러 클래스를 사용하여 Lambda 함수를 만듭니다. Java는 컴파일된 언어이므로 Lambda 콘솔에서 소스 코드를 보거나 편집할 수 없지만 구성을 수정하고 호출하고 트리거를 구성할 수 있습니다.

> **Note**
>
> 로컬 환경에서 애플리케이션 개발을 시작하려면 이 가이드의 GitHub 리포지토리에서 사용할 수 있는 샘플 애플리케이션 중 하나를 배포하세요.

`Hello` 클래스에는 이벤트 객체와 컨텍스트 객체를 취하는 `handleRequest`라는 이름의 함수가 있습니다. 이는 함수가 호출될 때 Lambda가 호출하는 핸들러 함수입니다. Java 함수 런타임은 Lambda에서 호출 이벤트를 가져와 핸들러로 전달합니다. 함수 구성에서 핸들러 값은 `example.Hello::handleRequest`입니다.

함수의 코드를 업데이트하려면 함수 코드가 포함된 .zip 파일 아카이브인 배포 패키지를 작성합니다. 함수 개발이 진행되면 소스 제어에 함수 코드를 저장하고 라이브러리를 추가하며 배포를 자동화할 필요가 있습니다. 먼저 배포 패키지를 생성하고 명령줄에서 코드를 업데이트하세요.

함수 런타임은 호출 이벤트 외에도 컨텍스트 객체를 핸들러에 전달합니다. 컨텍스트 객체에는 호출, 함수 및 실행 환경에 관한 추가 정보가 포함되어 있습니다. 자세한 내용은 환경 변수에서 확인할 수 있습니다.

Lambda 함수는 CloudWatch Logs 로그 그룹을 함께 제공됩니다. 함수 런타임은 각 호출에 대한 세부 정보를 CloudWatch Logs에 보냅니다. 호출 중 함수가 출력하는 로그를 전달합니다. 함수가 오류를 반환하면 Lambda은 오류에 서식을 지정한 후 이를 호출자에게 반환합니다.

주제

- Java에서 Lambda 함수 핸들러 정의
- .zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포
- 컨테이너 이미지로 Java Lambda 함수 배포
- Java Lambda 함수를 위한 계층 작업
- Lambda SnapStart를 사용하여 시작 성능 개선
- Java Lambda 함수 사용자 지정 설정
- AWS Lambda 컨텍스트 객체(Java)
- AWS Lambda 함수 로깅(Java)
- AWS Lambda에서 Java 코드 계측
- AWS Lambda용 Java 샘플 애플리케이션

Java에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

이 안내서의 GitHub 리포지토리에서는 쉽게 배포할 수 있고 다양한 핸들러 유형을 보여 주는 샘플 애플리케이션을 제공합니다. 자세한 내용은 이 주제의 끝 부분을 참조하세요.

Sections

예제 핸들러: Java 17 런타임

다음 Java 17 예제에서 `HandlerIntegerJava17`이라는 클래스는 `handleRequest`라는 핸들러 메서드를 정의합니다. 핸들러 메서드는 다음과 같은 입력을 받습니다.

- 이벤트 데이터를 나타내는 사용자 지정 Java 레코드인 `IntegerRecord`입니다. 이 예제에서는 `IntegerRecord`를 다음과 같이 정의합니다.

```
record IntegerRecord(int x, int y, String message) {
}
```

- 간접 호출, 함수, 실행 환경에 대한 정보를 제공하는 메서드와 속성을 제공하는 컨텍스트 객체입니다.

입력 `IntegerRecord`에서 `message`를 기록하고 `x`와 `y`의 합을 반환하는 함수를 작성한다고 가정합니다. 다음은 함수 코드입니다.

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) {
}
```

함수 구성에서 핸들러 파라미터를 설정하여 Lambda가 간접적으로 호출할 메서드를 지정합니다. 핸들러를 다음과 같은 형식으로 표현할 수 있습니다.

- *package*.*Class*::*method* – 전체 형식. 예: `example.Handler::handleRequest`.
- *package*.*Class* – 핸들러 인터페이스를 구현하는 클래스에 대한 단축 형식입니다. 예: `example.Handler`.

Lambda가 핸들러를 간접적으로 호출하면 Lambda 런타임은 이벤트를 JSON 형식 문자열로 수신하여 객체로 변환합니다. 이전 예제에서 샘플 이벤트는 다음과 같을 수 있습니다.

Example event.json

```
{
  "x": 1,
  "y": 20,
  "message": "Hello World!"
}
```

이 파일을 저장하고 다음 AWS Command Line Interface(CLI) 명령을 사용하여 로컬에서 함수를 테스트할 수 있습니다.

```
aws lambda invoke --function-name function_name --payload file://event.json out.json
```

예제 핸들러: Java 11 런타임 이하

Lambda는 Java 17 런타임 및 그 이상에서의 레코드만 지원합니다. 모든 Java 런타임에서 클래스를 사용하여 이벤트 데이터를 나타낼 수 있습니다. 다음 예제에서는 정수 목록과 컨텍스트 객체를 입력으로 사용하고 목록에 있는 모든 정수의 합계를 반환합니다.

Example Handler.java

다음 예제에서 `Handler`라는 클래스는 `handleRequest`라는 핸들러 메서드를 정의합니다. 핸들러 메서드는 이벤트 및 컨텍스트 객체를 입력으로 받아 문자열을 반환합니다.

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();
  }
}
```

더 많은 예제는 샘플 핸들러 코드를 참조하세요.

초기화 코드

Lambda는 함수를 처음 호출하기 전 초기화 단계에서 정적 코드와 클래스 생성자를 실행합니다. 초기화 중에 생성된 리소스는 간접 호출 간에 메모리에 남아 있으며 핸들러에서 수천 번 재사용할 수 있습니다. 따라서 기본 핸들러 메서드 외부에 초기화 코드를 추가하여 컴퓨팅 시간을 절약하고 여러 간접 호출에서 리소스를 재사용할 수 있습니다.

다음 예제에서 클라이언트 초기화 코드는 기본 핸들러 메서드 외부에 있습니다. 런타임은 함수가 클라이언트의 첫 번째 이벤트를 제공하기 전에 클라이언트를 초기화합니다. Lambda가 클라이언트를 다시 초기화할 필요가 없으므로 후속 이벤트는 훨씬 더 빠릅니다.

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";
    }
}
```

입력 및 출력 유형 선택

핸들러 메서드의 서명에서 이벤트가 매핑되는 객체 유형을 지정합니다. 앞의 예제에서 Java 런타임은 이벤트를 `Map<String,String>` 인터페이스를 구현하는 유형으로 역직렬화합니다. 문자열 간 맵은 다음과 같은 플랫 이벤트에서 작동합니다.

Example Event.json – 날씨 데이터

```
{
  "temperatureK": 281,
  "windKmh": -3,
  "humidityPct": 0.55,
  "pressureHPa": 1020
}
```

그러나 각 필드의 값은 문자열이나 숫자여야 합니다. 이벤트에 객체 값이 있는 필드가 포함되어 있으면 런타임에서 이벤트를 역직렬화할 수 없으며 오류를 반환합니다.

함수가 처리하는 이벤트 데이터에서 작동하는 입력 유형을 선택합니다. 기본 형식, 제네릭 형식 또는 적절히 정의된 형식을 사용할 수 있습니다.

입력 유형

- `Integer`, `Long`, `Double` 등. - 이벤트가 추가 형식이 없는 숫자입니다(예: `3.5`). 런타임에서 값을 지정된 유형의 객체로 변환합니다.
- `String` – 이벤트가 인용 부호를 포함한 JSON 문자열입니다(예: `"My string."`). 런타임에서 값을 인용 부호 없이 `String` 객체로 변환합니다.
- *`Type`*, `Map<String,`*`Type`*`>` 등. - 이벤트가 JSON 객체입니다. 런타임에서 이벤트를 지정된 유형 또는 인터페이스의 객체로 역직렬화합니다.
- `List<Integer>`, `List<String>`, `List<Object>` 등. - 이벤트가 JSON 배열입니다. 런타임에서 이벤트를 지정된 유형 또는 인터페이스의 객체로 역직렬화합니다.
- `InputStream` – 이벤트가 임의의 JSON 유형입니다. 런타임에서 문서의 바이트 스트림을 수정하지 않고 핸들러에 전달합니다. 사용자가 입력을 역직렬화하고 출력 스트림에 출력을 써야 합니다.

- 라이브러리 유형 – AWS 서비스에서 전송하는 이벤트의 경우 aws-lambda-java-events 라이브러리에 있는 유형을 사용합니다.

고유한 입력 유형을 정의하는 경우 이벤트의 각 필드에 기본 생성자와 속성을 사용하여 역직렬화 가능하고 변경 가능한 POJO(Plain Old Java Object) 유형이어야 합니다. 이벤트에 포함되지 않은 속성과 속성에 매핑되지 않은 이벤트의 키는 오류 없이 삭제됩니다.

출력 유형은 객체 또는 void가 될 수 있습니다. 런타임은 반환 값을 텍스트로 직렬화합니다. 출력이 필드가 포함된 객체인 경우 런타임에서 이를 JSON 문서로 직렬화합니다. 프리미티브 값을 래핑하는 유형인 경우 런타임이 해당 값의 텍스트 표현을 반환합니다.

핸들러 인터페이스

aws-lambda-java-core 라이브러리는 핸들러 메서드에 대한 두 가지 인터페이스를 정의합니다. 제공된 인터페이스를 사용하여 핸들러 구성을 간소화하고 컴파일 타임에 핸들러 메서드 서명의 유효성을 검사하세요.

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

RequestHandler 인터페이스는 입력 유형과 출력 유형의 두 가지 파라미터를 취하는 제네릭 형식입니다. 두 유형 모두 객체여야 합니다. 이 인터페이스를 사용하면 Java 런타임에서 이벤트를 입력 유형의 객체로 역직렬화하고 출력을 텍스트로 직렬화합니다. 해당 입력 및 출력 유형에 대해 기본 제공 직렬화가 작동하는 경우 이 인터페이스를 사용하세요.

Example Handler.java – 핸들러 인터페이스

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, String>{
  @Override
  public String handleRequest(Map<String,String> event, Context context)
```

고유한 직렬화를 사용하려면 RequestStreamHandler 인터페이스를 구현합니다. 이 인터페이스를 사용하면 Lambda에서 핸들러에 입력 스트림과 출력 스트림을 전달합니다. 핸들러는 입력 스트림에서 바이트를 읽어 출력 스트림에 쓰고 void를 반환합니다.

다음 예제에서는 입력 및 출력 스트림 작업을 위해 버퍼링된 리더 및 라이터 유형을 사용합니다.

Example HandlerStream.java

```
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();
    }
  }
}
```

샘플 핸들러 코드

이 안내서의 GitHub 리포지토리에는 다양한 핸들러 유형 및 인터페이스의 사용을 보여주는 샘플 애플리케이션이 들어 있습니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS SAM 템플릿 및 지원 리소스가 포함되어 있습니다.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.
- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

`java-events` 및 `s3-java` 애플리케이션은 AWS 서비스 이벤트를 입력으로 받아 문자열을 반환합니다. `java-basic` 애플리케이션에는 여러 유형의 핸들러가 포함되어 있습니다.

- Handler.java – `Map<String,String>`을 입력으로 받습니다.
- HandlerInteger.java – `Integer`를 입력으로 받습니다.
- HandlerList.java – `List<Integer>`를 입력으로 받습니다.
- HandlerStream.java – `InputStream` 및 `OutputStream`을 입력으로 받습니다.
- HandlerString.java – `String`을 입력으로 받습니다.
- HandlerWeatherData.java – 사용자 지정 유형을 입력으로 받습니다.

다른 핸들러 유형을 테스트하려면 AWS SAM 템플릿에서 핸들러 값을 변경합니다. 자세한 지침은 샘플 애플리케이션의 Readme 파일을 참조하세요.

.zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포

AWS Lambda 함수의 코드는 스크립트 또는 컴파일된 프로그램과 해당 종속 항목으로 구성됩니다. 함수 코드는 배포 패키지를 사용하여 Lambda에 배포합니다. Lambda는 컨테이너 이미지 및 .zip 파일 아카이브의 두 가지 배포 패키지를 지원합니다.

이 페이지에서는 배포 패키지를 .zip 파일 또는 Jar 파일로 생성한 후 해당 배포 패키지를 사용하여 AWS Lambda(AWS Command Line Interface)을(를) 사용하는 AWS CLI에 함수 코드를 배포하는 방법을 설명합니다.

Sections

사전 조건

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - `aws configure`을 통한 빠른 구성

도구 및 라이브러리

Lambda는 Java 함수를 위한 다음 라이브러리를 제공합니다.

- com.amazonaws:aws-lambda-java-core(필수) – 핸들러 메서드 인터페이스와 런타임이 핸들러에 전달하는 컨텍스트 객체를 정의합니다. 고유한 입력 유형을 정의하는 경우 이 라이브러리만 필요합니다.

- com.amazonaws:aws-lambda-java-events – Lambda 함수를 호출하는 서비스의 이벤트 입력 유형입니다.
- com.amazonaws:aws-lambda-java-log4j2 – 현재 호출의 요청 ID를 함수 로그에 추가하는 데 사용할 수 있는 Apache Log4j 2용 appender 라이브러리입니다.
- AWS SDK for Java 2.0: Java 프로그래밍 언어용 공식 AWS SDK입니다.

이 라이브러리는 Maven Central Repository를 통해 사용할 수 있습니다. 다음과 같이 빌드 정의에 이러한 라이브러리를 추가하세요.

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>
```

배포 패키지를 만들려면 함수 코드와 종속 항목을 단일 .zip 파일 또는 Java 아카이브(JAR) 파일로 컴파일합니다. Gradle의 경우 `Zip` 빌드 유형을 사용합니다. Apache Maven의 경우 Maven Shade 플러

그인을 사용합니다. 배포 패키지를 업로드하려면 Lambda 콘솔, Lambda API 또는 AWS Serverless Application Model(AWS SAM)을 사용합니다.

> **Note**
>
> 배포 패키지 크기를 작게 유지하려면 함수의 계층별 종속성을 패키지화하세요. 계층을 사용하면 독립적으로 종속성을 관리하고, 여러 함수에서 사용할 수 있으며 다른 계정과 공유할 수 있습니다. 자세한 내용은 Lambda 계층 단원을 참조하십시오.

Gradle을 사용하여 배포 패키지 빌드

Gradle에서 함수의 코드와 종속 구성 요소를 사용하여 배포 패키지를 생성하려면 `Zip` 빌드 유형을 사용합니다. 다음은 전체 샘플 build.gradle 파일의 예제입니다.

Example build.gradle – 빌드 작업

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

이 빌드 구성은 `build/distributions` 디렉터리에 배포 패키지를 생성합니다. `into('lib')` 문 내에서 `jar` 작업은 기본 클래스를 포함하는 jar 아카이브를 `lib`라는 폴더에 어셈블합니다. 또한 `configurations.runtimeClassPath` 작업은 빌드의 클래스 경로에서 동일한 `lib` 폴더로 종속 항목 라이브러리를 복사합니다.

Example build.gradle – 종속성

```
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'
    ...
```

```
}
```

Lambda는 JAR 파일을 유니코드 알파벳순으로 로드합니다. `lib` 디렉터리의 여러 JAR 파일에 동일한 클래스가 있을 경우 첫 번째 클래스가 사용됩니다. 다음 셸 스크립트를 사용하여 중복 클래스를 식별할 수 있습니다.

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
```

종속 항목을 위한 Java 계층 생성

> **Note**
>
> 계층을 Java와 같은 컴파일된 언어의 함수와 함께 사용하면 Python과 같은 해석된 언어의 함수와 함께 사용할 때와 같은 이점을 얻지 못할 수 있습니다. Java는 컴파일된 언어이므로 함수가 초기화 단계에서 수동으로 공유 어셈블리를 메모리로 로드해야 하기 때문에 콜드 시간 시간이 늘어날 수 있습니다. 대신 컴파일 시간에 모든 공유 코드를 포함하여 기본 제공 최적화를 활용하는 것이 좋습니다.

이 섹션의 지침은 계층에 종속 항목을 포함하는 방법을 보여줍니다. 배포 패키지에 종속 항목을 포함하는 방법에 대한 지침은 the section called “Gradle을 사용하여 배포 패키지 빌드” 또는 the section called “Maven을 사용하여 배포 패키지 빌드” 섹션을 참조하세요.

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 `/opt` 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 `PATH` 변수에는 `/opt` 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. `PATH` 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

- `java/lib` (`CLASSPATH`)

예를 들어, 계층.zip 파일 구조는 다음과 같을 수 있습니다.

```
jackson.zip
```

```
# java/lib/jackson-core-2.2.3.jar
```

또한 Lambda는 /opt/lib 디렉터리의 모든 라이브러리와 /opt/bin 디렉터리의 모든 바이너리를 자동으로 감지합니다. Lambda가 계층 콘텐츠를 제대로 찾을 수 있도록 다음 구조로 계층을 생성할 수도 있습니다.

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

계층을 패키징한 후 the section called “계층 생성 및 삭제” 및 the section called “계층 추가”을 참조하여 계층 설정을 완료합니다.

Maven을 사용하여 배포 패키지 빌드

Maven을 사용하여 배포 패키지를 빌드하려면 Maven Shade 플러그인을 사용합니다. 이 플러그인은 컴파일된 함수 코드와 모든 종속 항목이 포함된 JAR 파일을 생성합니다.

Example pom.xml – 플러그인 구성

```
      <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>
```

배포 패키지를 빌드하려면 mvn package 명령을 사용합니다.

```
[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] ------------------------------------------------------------------------
```

이 명령은 `target` 디렉터리에 JAR 파일을 생성합니다.

> **Note**
>
> 다중 릴리스 JAR(MRJAR)로 작업하는 경우 배포 패키지를 Lambda에 업로드하기 전에 `lib` 디렉터리에 MRJAR(즉, Maven Shade 플러그인에서 생성된 음영 처리된 JAR)을 포함하고 압축해야 합니다. 그렇지 않으면 Lambda가 JAR 파일의 압축을 제대로 풀지 못해 `MANIFEST.MF` 파일이 무시될 수 있습니다.

appender 라이브러리(`aws-lambda-java-log4j2`)를 사용하는 경우 Maven Shade 플러그인에 대한 변환기도 구성해야 합니다. 변환기 라이브러리는 appender 라이브러리와 Log4j에 모두 나타나는 캐시 파일 버전을 결합합니다.

Example pom.xml – Log4j 2 appender를 사용하여 플러그인 구성

```
      <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 콘솔을 사용하여 배포 패키지 업로드

새 함수를 생성하려면 먼저 콘솔에서 함수를 생성한 다음.zip 또는 JAR 파일을 업로드해야 합니다. 기존 함수를 업데이트하려면 함수에 대한 페이지를 연 다음 동일한 절차에 따라 업데이트된 .zip 또는 JAR 파일을 추가합니다.

배포 패키지 파일이 50MB 미만인 경우 로컬 컴퓨터에서 직접 파일을 업로드하여 함수를 생성하거나 업데이트할 수 있습니다. 50MB보다 큰 .zip 또는 JAR 파일의 경우 먼저 패키지를 Amazon S3 버킷에 업로드해야 합니다. AWS Management Console을 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 Amazon S3 시작하기를 참조하세요. AWS CLI를 사용하여 파일을 업로드하려면 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> 기존 함수의 배포 패키지 유형(.zip 또는 컨테이너 이미지)은 변경할 수 없습니다. 예를 들어 .zip 파일 아카이브를 사용하도록 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

새 함수 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.
3. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름에 함수 이름을 입력합니다.
   b. 런타임에서 사용할 런타임을 선택합니다.
   c. (선택 사항) 아키텍처에서 함수에 대한 명령 세트 아키텍처를 선택합니다. 기본 아키텍처는 x86_64입니다. 함수에 대한 .zip 배포 패키지가 선택한 명령 세트 아키텍처와 호환되는지 확인합니다.
4. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 실행 역할을 사용할 수 있습니다.
5. 함수 생성을 선택합니다. Lambda에서 선택한 런타임을 사용하여 기본 'Hello World' 함수를 생성합니다.

로컬 시스템에서 .zip 또는 JAR 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서.zip 또는 JAR 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. .zip 또는.jar 파일을 선택합니다.
5. .zip 또는 JAR 파일을 업로드하려면 다음을 수행합니다.
   a. 업로드를 선택한 다음 파일 선택기에서.zip 또는 JAR 파일을 선택합니다.
   b. Open을 선택합니다.
   c. Save(저장)를 선택합니다.

Amazon S3 버킷에서.zip 또는 JAR 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서 새 .zip 또는 JAR 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. Amazon S3 위치를 선택합니다.
5. .zip 파일의 Amazon S3 링크 URL을 붙여 넣고 저장을 선택합니다.

AWS CLI를 사용하여 배포 패키지 업로드

AWS CLI를 사용하여 새 함수를 생성하거나.zip 또는 JAR 파일로 기존 함수를 업데이트할 수 있습니다. create-function 및 update-function-code 명령을 사용하여 .zip 또는 JAR 패키지를 배포합니다. 파일이 50MB보다 작은 경우 로컬 빌드 시스템의 파일 위치에서 패키지를 업로드할 수 있습니다. 더 큰 파일의 경우 Amazon S3 버킷에서 .zip 또는 JAR 패키지를 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> AWS CLI를 사용하여 Amazon S3 버킷에서 .zip 또는 JAR 파일을 업로드하는 경우 버킷은 함수와 동일한 AWS 리전에 있어야 합니다.

AWS CLI에서 .zip 또는 JAR 파일을 사용하여 새 함수를 생성하려면 다음을 지정해야 합니다.

- 함수의 이름(`--function-name`)
- 함수의 런타임(`--runtime`)
- 함수의 실행 역할(`--role`)의 Amazon 리소스 이름(ARN)
- 함수 코드에 있는 핸들러 메서드의 이름(`--handler`)

.zip 또는 JAR 파일의 위치도 지정해야 합니다. .zip 또는 JAR 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
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 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--code` 옵션을 사용합니다. 버전이 지정된 객체에만 `S3ObjectVersion` 파라미터를 사용해야 합니다.

```
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
```

CLI를 사용하여 기존 함수를 업데이트하려면 `--function-name` 파라미터를 사용하여 함수 이름을 지정합니다. 함수 코드를 업데이트하는 데 사용할.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--s3-bucket` 및 `--s3-key` 옵션을 사용합니다. 버전이 지정된 객체에만 `--s3-object-version` 파라미터를 사용해야 합니다.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

AWS SAM을 사용하여 배포 패키지 업로드

AWS SAM을 사용하여 함수 코드, 구성 및 종속 항목의 배포를 자동화할 수 있습니다. AWS SAM은 서버리스 애플리케이션을 정의하기 위한 단순화된 구문을 제공하는 AWS CloudFormation의 익스텐션입니다. 다음 예제 템플릿은 Gradle이 사용하는 `build/distributions` 디렉터리의 배포 패키지로 함수를 정의합니다.

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
```

함수를 만들려면 package 및 deploy 명령을 사용합니다. 이러한 명령은 AWS CLI에 대한 사용자 지정 명령으로, 다른 명령을 래핑하여 배포 패키지를 Amazon S3에 업로드하고 객체 URI로 템플릿을 다시 작성하며 함수의 코드를 업데이트합니다.

다음 예제 스크립트는 Gradle 빌드를 실행하고 생성된 배포 패키지를 업로드합니다. 이 스크립트를 처음 실행하면 AWS CloudFormation 스택이 생성되며, 스택이 이미 있으면 스크립트를 통해 스택이 업데이트됩니다.

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
```

전체 예제를 보려면 다음 샘플 애플리케이션을 참조하세요.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의

aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.

- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

컨테이너 이미지로 Java Lambda 함수 배포

Java Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- Java용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.

- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Java용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 Java용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

주제

- AWSJava용 기본 이미지
- Java용 AWS 기본 이미지 사용
- 런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

AWSJava용 기본 이미지

AWS는 Java에 대한 다음과 같은 기본 이미지를 제공합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
21	Java 21	Amazon Linux 2023	GitHub의 Java 21용 Dockerfile	
17	Java 17	Amazon Linux 2	GitHub의 Java 17용 Dockerfile	
11	Java 11	Amazon Linux 2	GitHub의 Java 11용 Dockerfile	
8.al2	Java 8	Amazon Linux 2	GitHub의 Java 8용 Dockerfile	

Amazon ECR 리포지토리: gallery.ecr.aws/lambda/java

Java 21 이상의 기본 이미지는 Amazon Linux 2023 최소 컨테이너 이미지를 기반으로 합니다. 이전 기본 이미지는 Amazon Linux 2를 사용합니다. AL2023은 작은 배포 공간과 `glibc`와 같이 업데이트된 라이브러리 버전을 포함하여 Amazon Linux 2에 비해 여러 가지 이점을 제공합니다.

AL2023 기반 이미지는 `microdnf`(`dnf` 심볼릭 링크)를 Amazon Linux 2에서 기본 패키지 관리자인 `yum` 대신 패키지 관리자로 사용합니다. `microdnf`는 `dnf`의 독립 실행형 구현입니다. AL2023 기반 이미지에 포함된 패키지 목록의 경우 Comparing packages installed on Amazon Linux 2023 Container Images의 Minimal Container 열을 참조하세요. AL2023과 Amazon Linux 2의 차이점에 대한 자세한 내용은 AWS 컴퓨팅 블로그의 Introducing the Amazon Linux 2023 runtime for AWS Lambda를 참조하세요.

> **Note**
>
> AWS Serverless Application Model(AWS SAM)을 포함하여 AL2023 기반 이미지를 로컬에서 실행하려면 Docker 버전 20.10.10 이상을 사용해야 합니다.

Java용 AWS 기본 이미지 사용

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Java(예: Amazon Corretto)
- Docker(Java 21 이상 기본 이미지의 경우 최소 버전 20.10.10)
- Apache Maven 또는 Gradle
- AWS Command Line Interface(AWS CLI) 버전 2

기본 이미지에서 이미지 생성

Maven

1. 다음 명령을 실행하여 Lambda의 아키타이프를 사용하는 Maven 프로젝트를 생성합니다. 다음 파라미터는 필수 파라미터입니다.

   - 서비스 - Lambda 함수에서 사용할 AWS 서비스 클라이언트입니다. 사용 가능한 소스 목록은 GitHub의 aws-sdk-java-v2/services를 참조하세요.
   - 리전 – Lambda 함수를 생성하려는 AWS 리전입니다.
   - groupId - 애플리케이션의 전체 패키지 네임스페이스입니다.
   - artifactId - 프로젝트 이름입니다. 이는 프로젝트의 디렉터리 이름이 됩니다.

   Linux 및 macOS에서 실행하는 명령:

   ```
   mvn -B archetype:generate \
      -DarchetypeGroupId=software.amazon.awssdk \
      -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
      -DgroupId=com.example.myapp \
      -DartifactId=myapp
   ```

   Powershell에서 실행하는 명령:

   ```
   mvn -B archetype:generate `
      "-DarchetypeGroupId=software.amazon.awssdk" `
      "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
    `
   ```

```
    "-DgroupId=com.example.myapp" `
    "-DartifactId=myapp"
```

Lambda의 Maven 아키타이프는 Java SE 8로 컴파일되도록 미리 구성되어 있으며 AWS SDK for Java에 대한 종속 구성 요소를 포함합니다. 다른 아키타이프 또는 다른 메서드를 사용하여 프로젝트를 생성하는 경우 Maven용 Java 컴파일러를 구성하고 SDK를 종속 구성 요소로 선언해야 합니다.

2. *myapp*/src/main/java/com/example/*myapp* 디렉터리를 열고 App.java 파일을 찾습니다. Lambda 함수에 대한 코드입니다. 제공된 샘플 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.
3. 프로젝트의 루트 디렉터리로 다시 이동하고 다음 구성을 사용하여 새 Dockerfile을 생성합니다.
   - FROM 속성을 기본 이미지의 URI로 설정합니다.
   - CMD 인수를 Lambda 함수 핸들러로 설정합니다.

   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. 프로젝트를 컴파일하고 런타임 종속 구성 요소를 수집합니다.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. docker build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

Gradle

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir example
cd example
```

2. 다음 명령을 실행하여 Gradle이 환경의 `example` 디렉터리에 새 Java 애플리케이션 프로젝트를 생성하도록 합니다. 빌드 스크립트 DSL 선택의 경우 2: Groovy를 선택합니다.

```
gradle init --type java-application
```

3. */example*/app/src/main/java/*example* 디렉터리를 열고 `App.java` 파일을 찾습니다. Lambda 함수에 대한 코드입니다. 다음 샘플 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.

   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` 파일을 엽니다. 이전 단계의 샘플 함수 코드를 사용하는 경우 `build.gradle`의 내용을 다음으로 바꿉니다. 자체 함수 코드를 사용하는 경우 필요에 따라 `build.gradle` 파일을 수정합니다.

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. 2단계의 `gradle init` 명령은 `app/test` 디렉터리에 더미 테스트 사례도 생성했습니다. 이 자습서에서는 `/test` 디렉터리를 삭제하여 테스트 실행을 건너뜁니다.
6. 프로젝트를 빌드합니다.

```
gradle build
```

7. 프로젝트의 루트 디렉터리(`/example`)에서 다음 구성으로 Dockerfile을 생성합니다.

   - `FROM` 속성을 기본 이미지의 URI로 설정합니다.
   - `COPY` 명령을 사용하여 함수 코드와 런타임 종속성을 Lambda 정의 환경 변수인 `{LAMBDA_TASK_ROOT}`에 복사합니다.
   - `CMD` 인수를 Lambda 함수 핸들러로 설정합니다.

   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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 --platform linux/amd64 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 --platform linux/arm64 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 이 예제에서 docker-image는 이미지 이름이고 test는 태그입니다.

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

이 명령은 이미지를 컨테이너로 실행하고 localhost:9000/2015-03-31/functions/function/invocations에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 --platform linux/*arm64* 옵션을 --platform linux/*amd64* 대신 사용해야 합니다.

2. 새 터미널 창에서 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 curl 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. 컨테이너 ID를 가져옵니다.

```
docker ps
```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.

   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.

- 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:

- `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
- `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API을 확장합니다.

Java용 런타임 인터페이스 클라이언트를 Docker 파일에 설치하거나 프로젝트의 종속 항목으로 설치합니다. 예를 들어, Maven 패키지 관리자를 사용하여 런타임 인터페이스 클라이언트를 설치하려면 `pom.xml` 파일에 다음을 추가하세요.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

패키지에 대한 자세한 내용은 Maven Central Repository의 AWS Lambda Java 런타임 인터페이스 클라이언트를 참조하세요. AWS Lambda Java 지원 라이브러리 GitHub 리포지토리에서 런타임 인터페이스 클라이언트 소스 코드를 검토할 수도 있습니다.

다음 예제에서는 Amazon Corretto 이미지를 사용하여 Java용 컨테이너 이미지를 빌드하는 방법을 보여줍니다. Amazon Corretto는 무료로 사용할 수 있는 Open Java Development Kit(OpenJDK)의 프로

덕션용 멀티플랫폼 배포판입니다. Maven 프로젝트에는 런타임 인터페이스 클라이언트가 종속 항목으로 포함되어 있습니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Java(예: Amazon Corretto)
- Docker
- Apache Maven
- AWS Command Line Interface(AWS CLI) 버전 2

대체 기본 이미지에서 이미지 생성

1. Maven 프로젝트를 생성합니다. 다음 파라미터는 필수 파라미터입니다.

   - groupId - 애플리케이션의 전체 패키지 네임스페이스입니다.
   - artifactId - 프로젝트 이름입니다. 이는 프로젝트의 디렉터리 이름이 됩니다.

   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. 프로젝트 디렉터리를 엽니다.

   ```
   cd myapp
   ```

3. pom.xml 파일을 열고 내용을 다음으로 바꿉니다. 이 파일에는 aws-lambda-java-runtime-interface-client가 종속 항목으로 포함되어 있습니다. 또는 Dockerfile에 런타임 인터페이스 클라이언트를 설치할 수 있습니다. 그러나 가장 간단한 방법은 라이브러리를 종속 항목으로 포함하는 것입니다.

```
<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* 디렉터리를 열고 App.java 파일을 찾습니다. Lambda 함수에 대한 코드입니다. 코드를 다음으로 바꿉니다.

   Example 함수 핸들러

   ```
   package example;

   public class App {
       public static String sayHello() {
           return "Hello world!";
       }
   }
   ```

5. 1단계의 mvn -B archetype:generate 명령은 src/test 디렉터리에 더미 테스트 케이스도 생성했습니다. 본 자습서의 목적에 맞게 생성된 전체 /test 디렉터리를 삭제하여 실행 중인 테스트를 건너뛰는 것입니다.

6. 프로젝트의 루트 디렉터리로 다시 이동하고 새 Dockerfile을 생성합니다. 다음 예제 Dockerfile에서는 Amazon Corretto 이미지를 사용합니다. Amazon Corretto는 OpenJDK의 프로덕션용 무료 멀티플랫폼 배포판입니다.

   - FROM 속성을 기본 이미지의 URI로 설정합니다.
   - Docker 컨테이너가 시작될 때 실행할 모듈로 ENTRYPOINT를 설정합니다. 이 경우 모듈은 런타임 인터페이스 클라이언트입니다.
   - CMD 인수를 Lambda 함수 핸들러로 설정합니다.

   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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. 에뮬레이터를 이미지에 빌드하거나 다음 절차를 사용하여 로컬 시스템에 설치할 수 있습니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 설치 및 실행

1. 프로젝트 디렉터리에서 다음 명령을 실행하여 GitHub에서 런타임 인터페이스 에뮬레이터(x86-64 아키텍처)를 다운로드하고 로컬 시스템에 설치합니다.

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 에뮬레이터를 설치하려면 이전 명령의 GitHub 리포지토리 URL을 다음과 같이 바꿉니다.

```
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 에뮬레이터를 설치하려면 `$downloadLink`을(를) 다음과 같이 바꿉니다.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello`는 Docker 파일의 `CMD` 다음에 오는 `ENTRYPOINT`입니다.

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
```

이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

> **Note**
>
> ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

3. 로컬 엔드포인트에 이벤트를 게시합니다.

Linux/macOS

Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 컨테이너 ID를 가져옵니다.

```
docker ps
```

5. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

```
  "StatusCode": 200
}
```

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

Java Lambda 함수를 위한 계층 작업

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다. 계층을 생성하려면 세 가지 일반적인 단계를 거칩니다.

1. 계층 콘텐츠를 패키징합니다. 즉, 함수에 사용하려는 종속성이 포함된 .zip 파일 아카이브를 생성합니다.
2. Lambda에서 계층을 생성합니다.
3. 계층을 함수에 추가합니다.

이 주제에는 외부 라이브러리 종속성이 있는 Java Lambda 계층을 올바르게 패키징하고 생성하는 방법에 대한 단계와 지침이 포함되어 있습니다.

주제

필수 조건

이 섹션의 단계를 수행하려면 다음이 필요합니다.

- Java 21
- Apache Maven 3.8.6 이상
- AWS Command Line Interface(AWS CLI) 버전 2

> **Note**
>
> Maven이 참조하는 Java 버전이 배포하려는 함수의 Java 버전과 동일한지 확인합니다. 예를 들어 Java 21 함수의 경우에 `mvn -v` 명령은 출력에 Java 버전 21을 나열해야 합니다.

```
Apache Maven 3.8.6
...
Java version: 21.0.2, vendor: Oracle Corporation, runtime: /Library/Java/
JavaVirtualMachines/jdk-21.jdk/Contents/Home
...
```

이 주제에서는 awsdocs GitHub 리포지토리에 있는 layer-java 샘플 애플리케이션을 참조합니다. 이 애플리케이션은 종속성을 다운로드하고 계층을 생성하는 스크립트를 포함합니다. 애플리케이션은 계층의 종속성을 사용하는 해당 함수도 포함합니다. 계층을 생성한 후 해당 함수를 배포하고 호출하여 모든 것이 제대로 작동하는지 확인할 수 있습니다. 함수에 Java 21 런타임을 사용하므로 계층도 Java 21과 호환되어야 합니다.

layer-java 샘플 애플리케이션에는 두 개의 하위 디렉터리 내에 하나의 예제가 포함되어 있습니다. layer 디렉터리에는 계층 종속성을 정의하는 pom.xml 파일과 계층을 생성하기 위한 스크립트가 포함되어 있습니다. function 디렉터리는 계층이 작동하는지 테스트하는 데 도움이 되는 샘플 함수를 포함합니다. 이 자습서는 이 계층을 생성하고 패키징하는 방법을 안내합니다.

Amazon Linux와의 Java 계층 호환성

계층을 생성하는 첫 번째 단계는 모든 계층 콘텐츠를.zip 파일 아카이브로 번들링하는 것입니다. Lambda 함수는 Amazon Linux에서 실행되기 때문에 계층 콘텐츠는 Linux 환경에서 컴파일하고 빌드할 수 있어야 합니다.

Java 코드는 플랫폼에 구애받지 않도록 설계되었으므로 Linux 환경을 사용하지 않더라도 로컬 시스템에서 계층을 패키징할 수 있습니다. Lambda에 Java 계층을 업로드한 후에도 여전히 Amazon Linux와 호환됩니다.

Java 런타임의 계층 경로

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 /opt 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 PATH 변수에는 /opt 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. PATH 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

- java/lib

예를 들어 이 자습서에서 생성하는 결과 레이어 .zip 파일의 디렉터리 구조는 다음과 같습니다.

```
layer_content.zip
# java
    # lib
        # layer-java-layer-1.0-SNAPSHOT.jar
```

`layer-java-layer-1.0-SNAPSHOT.jar` JAR 파일(필요한 모든 종속성을 포함하는 uber-jar)이 `java/lib` 디렉터리에 올바르게 위치해 있습니다. 이렇게 하면 함수 호출 중에 Lambda가 라이브러리를 찾을 수 있습니다.

계층 콘텐츠의 패키징

이 예제에서는 다음 2개의 Java 라이브러리를 하나의 JAR 파일로 패키징합니다.

- aws-lambda-java-core - AWS Lambda에서 Java로 작업하기 위한 최소한의 인터페이스 정의 집합입니다.
- Jackson - 특히 JSON 작업에 널리 사용되는 데이터 처리 도구 모음입니다.

다음 단계를 완료하여 계층 콘텐츠를 설치하고 패키징합니다.

계층 콘텐츠를 설치하고 패키징하려면 다음을 수행합니다.

1. `sample-apps/layer-java` 디렉터리에 필요한 샘플 코드가 포함된 aws-lambda-developer-guide GitHub 리포지토리를 복제합니다.

   ```
   git clone https://github.com/awsdocs/aws-lambda-developer-guide.git
   ```

2. `layer-java` 샘플 앱의 `layer` 디렉터리로 이동합니다. 이 디렉터리에는 계층을 올바르게 생성하고 패키징하는 데 사용하는 스크립트가 포함되어 있습니다.

   ```
   cd aws-lambda-developer-guide/sample-apps/layer-java/layer
   ```

3. pom.xml 파일을 검사합니다. 이 <dependencies> 섹션에서는 계층에 포함할 종속성, 즉 `aws-lambda-java-core` 및 `jackson-databind` 라이브러리를 정의합니다. 이 파일을 업데이트하여 자체 계층에 포함하려는 종속성을 포함할 수 있습니다.

   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**
>
> 이 pom.xml 파일의 <build> 섹션에는 2개의 플러그인이 포함되어 있습니다. maven-compiler-plugin은 소스 코드를 컴파일합니다. maven-shade-plugin은 아티팩트를 단일 uber-jar로 패키징합니다.

4. 두 스크립트를 모두 실행할 수 있는 권한이 있는지 확인하세요.

```
chmod 744 1-install.sh && chmod 744 2-package.sh
```

5. 다음 명령을 사용하여 1-install.sh 스크립트를 실행하세요.

```
./1-install.sh
```

이 스크립트는 현재 디렉터리에서 mvn clean install을 실행합니다. 그러면 target/ 디렉터리에 필요한 모든 종속성이 포함된 uber-jar가 생성됩니다.

Example 1-install.sh

```
mvn clean install
```

6. 다음 명령을 사용하여 2-package.sh 스크립트를 실행하세요.

```
./2-package.sh
```

이 스크립트는 계층 콘텐츠를 적절하게 패키징하는 데 필요한 java/lib 디렉터리 구조를 생성합니다. 그런 다음 /target 디렉터리의 uber-jar를 새로 생성한 java/lib 디렉터리로 복사합니다. 마지막으로 스크립트는 java 디렉터리의 내용을 layer_content.zip이라는 파일로 압축합니

다. 이는 계층의 .zip 파일입니다. the section called “Java 런타임의 계층 경로” 섹션에 표시된 것처럼 파일의 압축을 풀고 올바른 파일 구조가 포함되어 있는지 확인할 수 있습니다.

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
```

계층의 생성

이 섹션에서는 이전 섹션에서 생성한 `layer_content.zip` 파일을 가져와 Lambda 계층으로 업로드합니다. AWS Command Line Interface(AWS CLI)를 통해 AWS Management Console 또는 Lambda API를 사용하여 계층을 업로드할 수 있습니다. 계층 .zip 파일을 업로드할 때 다음 PublishLayerVersion AWS CLI 명령에서 `java21`을 호환 런타임으로, `arm64`를 호환 아키텍처로 지정합니다.

```
aws lambda publish-layer-version --layer-name java-jackson-layer \
    --zip-file fileb://layer_content.zip \
    --compatible-runtimes java21 \
    --compatible-architectures "arm64"
```

응답에서 `arn:aws:lambda:us-east-1:123456789012:layer:java-jackson-layer:1`처럼 보이는 `LayerVersionArn`에 유의하세요. 이 자습서의 다음 단계인 함수에 계층을 추가할 때 이 Amazon 리소스 이름(ARN)이 필요합니다.

함수에 계층 추가

이 섹션에서는 함수 코드에 Jackson 라이브러리를 사용하는 샘플 Lambda 함수를 배포한 다음 계층을 연결합니다. 함수를 배포하려면 the section called “실행 역할(함수가 다른 리소스에 액세스할 수 있는 권한)”이 필요합니다. 기존 실행 역할이 없으면 접을 수 있는 섹션의 단계를 따르세요. 그렇지 않은 경우 다음 섹션으로 건너뛰어 기능을 배포하세요.

(선택 사항) 실행 역할 생성

실행 역할을 만들려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.

3. 다음 속성을 사용하여 역할을 만듭니다.

   - 신뢰할 수 있는 엔터티 – Lambda.
   - 권한 – AWSLambdaBasicExecutionRole.
   - 역할 이름 – **`lambda-role`**.

   AWSLambdaBasicExecutionRole 정책은 함수가 CloudWatch Logs에 로그를 쓰는 데 필요한 권한을 가집니다.

Lambda 함수를 배포하려면 다음을 수행합니다.

1. `function/` 디렉터리로 이동합니다. 현재 `layer/` 디렉터리에 있는 경우 다음 명령을 실행하세요.

   ```
   cd ../function
   ```

2. 함수 코드를 검토합니다. 이 함수는 `Map<String, String>`을 입력으로 받고 Jackson을 사용하여 입력을 JSON 문자열로 작성한 후 미리 정의된 F1Car Java 객체로 변환합니다. 마지막으로 함수는 F1Car 객체의 필드를 사용하여 함수가 반환하는 문자열을 구성합니다.

   ```
   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. 다음 Maven 명령을 사용하여 프로젝트를 빌드합니다.

```
mvn package
```

이 명령은 `target/` 디렉터리에 `layer-java-function-1.0-SNAPSHOT.jar`라는 이름의 JAR 파일을 생성합니다.

4. 함수를 배포합니다. 다음 AWS CLI 명령에서 `--role` 파라미터를 실행 역할 ARN으로 바꿉니다.

```
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
```

(선택 사항) 계층을 연결하지 않고 함수를 호출합니다.

이 시점에서 계층을 연결하기 전에 함수를 호출해 볼 수 있습니다(선택 사항). 이렇게 시도하면 함수가 `requests` 패키지를 참조할 수 없으므로 `ClassNotFoundException`이 반환됩니다. 함수를 호출하려면 다음 AWS CLI 명령을 사용합니다.

```
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
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

특정 오류를 보려면 출력 `response.json` 파일을 엽니다. 다음 오류 메시지와 함께 `ClassNotFoundException`이 표시됩니다.

```
"errorMessage":"com.fasterxml.jackson.databind.ObjectMapper","errorType":"java.lang.ClassNotFou
```

그런 다음 계층을 함수에 연결합니다. 다음 AWS CLI 명령에서 `--layers` 파라미터를 이전에 기록해 둔 계층 버전 ARN으로 바꿉니다.

```
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"
```

마지막으로 다음 AWS CLI 명령을 사용하여 함수를 호출합니다.

```
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
```

다음과 유사한 출력 화면이 표시되어야 합니다.

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

이는 함수가 Jackson 종속성을 사용하여 함수를 올바르게 실행할 수 있었음을 나타냅니다. 출력 `response.json` 파일이 올바른 반환된 문자열을 포함하는지 확인할 수 있습니다.

```
"Max Verstappen is a driver for team Red Bull"
```

(선택 사항) 리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 계층을 삭제하려면 다음을 수행합니다.

1. Lambda 콘솔의 계층 페이지를 엽니다.
2. 생성한 계층을 선택합니다.

3. 삭제를 선택한 다음 삭제를 다시 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

Lambda SnapStart를 사용하여 시작 성능 개선

Java용 Lambda SnapStart는 일반적으로 함수 코드를 변경하지 않고도 추가 비용 없이 지연 시간에 민감한 애플리케이션의 시작 성능을 최대 10배까지 높일 수 있습니다. 시작 지연 시간(콜드 스타트 시간이라고도 함)이 발생하는 가장 큰 원인은 Lambda가 함수를 초기화하는 데 시간을 소비하기 때문입니다. 여기에는 함수 코드 로드 시간, 런타임 시작 시간, 함수 코드 초기화 시간이 포함됩니다.

SnapStart를 사용하면 함수 버전을 게시할 때 Lambda가 함수를 초기화합니다. Lambda는 초기화된 실행 환경의 메모리 및 디스크 상태 Firecracker microVM 스냅샷을 생성하고 스냅샷을 암호화하며 짧은 지연 시간으로 액세스할 수 있도록 스냅샷을 캐싱합니다. 함수 버전을 처음 호출하는 경우와 호출이 스케일 업되는 경우 Lambda는 실행 환경을 처음부터 초기화하는 대신 캐싱된 스냅샷에서 새 실행 환경을 재개하여 시작 지연 시간을 개선합니다.

> **Important**
>
> 애플리케이션에서 상태의 고유성이 중요한 경우 함수 코드를 평가하여 스냅샷 작업에 복원력이 있는지 확인해야 합니다. 자세한 내용은 Lambda를 사용한 고유성 다루기 SnapStart 단원을 참조하십시오.

주제

지원 기능 및 제한 사항

SnapStart는 Java 11 및 그 이상의 Java 관리형 런타임을 지원합니다. 다른 관리형 런타임(예: `nodejs20.x` 및 `python3.12`), OS 전용 런타임, 컨테이너 이미지는 지원되지 않습니다.

SnapStart는 프로비저닝된 동시성, arm64 아키텍처, Amazon Elastic File System(Amazon EFS) 또는 512MB보다 큰 임시 스토리지를 지원하지 않습니다.

SnapStart로 작업을 수행하는 데에는 Lambda 콘솔, AWS Command Line Interface(AWS CLI), Lambda API, AWS SDK for Java, AWS CloudFormation, AWS Serverless Application Model(AWS SAM) 및 AWS Cloud Development Kit (AWS CDK)를 사용할 수 있습니다. 자세한 내용은 Lambda 활성화 및 관리 SnapStart 단원을 참조하십시오.

> **Note**
>
> 게시된 함수 버전 및 버전을 가리키는 별칭에서만 SnapStart를 사용할 수 있습니다. 게시되지 않은 함수 버전($LATEST)에서는 SnapStart를 사용할 수 없습니다.

지원되는 리전

SnapStart는 다음 AWS 리전에서 사용할 수 있습니다.

- 미국 동부(버지니아 북부)
- 미국 동부(오하이오)
- 미국 서부(캘리포니아 북부)
- 미국 서부(오레곤)
- 아프리카(케이프타운)
- 아시아 태평양(홍콩)
- 아시아 태평양(뭄바이)
- 아시아 태평양(하이데라바드)
- 아시아 태평양(도쿄)
- 아시아 태평양(서울)
- 아시아 태평양(오사카)
- 아시아 태평양(싱가포르)

- 아시아 태평양(시드니)
- 아시아 태평양(자카르타)
- 아시아 태평양(멜버른)
- 캐나다(중부)
- 유럽(스톡홀름)
- 유럽(프랑크푸르트)
- 유럽(취리히)
- 유럽(아일랜드)
- 유럽(런던)
- 유럽(파리)
- 유럽(밀라노)
- 유럽(스페인)
- 중동(UAE)
- 중동(바레인)
- 남아메리카(상파울루)

호환성 고려 사항

SnapStart를 사용하면 Lambda가 단일 스냅샷을 여러 실행 환경의 초기 상태로 사용합니다. 초기화 단계에서 함수가 다음을 사용하는 경우 SnapStart를 사용하기 전에 몇 가지 사항을 변경해야 할 수 있습니다.

Uniqueness

초기화 코드가 스냅샷에 포함된 고유한 콘텐츠를 생성하는 경우, 해당 콘텐츠가 여러 실행 환경에서 재사용될 때 콘텐츠가 고유하지 않게 될 수 있습니다. SnapStart를 사용할 때 고유성을 유지하려면 초기화 후에 고유한 컨텐츠를 생성해야 합니다. 여기에는 유사 무작위성을 생성하는 데 사용되는 고유 ID, 고유 보안 암호 및 엔트로피가 포함됩니다. 고유성을 복원하는 방법은 Lambda를 사용한 고유성 다루기 SnapStart 섹션을 참조하세요.

네트워크 연결

Lambda가 스냅샷에서 함수를 재개할 때, 초기화 단계에서 함수가 설정한 연결 상태는 보장되지 않습니다. 네트워크 연결 상태를 확인하고 필요에 따라 다시 설정하세요. 대부분의 경우 AWS SDK가 설정한 네트워크 연결이 자동으로 재개됩니다. 다른 연결에 대해서는 모범 사례를 참조하세요.

임시 데이터

일부 함수는 초기화 단계에서 임시 보안 인증 정보나 캐싱된 타임스탬프 같은 임시 데이터를 다운로드하거나 초기화합니다. SnapStart를 사용하지 않는 경우에도 임시 데이터를 사용하기 전에 함수 핸들러에서 해당 데이터를 새로 고치세요.

SnapStart 요금

SnapStart를 사용하는 데 따른 추가 비용이 없습니다. 함수에 대한 요청 수, 코드를 실행하는 데 걸리는 시간, 함수에 대해 구성된 메모리를 기준으로 요금이 청구됩니다. 지속 시간은 코드 실행을 시작한 시점부터 코드가 반환되거나 종료될 때까지 계산되며 가장 가까운 1ms 단위로 반올림합니다.

지속 시간 요금은 함수 핸들러에서 실행되는 코드, 핸들러 외부에서 선언된 초기화 코드, 런타임(JVM)을 로드하는 데 걸리는 시간, 런타임 후크에서 실행되는 모든 코드에 적용됩니다. Lambda의 기간 계산 방식에 대한 자세한 내용은 Lambda에 대한 모니터링 SnapStart의 내용을 참조하세요.

SnapStart를 사용하여 구성된 함수의 경우 Lambda는 주기적으로 실행 환경을 재활용하고 초기화 코드를 다시 실행합니다. Lambda는 복원성을 위해 여러 가용 영역에 스냅샷을 생성합니다. Lambda가 다른 가용 영역에서 초기화 코드를 다시 실행할 때마다 요금이 부과됩니다. Lambda의 요금 계산 방식에 대한 자세한 내용은 AWS Lambda 요금을 참조하세요.

Lambda SnapStart와 프로비저닝된 동시성 비교

Lambda SnapStart와 프로비저닝된 동시성은 모두 함수가 스케일 업될 때 콜드 스타트와 비정상적인 지연 시간을 줄일 수 있습니다. SnapStart를 사용하면 추가 비용 없이 시작 성능을 최대 10배까지 높일 수 있습니다. 프로비저닝된 동시성은 두 자릿수 밀리초 내에 함수가 초기화되고 응답 준비가 되도록 합니다. 프로비저닝된 동시성을 구성하면 AWS 계정에 요금이 부과됩니다. 애플리케이션에 엄격한 콜드 스타트 지연 시간 요구 사항이 있는 경우 프로비저닝된 동시성을 사용하세요. 동일한 함수 버전에 SnapStart와 프로비저닝된 동시성을 모두 사용할 수는 없습니다.

> **Note**
>
> SnapStart는 대규모 함수 호출에서 사용할 때 가장 효과적입니다. 자주 호출되지 않는 함수에서는 동일한 성능 향상이 이루어지지 않을 수 있습니다.

추가적인 리소스

이 장의 다른 주제를 읽어보는 것 외에도 AWS Lambda SnapStart를 통한 더 빠른 시작 워크숍에 참여해 보고 AWS re:Invent 2022에서 제공하는 Fast cold starts for your Java functions 세션을 시청하는 것이 좋습니다.

Lambda 활성화 및 관리 SnapStart

사용하려면 SnapStart 새 Lambda SnapStart 함수 또는 기존 Lambda 함수를 활성화하십시오. 그런 다음 함수 버전을 게시하고 호출합니다.

주제

- 활성화 SnapStart (콘솔)
- 활성화 중 SnapStart () AWS CLI
- 활성화 SnapStart (API)
- SnapStart Lambda 및 함수 상태
- 스냅샷 업데이트
- 와 함께 사용 SnapStart AWS SDK for Java
- AWS CloudFormationAWS SAM, 및 SnapStart 와 함께 사용 AWS CDK
- 스냅샷 삭제

활성화 SnapStart (콘솔)

기능을 SnapStart 활성화하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. Configuration(구성)을 선택한 다음 General configuration(일반 구성)을 선택합니다.
4. General configuration(일반 구성) 창에서 Edit(편집)를 선택합니다.
5. 기본 설정 편집 페이지에서 게시된 버전을 선택합니다. SnapStart
6. 저장을 선택합니다.
7. 함수 버전을 게시합니다. Lambda는 코드를 초기화하고, 초기화된 실행 환경의 스냅샷을 생성한 다음, 액세스 지연 시간이 짧아지도록 스냅샷을 캐싱합니다.
8. 함수 버전을 호출합니다.

활성화 중 SnapStart () AWS CLI

기존 기능을 SnapStart 활성화하려면

1. --snap-start옵션과 함께 update-function-configuration명령을 실행하여 함수 구성을 업데이트하십시오.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. publish-version 명령을 사용하여 함수 버전을 게시합니다.

```
aws lambda publish-version \
  --function-name my-function
```

3. get-function-configuration명령을 실행하고 버전 번호를 지정하여 함수 버전에 대해 SnapStart 활성화되었는지 확인합니다. 다음은 버전 1을 지정하는 예입니다.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

응답이 OptimizationStatus0n'예'이고 State가 `Active` SnapStart 's'로 표시되면 활성화되고 지정된 함수 버전에 대한 스냅샷을 사용할 수 있습니다.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. invoke 명령을 실행하고 버전을 지정하여 함수 버전을 호출합니다. 다음은 버전 1을 호출하는 예입니다.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

새 함수를 만들 SnapStart 때 활성화하려면

1. --snap-start 옵션과 함께 create-function 명령을 실행하여 함수를 생성합니다. --role에 실행 역할의 Amazon 리소스 이름(ARN)을 지정합니다.

```
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 명령을 사용하여 버전을 생성합니다.

```
aws lambda publish-version \
  --function-name my-function
```

3. get-function-configuration명령을 실행하고 버전 번호를 지정하여 함수 버전에 대해 SnapStart 활성화되었는지 확인합니다. 다음은 버전 1을 지정하는 예입니다.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

응답이 OptimizationStatus0n'예'이고 State가 `Active` SnapStart 's'로 표시되면 활성화되고 지정된 함수 버전에 대한 스냅샷을 사용할 수 있습니다.

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. invoke 명령을 실행하고 버전을 지정하여 함수 버전을 호출합니다. 다음은 버전 1을 호출하는 예입니다.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

활성화 SnapStart (API)

활성화하려면 SnapStart

1. 다음 중 하나를 수행합니다.
   - SnapStart파라미터와 함께 SnapStart CreateFunctionAPI 작업을 사용하여 activate로 새 함수를 생성합니다.
   - SnapStart매개변수와 함께 UpdateFunctionConfiguration액션을 사용하여 기존 함수를 SnapStart 활성화합니다.
2. PublishVersion액션과 함께 함수 버전을 게시하십시오. Lambda는 코드를 초기화하고, 초기화된 실행 환경의 스냅샷을 생성한 다음, 액세스 지연 시간이 짧아지도록 스냅샷을 캐싱합니다.
3. GetFunctionConfiguration액션을 사용하여 함수 버전에서 SnapStart 활성화되었는지 확인합니다. 버전 번호를 지정하여 해당 버전에서 SnapStart 활성화되었는지 확인합니다. 응답에 On 'State'가 `Active` '예'로 SnapStart 표시되면 활성화되고 지정된 함수 버전에 대한 스냅샷을 사용할 수 있습니다. OptimizationStatus

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. Invoke 작업을 사용하여 함수 버전을 호출합니다.

SnapStart Lambda 및 함수 상태

를 사용할 때 다음과 같은 함수 상태가 발생할 수 있습니다. SnapStart 또한 Lambda가 주기적으로 실행 환경을 재활용하고 로 구성된 함수의 초기화 코드를 다시 실행할 때도 발생할 수 있습니다. SnapStart

- `Pending` - Lambda가 코드를 초기화하고 초기화된 실행 환경의 스냅샷을 생성하고 있습니다. 해당 함수 버전에서 작동하는 호출 또는 기타 API 작업은 실패합니다.
- `Active` - 스냅샷 생성이 완료되었으므로 함수를 호출할 수 있습니다. 사용하려면 SnapStart 게시되지 않은 버전 ($LATEST) 이 아니라 게시된 함수 버전을 호출해야 합니다.
- `Inactive` - 해당 함수 버전이 14일 동안 호출되지 않았습니다. 함수 버전이 `Inactive` 상태가 되면 Lambda가 해당 스냅샷을 삭제합니다. 14일이 지난 후에 함수 버전을 호출하면 Lambda는 `SnapStartNotReadyException` 응답을 반환하고 새 스냅샷을 초기화하기 시작합니다. 함수 버전이 `Active` 상태에 도달할 때까지 기다린 후 다시 호출하세요.
- `Failed` - 초기화 코드를 실행하거나 스냅샷을 생성할 때 Lambda에서 오류가 발생했습니다.

스냅샷 업데이트

Lambda는 게시된 각 함수 버전마다 스냅샷을 생성합니다. 스냅샷을 업데이트하려면 새 함수 버전을 게시합니다. Lambda는 최신 런타임 및 보안 패치를 사용하여 스냅샷을 자동으로 업데이트합니다.

와 함께 사용 SnapStart AWS SDK for Java

함수에서 AWS SDK를 호출하기 위해 Lambda는 함수의 실행 역할을 가정하여 휘발성 자격 증명 세트를 생성합니다. 이러한 보안 인증 정보는 함수를 호출할 때 환경 변수로 사용할 수 있습니다. 코드에서 직접 SDK의 보안 인증 정보를 제공할 필요가 없습니다. 기본적으로 보안 인증 공급자 체인은 사용자가 보안 인증 정보를 설정할 수 있는 각 위치를 순차적으로 확인하고, 일반적으로 환경 변수 (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` 및 `AWS_SESSION_TOKEN`)인 사용 가능한 첫 번째 위치를 선택합니다.

> **Note**
>
> SnapStart 가 활성화되면 Java 런타임은 액세스 키 환경 변수 대신 컨테이너 자격 증명 (`AWS_CONTAINER_CREDENTIALS_FULL_URI`및`AWS_CONTAINER_AUTHORIZATION_TOKEN`)을 자동으로 사용합니다. 따라서 함수가 복원되기 전에 보안 인증 정보가 만료되지 않습니다.

AWS CloudFormationAWS SAM, 및 SnapStart 와 함께 사용 AWS CDK

- AWS CloudFormation: 템플릿에서 SnapStart엔티티를 선언합니다.
- AWS Serverless Application Model(AWS SAM): 템플릿에서 SnapStart속성을 선언합니다.
- AWS Cloud Development Kit (AWS CDK): SnapStartProperty유형을 사용하세요.

스냅샷 삭제

Lambda는 다음 경우에 스냅샷을 삭제합니다.

- 함수 또는 함수 버전을 삭제한 경우.
- 14일 동안 함수 버전을 호출하지 않은 경우. 호출 없이 14일이 지나면 함수 버전이 비활성 상태로 전환됩니다. 14일이 지난 후에 함수 버전을 호출하면 Lambda는 `SnapStartNotReadyException` 응답을 반환하고 새 스냅샷을 초기화하기 시작합니다. 함수 버전이 활성 상태에 도달할 때까지 기다린 후 다시 호출하세요.

Lambda는 일반 데이터 보호 규정(GDPR)에 따라, 삭제된 스냅샷에 연결된 모든 리소스를 제거합니다.

Lambda를 사용한 고유성 다루기 SnapStart

SnapStart 함수에서 호출이 확장되면 Lambda는 초기화된 단일 스냅샷을 사용하여 여러 실행 환경을 재개합니다. 초기화 코드가 스냅샷에 포함된 고유한 콘텐츠를 생성하는 경우, 해당 콘텐츠가 여러 실행 환경에서 재사용될 때 콘텐츠가 고유하지 않게 될 수 있습니다. 사용 시 고유성을 유지하려면 초기화 SnapStart 후 고유한 콘텐츠를 생성해야 합니다. 여기에는 유사 무작위성을 생성하는 데 사용되는 고유 ID, 고유 보안 암호 및 엔트로피가 포함됩니다.

코드에서 고유성을 유지하려면 다음 모범 사례를 따르는 것이 좋습니다. 또한 Lambda는 고유성을 가정하는 코드를 확인하는 데 도움이 되는 SnapStart 오픈 소스 스캔 도구를 제공합니다. 초기화 단계에서 고유한 데이터를 생성하는 경우 런타임 후크를 사용하여 고유성을 복원할 수 있습니다. 런타임 후크를 사용하면 Lambda가 스냅샷을 생성하기 직전이나 Lambda가 스냅샷에서 함수를 재개한 직후에 특정 코드를 실행할 수 있습니다.

초기화 중에 고유성에 의존하는 상태를 저장하지 마세요.

함수의 초기화 단계에서는 로깅을 위한 고유 ID를 생성하는 경우와 같이 고유하도록 의도된 데이터를 캐싱하지 마세요. 대신 함수 핸들러 내에서 고유한 데이터를 생성하거나 런타임 후크를 사용하는 것이 좋습니다.

Example - 함수 핸들러에서 고유 ID 생성

다음 예제는 함수 핸들러에서 UUID를 생성하는 방법을 보여줍니다.

```
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!";
    }
  }
```

암호학적으로 안전한 가상 난수 생성기(CSPRNG) 사용

애플리케이션이 무작위성에 의존하는 경우 암호학적으로 안전한 난수 생성기(CSPRNG)를 사용하는 것이 좋습니다. Java용 Lambda 관리형 런타임에는 임의성을 자동으로 유지하는 두 개의 내장

CSPRNG (OpenSSL 1.0.2 및) 가 포함되어 있습니다. `java.security.SecureRandom` SnapStart 항상 임의의 숫자를 가져오거나 무작위성을 유지하는 소프트웨어. `/dev/random` `/dev/urandom` SnapStart

Example — 자바. 보안. SecureRandom

다음 예제에서는 `java.security.SecureRandom`을 사용하여 스냅샷에서 함수를 복원한 경우에도 고유한 번호 시퀀스를 생성합니다.

```
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 스캔 도구

Lambda는 고유성을 가정한 코드를 검사하는 데 도움이 되는 검사 도구를 제공합니다. SnapStart 스캐닝 도구는 일련의 규칙에 대해 정적 분석을 실행하는 오픈 소스 SpotBugs플러그인입니다. 이 검사 도구는 고유성과 관련한 가정에 위배될 수 있는 잠재적인 코드 구현을 식별하는 데 도움이 됩니다. 설치 지침과 검사 도구가 수행하는 검사 목록은 aws-lambda-snapstart-java-rules 저장소를 참조하십시오. GitHub

를 사용하여 고유성을 처리하는 방법에 대해 자세히 SnapStart 알아보려면 AWS Compute AWS Lambda SnapStart 블로그에서 더 빠르게 시작하기를 참조하십시오.

Lambda 함수 스냅샷 전후 코드 구현

Lambda가 스냅샷을 생성하기 전이나 Lambda가 스냅샷에서 함수를 재개한 후에 런타임 후크를 사용하여 코드를 구현할 수 있습니다. 런타임 후크는 오픈 소스 체크포인트 Coordinated Restore at Checkpoint(CRaC) 프로젝트의 일부로 제공됩니다. Open Java Development Kit(OpenJDK)용 CRaC는 개발 중입니다. 참조 애플리케이션에 CRaC를 사용하는 방법의 예는 GitHub에서 CRaC 리포지토리를 참조하세요. CRaC는 세 가지 주요 요소를 사용합니다.

- `Resource` - `beforeCheckpoint()`와 `afterRestore()`라는 두 가지 메서드가 있는 인터페이스입니다. 이러한 메서드를 사용하여 스냅샷 생성 전과 복원 후에 실행할 코드를 구현합니다.
- `Context <R extends Resource>` - 체크포인트 및 복원에 대한 알림을 받으려면 `Resource`를 `Context`에 등록해야 합니다.
- `Core` - 정적 메서드 `Core.getGlobalContext()`를 통해 기본 글로벌 `Context`를 제공하는 조정 서비스입니다.

`Context` 및 `Resource`에 대한 자세한 내용은 CRaC 설명서에서 org.crac 패키징을 참조하세요.

다음 단계에 따라 org.crac 패키지를 사용하여 런타임 후크를 구현합니다. Lambda 런타임에는 체크포인트를 생성하기 전과 복원 후에 런타임 후크를 호출하는 사용자 지정 CRaC 컨텍스트 구현이 포함되어 있습니다.

1단계: 빌드 구성 업데이트

빌드 구성에 `org.crac` 종속성을 추가합니다. 다음 예에서는 Gradle을 사용합니다. 다른 빌드 시스템의 예제는 Apache Maven 설명서를 참조하세요.

```
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'
}
```

2단계: Lambda 핸들러 업데이트

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

자세한 내용은 Java에서 Lambda 함수 핸들러 정의 단원을 참조하십시오.

다음 예제 핸들러는 체크포인트 생성 전(beforeCheckpoint())과 복원 후(afterRestore())에 코드를 실행하는 방법을 보여줍니다. 또한 이 핸들러는 런타임 관리 글로벌 Resource에 Context를 등록합니다.

> **Note**
>
> Lambda가 스냅샷을 생성할 때 초기화 코드를 최대 15분 동안 실행할 수 있습니다. 시간 제한은 130초 또는 구성된 함수 제한 시간(최대 900초) 중 더 높은 값입니다. beforeCheckpoint() 런타임 후크는 초기화 코드 시간 제한에 포함됩니다. Lambda가 스냅샷을 복원할 때, 런타임(JVM)이 로드되고 afterRestore() 런타임 후크가 제한 시간(10초) 내에 완료되어야 합니다. 그렇지 않으면 SnapStartTimeoutException이 발생합니다.

```
...
  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");
```

Context는 등록된 객체에 대해서만 WeakReference를 유지합니다. Resource가 가비지 수집된 경우 런타임 후크가 실행되지 않습니다. 런타임 후크가 실행되도록 하려면 코드에서 Resource에 대한 강력한 참조를 유지해야 합니다.

다음은 피해야 할 두 가지 패턴의 예입니다.

Example - 강력한 참조가 없는 객체

```
Core.getGlobalContext().register( new MyResource() );
```

Example - 익명 클래스의 객체

```
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 {
    // ...
   }

} );
```

대신 강력한 참조를 유지하세요. 다음 예제에서 등록된 리소스는 가비지 수집되지 않았으며 런타임 후크가 일관되게 실행됩니다.

Example - 강력한 참조가 있는 객체

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda에 대한 모니터링 SnapStart

CloudWatchAmazonAWS X-Ray, 및 를 사용하여 SnapStart Lambda 함수를 모니터링할 수 있습니다. Lambda 텔레메트리 API

> **Note**
>
> AWS_LAMBDA_LOG_GROUP_NAME및 AWS_LAMBDA_LOG_STREAM_NAME 환경 변수는 SnapStart Lambda 함수에서 사용할 수 없습니다.

CloudWatch 에 대한 SnapStart

SnapStart 함수의 CloudWatch 로그 스트림 형식에는 몇 가지 차이점이 있습니다.

- 초기화 로그 — 새 실행 환경이 생성되면 REPORT에는 Init Duration 필드가 포함되지 않습니다. Lambda는 함수 호출 대신 버전을 생성할 때 함수를 SnapStart 초기화하기 때문입니다. SnapStart 함수의 경우 Init Duration 필드는 레코드에 있습니다. INIT_REPORT 이 레코드에는 beforeCheckpoint 런타임 후크의 지속 시간을 비롯한 초기화 단계의 지속 시간 세부 정보가 표시됩니다.
- 호출 로그 — 새 실행 환경이 생성되면 REPORT에는 Restore Duration와 Billed Restore Duration 필드가 포함됩니다.
  - Restore Duration: Lambda가 스냅샷을 복원하고, 런타임(JVM)을 로드하고, afterRestore 런타임 후크를 실행하는 데 걸리는 시간입니다. 스냅샷 복원 프로세스에는 microVM 외부 작업에 소요되는 시간이 포함될 수 있습니다. 이 시간은 Restore Duration에서 보고됩니다.
  - Billed Restore Duration: Lambda가 런타임(JVM)을 로드하고, afterRestore 후크를 실행하는 데 걸리는 시간입니다. 스냅샷을 복원하는 데 걸리는 시간에 대해서는 요금이 청구되지 않습니다.

> **Note**
>
> 지속 시간 요금은 함수 핸들러에서 실행되는 코드, 핸들러 외부에서 선언된 초기화 코드, 런타임(JVM)을 로드하는 데 걸리는 시간, 런타임 후크에서 실행되는 모든 코드에 적용됩니다. 자세한 설명은 SnapStart 요금 섹션을 참조하세요.

콜드 스타트 지속 시간은 Restore Duration과 Duration의 합계입니다.

다음 예는 함수의 지연 시간 백분위수를 반환하는 Lambda Insights 쿼리입니다. SnapStart Lambda Insights 쿼리에 대한 자세한 내용은 쿼리를 사용하여 함수 문제를 해결하는 예제 워크플로 섹션을 참조하세요.

```
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
```

X-Ray 액티브 트레이싱에 대한 SnapStart

X-Ray를 사용하여 SnapStart Lambda 함수에 대한 요청을 추적할 수 있습니다. SnapStart 함수의 X-Ray 서브세그먼트에는 몇 가지 차이점이 있습니다.

- 함수에는 `Initialization` 하위 세그먼트가 없습니다. SnapStart
- `Restore` 하위 세그먼트는 Lambda가 스냅샷을 복원하고, 런타임(JVM)을 로드하고, `afterRestore` 런타임 후크를 실행하는 데 걸리는 시간을 보여줍니다. 스냅샷 복원 프로세스에는 microVM 외부 작업에 소요되는 시간이 포함될 수 있습니다. 이 시간은 `Restore` 하위 세그먼트에서 보고됩니다. 스냅샷을 복원하기 위해 microVM 외부에서 소요된 시간에 대한 요금은 부과되지 않습니다.

에 대한 텔레메트리 API 이벤트 SnapStart

Lambda는 SnapStart 다음 이벤트를 다음으로 전송합니다. 텔레메트리 API

- `platform.restoreStart` - `Restore` 단계가 시작된 시간을 표시합니다.
- `platform.restoreRuntimeDone` - `Restore` 단계가 성공했는지 여부를 표시합니다. Lambda는 런타임이 `restore/next` 런타임 API 요청을 보낼 때 이 로그 메시지를 보냅니다. 가능한 상태로는 성공, 실패 및 시간 초과가 있습니다.
- `platform.restoreReport` - `Restore` 단계가 지속된 시간과 이 단계에서 청구된 시간(밀리초)이 표시됩니다.

Amazon API Gateway 및 함수 URL 지표

API Gateway를 사용하여 웹 API를 생성하는 경우 이 IntegrationLatency지표를 사용하여 end-to-end 지연 시간 (API Gateway가 요청을 백엔드로 릴레이하는 시점과 백엔드로부터 응답을 받는 시점 사이의 시간) 을 측정할 수 있습니다.

Lambda 함수 URL을 사용하는 경우, 지표를 사용하여 지연 시간 (함수 UrlRequestLatencyURL이 요청을 수신하는 시점과 함수 URL이 응답을 반환하는 시점 사이의 시간) 을 end-to-end 측정할 수 있습니다.

Lambda를 위한 보안 모델 SnapStart

SnapStart Lambda는 저장 시 암호화를 지원합니다. Lambda는 AWS KMS key를 사용하여 스냅샷을 암호화합니다. 기본적으로 Lambda는 AWS 관리형 키를 사용합니다. 이 기본 동작이 워크플로에 적합한 경우 다른 작업을 설정할 필요가 없습니다. 그렇지 않으면 create-function 또는 update-function-configuration명령의 `--kms-key-arn` 옵션을 사용하여 고객 관리 키를 제공할 수 있습니다. AWS KMS KMS 키의 교체를 제어하거나 KMS 키를 관리하기 위한 조직의 요구 사항을 충족하기 위해 이 작업을 수행할 수 있습니다. 고객 관리형 키에는 표준 AWS KMS 요금이 발생합니다. 자세한 내용은 AWS Key Management Service 요금을 참조하십시오.

SnapStart 함수 또는 함수 버전을 삭제하면 해당 함수 또는 함수 버전에 `Invoke` 대한 모든 요청이 실패합니다. Lambda는 14일 동안 호출되지 않은 스냅샷을 자동으로 삭제합니다. Lambda는 일반 데이터 보호 규정(GDPR)에 따라, 삭제된 스냅샷에 연결된 모든 리소스를 제거합니다.

Lambda SnapStart 성능 극대화

주제

- 성능 튜닝
- 네트워킹 모범 사례

성능 튜닝

> **Note**
>
> SnapStart는 대규모 함수 호출에서 사용할 때 가장 효과적입니다. 자주 호출되지 않는 함수에서는 동일한 성능 향상이 이루어지지 않을 수 있습니다.

SnapStart의 이점을 극대화하려면 함수 핸들러가 아니라 초기화 코드에서 시작 지연 시간에 영향을 미치는 클래스를 미리 로드하는 것이 좋습니다. 이렇게 하면 과도한 클래스 로딩과 관련한 지연 시간이 호출 경로 외부로 오프로드되어 SnapStart를 사용한 시작 성능이 최적화됩니다.

초기화 중에 클래스를 미리 로드할 수 없는 경우 더미 호출을 통해 클래스를 미리 로드하는 것이 좋습니다. 이렇게 하려면 AWS Labs GitHub 리포지토리의 Pet Store 함수에서 가져온 다음 예제와 같이 함수 핸들러 코드를 업데이트합니다.

```
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);
        }
    }
```

네트워킹 모범 사례

Lambda가 스냅샷에서 함수를 재개할 때, 초기화 단계에서 함수가 설정한 연결 상태는 보장되지 않습니다. 대부분의 경우 AWS SDK가 설정한 네트워크 연결이 자동으로 재개됩니다. 다른 연결의 경우 다음 모범 사례를 따르는 것이 좋습니다.

네트워크 연결 재설정

스냅샷에서 함수를 재개할 때는 항상 네트워크 연결을 재설정합니다. 함수 핸들러에서 네트워크 연결을 재설정하는 것이 좋습니다. 또는 `afterRestore` 런타임 후크를 사용할 수도 있습니다.

호스트 이름을 고유한 실행 환경 식별자로 사용하지 않음

실행 환경을 애플리케이션의 고유한 노드 또는 컨테이너로 식별하는 데 `hostname`을 사용하지 않는 것이 좋습니다. SnapStart를 사용하면 단일 스냅샷이 여러 실행 환경의 초기 상태로 사용되며 모든 실행 환경은 `InetAddress.getLocalHost()`에 대해 동일한 `hostname` 값을 반환합니다. 고유한 실

행 환경 ID 또는 hostname 값이 필요한 애플리케이션의 경우 함수 핸들러에서 고유한 ID를 생성하는 것이 좋습니다. 또는afterRestore 런타임 후크를 사용하여 고유 ID를 생성한 다음 고유 ID를 실행 환경의 식별자로 사용할 수도 있습니다.

고정 소스 포트에 연결을 바인딩하지 않음

네트워크 연결을 고정 소스 포트에 바인딩하지 않는 것이 좋습니다. 스냅샷에서 함수가 재개되면 연결이 재설정되고 고정 소스 포트에 바인딩된 네트워크 연결이 실패할 수 있습니다.

Java DNS 캐시를 사용하지 않음

Lambda 함수는 이미 DNS 응답을 캐싱하고 있습니다. SnapStart에 다른 DNS 캐시를 사용하는 경우 스냅샷에서 함수를 재개할 때 연결 시간 초과가 발생할 수 있습니다.

java.util.logging.Logger 클래스는 JVM DNS 캐시를 간접적으로 활성화할 수 있습니다. 기본 설정을 재정의하려면 logger 초기화 전에 networkaddress.cache.ttl을 0으로 설정하십시오. 예제

```
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 오류를 방지하려면 networkaddress.cache.negative.ttl을 0으로 설정하는 것이 좋습니다. AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0 환경 변수를 사용하여 Lambda 함수에 대해 이 속성을 설정할 수 있습니다.

JVM DNS 캐시를 비활성화해도 Lambda의 관리형 DNS 캐싱은 비활성화되지 않습니다.

Java Lambda 함수 사용자 지정 설정

이 페이지에서는 AWS Lambda의 Java 함수와 관련된 설정을 설명합니다. 이러한 설정을 사용하여 Java 런타임 시작 동작을 사용자 지정할 수 있습니다. 이를 통해 코드를 수정하지 않고도 전체 함수 지연 시간을 줄이고 전체 함수 성능을 개선할 수 있습니다.

Sections

`JAVA_TOOL_OPTIONS` 환경 변수

Java에서 Lambda는 Lambda에서 추가 명령줄 변수를 설정할 수 있도록 `JAVA_TOOL_OPTIONS` 환경 변수를 지원합니다. 계층형 컴파일 설정을 사용자 지정하는 등 다양한 방식으로 이 환경 변수를 사용할 수 있습니다. 다음 예제에서는 이 사용 사례에 `JAVA_TOOL_OPTIONS` 환경 변수를 사용하는 방법을 설명합니다.

예제: 계층형 컴파일 설정 사용자 지정

계층형 컴파일은 Java 가상 머신(JVM)의 기능입니다. 특정 계층형 컴파일 설정을 사용하여 JVM의 JIT(Just-In-Time) 컴파일러를 최대한 활용할 수 있습니다. 일반적으로 C1 컴파일러는 빠른 시작 시간에 최적화되어 있습니다. C2 컴파일러는 최상의 전체 성능에 최적화되어 있지만 더 많은 메모리를 사용하고 이를 달성하는 데 더 오랜 시간이 걸립니다.

계층형 컴파일에는 5가지 수준이 있습니다. 수준 0에서 JVM은 Java 바이트 코드를 해석합니다. 수준 4에서 JVM은 C2 컴파일러를 사용하여 애플리케이션 시작 중 수집된 프로파일링 데이터를 분석합니다. 시간 경과에 따라 코드 사용을 모니터링하여 최상의 최적화를 식별합니다.

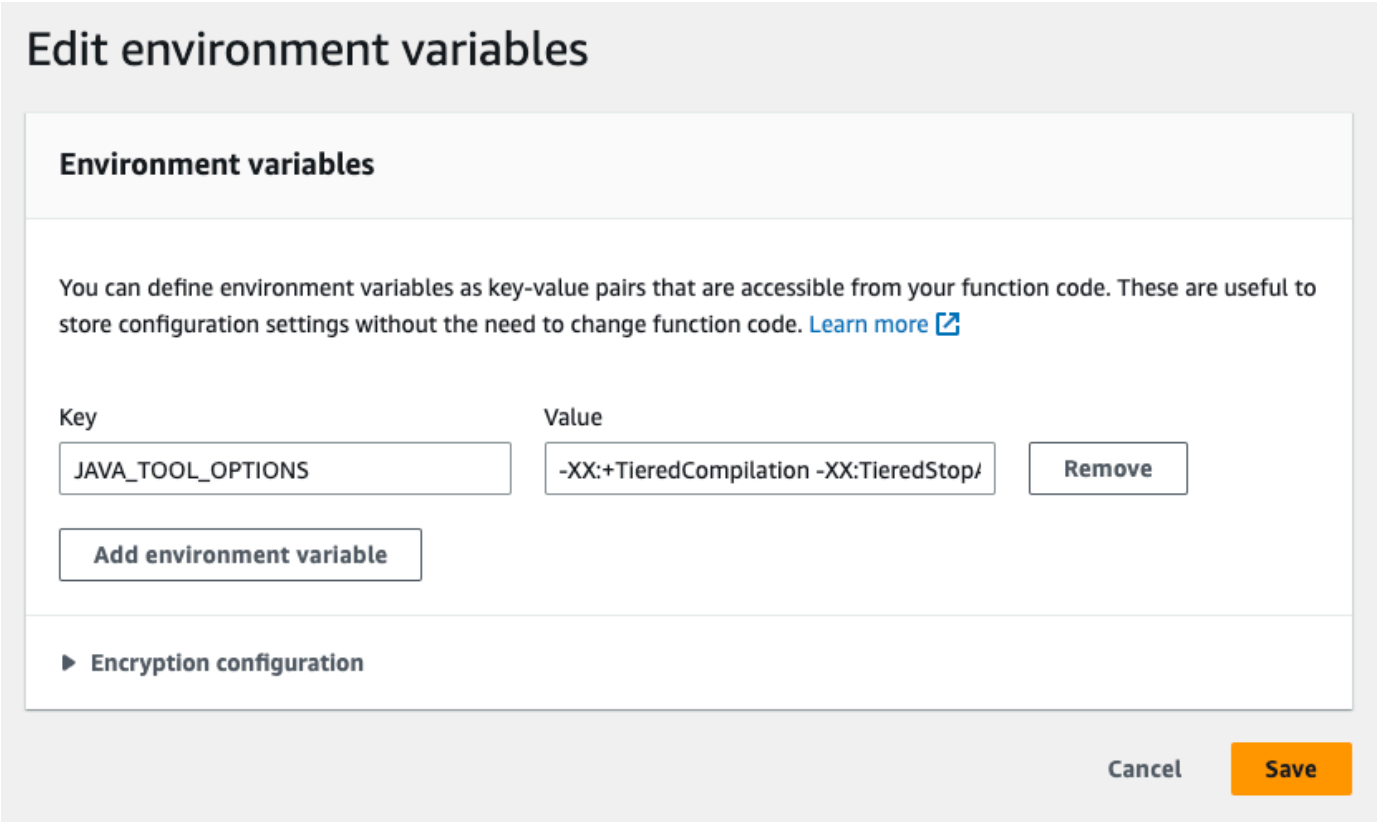
계층형 컴파일 수준을 사용자 지정하면 Java 함수 콜드 스타트 지연 시간을 줄이는 데 도움이 될 수 있습니다. 예를 들어, 계층형 컴파일 수준을 1로 설정하여 JVM이 C1 컴파일러를 사용하도록 합니다. 이 컴파일러는 최적화된 네이티브 코드를 빠르게 생성하지만 프로파일링 데이터를 생성하지 않으며 C2 컴파일러를 사용하지 않습니다.

Java 17 런타임에서 계층화된 컴파일에 대한 JVM 플래그는 기본적으로 수준 1에서 중지하도록 설정됩니다. Java 11 런타임 이하의 경우 다음 단계를 수행하여 계층화된 컴파일 수준을 1로 설정할 수 있습니다.

계층형 컴파일 설정 사용자 지정(콘솔)

1. Lambda 콘솔에서 함수 페이지를 엽니다.

2. 계층형 컴파일을 사용자 지정하려는 Java 함수를 선택합니다.
3. 구성 탭을 선택한 다음 왼쪽 메뉴에서 환경 변수를 선택합니다.
4. 편집을 선택합니다.
5. Add environment variable(환경 변수 추가)을 선택합니다.
6. 키로 `JAVA_TOOL_OPTIONS`를 입력합니다. 값으로 `-XX:+TieredCompilation -XX:TieredStopAtLevel=1`을 입력합니다.



7. Save(저장)를 선택합니다.

> **Note**
>
> Lambda SnapStart를 사용하여 콜드 스타트 문제를 완화할 수도 있습니다. SnapStart는 실행 환경의 캐시된 스냅샷을 사용하여 시작 성능을 크게 향상시킵니다. SnapStart 기능, 제한 사항 및 지원되는 리전에 대한 자세한 내용은 Lambda SnapStart를 사용하여 시작 성능 개선 섹션을 참조하세요.

예: JAVA_TOOL_OPTIONS를 사용하여 GC 동작 사용자 지정

Java 11 런타임은 가비지 수집을 위해 직렬 가비지 수집기(GC)를 사용합니다. 기본적으로 Java 17 런타임도 직렬 GC를 사용합니다. 그러나 Java 17에서는 `JAVA_TOOL_OPTIONS` 환경 변수를 사용하여 기본 GC를 변경할 수도 있습니다. 병렬 GC와 Shenandoah GC 중에서 선택할 수 있습니다.

예를 들어 워크로드에 더 많은 메모리와 여러 CPU를 사용하는 경우 병렬 GC를 사용하여 성능을 향상시킬 수 있습니다. `JAVA_TOOL_OPTIONS` 환경 변수 값에 다음을 추가하여 이 작업을 수행할 수 있습니다.

```
-XX:+UseParallelGC
```

AWS Lambda 컨텍스트 객체(Java)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러.에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 제공하는 메서드 및 속성들을 제공합니다.

컨텍스트 메서드

- `getRemainingTimeInMillis()` – 실행 시간이 초과되기까지 남은 시간(밀리초)을 반환합니다.
- `getFunctionName()` – Lambda 함수의 이름을 반환합니다.
- `getFunctionVersion()` – 함수의 버전을 반환합니다.
- `getInvokedFunctionArn()` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)을 반환합니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `getMemoryLimitInMB()` – 함수에 할당된 메모리의 양을 반환합니다.
- `getAwsRequestId()` – 호출 요청의 식별자를 반환합니다.
- `getLogGroupName()` – 함수에 대한 로그 그룹을 반환합니다.
- `getLogStreamName()` – 함수 인스턴스에 대한 로그 스트림을 반환합니다.
- `getIdentity()` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보를 반환합니다.
- `getClientContext()` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트를 반환합니다.
- `getLogger()` – 함수에 대한 로거 객체를 반환합니다.

다음 예제에서는 컨텍스트 객체를 사용하여 Lambda 로거에 액세스하는 함수를 보여 줍니다.

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;
  }
}
```

이 함수는 null을 반환하기 전에 수신 이벤트의 클래스 유형을 기록합니다.

Example 로그 출력

```
EVENT TYPE: class java.util.LinkedHashMap
```

컨텍스트 객체에 대한 인터페이스는 aws-lambda-java-core 라이브러리에서 사용할 수 있습니다. 이 인터페이스를 구현하여 테스트용 컨텍스트 클래스를 만들 수 있습니다. 다음 예제에서는 대부분의 속성 및 작업 테스트 로거에 대해 더미 값을 반환하는 컨텍스트 클래스를 보여 줍니다.

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();
  }

}
```

로깅에 대한 자세한 내용은 AWS Lambda 함수 로깅(Java) 단원을 참조하세요.

샘플 애플리케이션의 컨텍스트

이 안내서의 GitHub 리포지토리에는 컨텍스트 객체의 사용을 보여주는 샘플 애플리케이션이 들어 있습니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS Serverless Application Model(AWS SAM) 템플릿 및 지원 리소스가 포함되어 있습니다.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.

- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

AWS Lambda 함수 로깅(Java)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그 항목을 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 간접 호출에 대한 세부 정보와 함수 코드의 기타 출력을 로그 스트림으로 전송합니다. CloudWatch Logs에 대한 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 섹션을 참조하세요.

함수 코드에서 로그를 출력하려면 java.lang.System의 메서드 또는 stdout 또는 stderr에 쓰는 로깅 모듈을 사용할 수 있습니다.

단원

- 로그를 반환하는 함수 생성
- Java에서 Lambda 고급 로깅 제어 사용
- Log4j2 및 SLF4J를 사용한 고급 로깅
- 기타 도구 및 라이브러리
- 구조화된 로깅에 Powertools for AWS Lambda(Java) 및 AWS SAM 사용
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제
- 샘플 로깅 코드

로그를 반환하는 함수 생성

함수 코드의 로그를 출력하려면 java.lang.System에서 메서드를 사용하거나, `stdout` 또는 `stderr`에 쓰는 로깅 모듈을 사용합니다. aws-lambda-java-core 라이브러리는 컨텍스트 객체에서 액세스 할 수 있는 `LambdaLogger`라는 로거 클래스를 제공합니다. 로거 클래스는 여러 줄 로그를 지원합니다.

다음 예제에서는 컨텍스트 객체가 제공하는 `LambdaLogger` 로거를 사용합니다.

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 로그 형식

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 줄을 로깅합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

Java에서 Lambda 고급 로깅 제어 사용

함수의 로그를 캡처, 처리 및 사용하는 방법을 더 잘 제어할 수 있도록 지원되는 Java 런타임에 대한 다음의 로깅 옵션을 구성할 수 있습니다.

- 로그 형식 - 함수 로그의 경우 일반 텍스트와 구조화된 JSON 형식 중에서 선택
- 로그 수준 - JSON 형식의 로그의 경우, Lambda가 CloudWatch로 전송하는 로그의 세부 수준(ERROR, DEBUG 또는 INFO 등)을 선택
- 로그 그룹 - 함수가 로그를 보내는 CloudWatch 로그 그룹을 선택

이러한 로깅 옵션에 대한 자세한 내용과 이를 사용하도록 함수를 구성하는 방법에 대한 지침은 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”을 참조하세요.

Java Lambda 함수에서 로그 형식 및 로그 수준 옵션을 사용하려면 다음 섹션의 지침을 참조하세요.

Java에서 구조화된 JSON 로그 형식 사용

함수의 로그 형식으로 JSON을 선택하면 Lambda는 `LambdaLogger` 클래스를 사용하여 로그 출력을 구조화된 JSON으로 전송합니다. 각 JSON 로그 객체에는 다음 키가 있는 4개의 키 값 페어가 포함되어 있습니다.

- `"timestamp"` - 로그 메시지가 생성된 시간
- `"level"` - 메시지에 할당된 로그 수준
- `"message"` - 로그 메시지의 내용
- `"AWSrequestId"` - 함수 간접 호출의 고유한 요청 ID

사용하는 로깅 방법에 따라 JSON 형식으로 캡처된 함수의 로그 출력에는 추가 키값 페어가 포함될 수도 있습니다.

`LambdaLogger` 로거를 사용하여 생성한 로그에 수준을 할당하려면 다음 예에서 표시된 것처럼 로깅 명령에 `LogLevel` 인수를 제공해야 합니다.

Example Java 로깅 코드

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

이 예제 코드의 로그 출력은 다음과 같이 CloudWatch Logs에 캡처됩니다.

Example JSON 로그 레코드

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

로그 출력에 수준을 할당하지 않으면 Lambda는 자동으로 수준 INFO를 할당합니다.

코드에서 라이브러리를 사용하여 JSON 구조화된 로그를 생성하는 경우에는 변경할 필요가 없습니다. Lambda는 이미 JSON으로 인코딩된 로그를 이중 인코딩하지 않습니다. JSON 로그 형식을 사용하도록 함수를 구성하더라도 로깅 출력은 사용자가 정의한 JSON 구조로 CloudWatch에 표시됩니다.

Java에서 로그 수준 필터링 사용

AWS Lambda에서 애플리케이션 로그를 로그 수준에 따라 필터링하려면 함수에서 JSON 형식의 로그를 사용해야 합니다. 다음 두 가지 방법으로 이 작업을 달성할 수 있습니다.

- 표준 `LambdaLogger`를 사용하여 로그 출력을 생성하고 JSON 로그 형식을 사용하도록 함수를 구성합니다. 그런 다음 Lambda는 the section called “Java에서 구조화된 JSON 로그 형식 사용”에 설명된 JSON 객체의 “레벨” 키 값 쌍을 사용하여 로그 출력을 필터링합니다. 함수의 로그 형식을 구성하는 방법을 알아보려면 the section called “Lambda 함수에 대한 고급 로깅 제어 구성”를 참조하세요.
- 다른 로깅 라이브러리 또는 메서드를 사용하여 로그 출력 수준을 정의하는 “레벨” 키 값 쌍이 포함된 JSON 구조화된 로그를 코드에 만들 수 있습니다. `stdout` 또는 `stderr`에 JSON 로그를 쓸 수 있는 로깅 라이브러리를 사용할 수 있습니다. 예를 들어 AWS Lambda용 Powertools 또는 Log4j2 패키지를 사용하여 코드로부터 JSON 구조화된 로그 출력을 생성할 수 있습니다. 자세한 내용은 the section called “구조화된 로깅에 Powertools for AWS Lambda(Java) 및 AWS SAM 사용” 및 the section called “Log4j2 및 SLF4J를 사용한 고급 로깅”를 참조하세요.

로그 수준 필터링을 사용하도록 함수를 구성할 때는 Lambda가 CloudWatch 로그로 전송하기를 원하는 로그 수준을 다음 옵션 중에서 선택해야 합니다.

로그 수준	표준 사용량
TRACE(최대 세부 정보)	코드 실행 경로를 추적하는 데 사용되는 가장 세밀한 정보
DEBUG	시스템 디버깅에 대한 세부 정보
INFO	함수의 정상 작동을 기록하는 메시지
WARN	해결되지 않을 경우 예상치 못한 동작으로 이어질 수 있는 잠재적 오류에 대한 메시지
ERROR	코드가 예상대로 작동하지 못하게 하는 문제에 대한 메시지
FATAL(최소 세부 정보)	응용 프로그램 작동을 중지시키는 심각한 오류에 대한 메시지

Lambda가 함수 로그를 필터링하려면 JSON 로그 출력에 "timestamp" 키 값 쌍도 포함해야 합니다. 시간은 유효한 RFC 3339 타임스탬프 형식으로 지정해야 합니다. 유효한 타임스탬프를 제공하지 않으면 Lambda는 로그에 레벨 INFO를 할당하고 타임스탬프를 추가합니다.

Lambda는 선택한 수준 이하의 로그만 Cloudwatch로 전송합니다. 예를 들어 로그 수준을 WARN으로 구성하면 Lambda는 WARN, ERROR 및 FATAL에 해당하는 로그를 전송합니다.

Log4j2 및 SLF4J를 사용한 고급 로깅

> **Note**
>
> AWS Lambda는 관리형 런타임 또는 기본 컨테이너 이미지에 Log4j2를 포함하지 않습니다. 따라서 CVE-2021-44228, CVE-2021-45046 및 CVE-2021-45105에 설명된 문제에는 영향을 받지 않습니다.
> 고객 함수가 영향을 받는 Log4j2 버전전을 포함하는 경우, CVE-2021-44228, CVE-2021-45046 및 CVE-2021-45105의 문제를 완화하는 데 도움이 되는 Lambda Java 관리형 런타임과기본 컨테이너 이미지에 변경 사항을 적용했습니다. 이 변경의 결과로 Log4J2를 사용하는 고객은 'Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)'과 유사한 추가 로그 입력 항목을 볼 수 있습니다. Log4J2 출력에서 jndi 매퍼를 참조하는 모든 로그 문자열은 'Patched JndiLookup::lookup()으로 대체됩니다.
> 이러한 변경과 관계없이 Log4j2가 포함된 함수를 가진 모든 고객은 최신 버전으로 업데이트하는 것이 좋습니다. 특히, 함수에서 aws-lambda-java-log4j2 라이브러리를 사용하는 고객은 버전 1.5.0 이상으로 업데이트하고 함수를 다시 배포해야 합니다. 이 버전은 기본 Log4j2 유틸리티 종속성을 버전 2.17.0 이상으로 업데이트합니다. 업데이트된 aws-lambda-java-log4j2 바이너리는 Maven 리포지토리에서 사용할 수 있으며 소스 코드는 GitHub에서 확인할 수 있습니다.
> 마지막으로 aws-lambda-java-log4j(v1.0.0 또는 1.0.1)와 관련된 모든 라이브러리는 절대 어떤 상황에서도 사용해서는 안 된다는 점에 유의합니다. 이러한 라이브러리는 2015년에 수명이 끝난 log4j의 버전 1.x와(과) 관련이 있습니다. 라이브러리가 지원되지 않고, 유지 관리되지 않으며, 패치가 적용되지 않으며, 알려져 있는 보안 취약성이 있습니다.

로그 출력을 사용자 지정하고, 단위 테스트 중 로깅을 지원하고, AWS SDK 호출을 기록하려면 Apache Log4j2를 SLF4J와 함께 사용합니다. Log4j는 Java 프로그램을 위한 로깅 라이브러리로서, 로그 수준을 구성하고 appender 라이브러리를 사용할 수 있도록 합니다. SLF4J는 facade 라이브러리로서, 함수 코드를 변경하지 않고 사용하는 라이브러리를 변경할 수 있도록 합니다.

함수의 로그에 요청 ID를 추가하려면 aws-lambda-java-log4j2 라이브러리의 appender를 사용합니다.

Example src/main/resources/log4j2.xml – Appender 구성

```
<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>
```

<LambdaTextFormat> 및 <LambdaJSONFormat> 태그 아래에 레이아웃을 지정하여 Log4j2 로그가 일반 텍스트 또는 JSON 출력으로 구성되는 방법을 결정할 수 있습니다.

이 예제에서는 텍스트 모드에서 각 행 앞에 날짜, 시간, 요청 ID, 로그 수준 및 클래스 이름이 추가됩니다. JSON 모드에서는 <JsonTemplateLayout>은 aws-lambda-java-log4j2 라이브러리와 함께 제공되는 구성과 함께 사용됩니다.

SLF4J는 Java 코드에 로깅하기 위한 facade 라이브러리입니다. 함수 코드에서 SLF4J 로거 팩토리를 사용하여 info() 및 warn()과 같은 로그 수준에 대한 메서드로 로거를 가져옵니다. 빌드 구성에서 로깅 라이브러리와 SLF4J 어댑터를 클래스 경로에 포함합니다. 빌드 구성에서 라이브러리를 변경하면 함수 코드를 변경하지 않고 로거 유형을 변경할 수 있습니다. SLF4J는 SDK for Java에서 로그를 캡처하는 데 필요합니다.

다음 예제 코드에서 핸들러 클래스는 SLF4J를 사용하여 로거를 검색합니다.

Example src/main/java/example/HandlerS3.java – SLF4J를 사용한 로깅

```
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";
    }
}
```

이 코드는 다음과 같은 로그 출력을 생성합니다.

Example 로그 형식

```
{
    "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/"
}
```

빌드 구성은 Lambda appender 및 SLF4J 어댑터에 대한 런타임 종속성을 사용하고 Log4j2에 대한 구현 종속성을 사용합니다.

Example build.gradle – 종속성 로깅

```
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,)',
    ...
}
```

테스트를 위해 로컬에서 코드를 실행하면 Lambda 로거가 있는 컨텍스트 객체를 사용할 수 없으며 Lambda appender가 사용할 요청 ID가 없습니다. 테스트 구성에 대한 예는 다음 단원의 샘플 애플리케이션을 참조하세요.

기타 도구 및 라이브러리

Powertools for AWS Lambda(Java)는 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다. Logging 유틸리티는 JSON으로 구조화된 출력과 함께 모든 함수의 함수 컨텍스트에 대한 추가 정보를 포함하는 Lambda 최적화 로거를 제공합니다. 이 유틸리티를 사용하여 다음을 수행합니다.

- Lambda 컨텍스트, 콜드 스타트 및 구조 로깅 출력에서 JSON으로 주요 필드 캡처
- 지시 시 Lambda 호출 이벤트 로깅(기본적으로 비활성화됨)
- 로그 샘플링을 통해 호출 비율에 대해서만 모든 로그 인쇄(기본적으로 비활성화됨)
- 언제든지 구조화된 로그에 추가 키 추가
- 사용자 지정 로그 포맷터(Bring Your Own Formatter)를 사용하여 조직의 로깅 RFC와 호환되는 구조로 로그 출력

구조화된 로깅에 Powertools for AWS Lambda(Java) 및 AWS SAM 사용

다음 단계를 따라 AWS SAM을 사용하는 통합 Powertools for AWS Lambda(Java)~ 모듈을 사용하여 샘플 Hello World Java 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Java 11
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World Java 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name sam-app
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

로그 보존 관리

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 경과 후 CloudWatch가 로그를 자동으로 삭제하는 보존 기간을 구성하세요. 로그 보존을 설정하려면 AWS SAM 템플릿에 다음을 추가합니다.

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - aws configure을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 `base64` 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 `my-function`에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한

내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```


로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

샘플 로깅 코드

이 안내서의 GitHub 리포지토리에는 다양한 로깅 구성의 사용을 보여주는 샘플 애플리케이션이 들어 있습니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS SAM 템플릿 및 지원 리소스가 포함되어 있습니다.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.
- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

`java-basic` 샘플 애플리케이션은 로깅 테스트를 지원하는 최소 로깅 구성을 보여 줍니다. 핸들러 코드는 컨텍스트 객체가 제공하는 `LambdaLogger` 로거를 사용합니다. 애플리케이션은 테스트를 위해

Log4j2 로거와 함께 `TestLogger` 인터페이스를 구현하는 사용자 지정 `LambdaLogger` 클래스를 사용합니다. AWS SDK와의 호환성을 위한 facade로 SLF4J를 사용합니다. 로깅 라이브러리는 배포 패키지를 작게 유지하기 위해 빌드 출력에서 제외되어 있습니다.

AWS Lambda에서 Java 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 두 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS Distro for OpenTelemetry(ADOT) - 안전하게 프로덕션 준비가 된 AWS에서 지원하는 OpenTelemetry(OTEL) SDK의 배포입니다.
- AWS X-Ray SDK for Java — 추적 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.
- Powertools for AWS Lambda(Java) - 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 어떤 솔루션을 사용하든 X-Ray에서 엔드 투 엔드 추적 기능을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

- AWS CloudFormation을 사용하여 추적 활성화
- X-Ray 추적 해석
- 계층에 런타임 종속성 저장(X-Ray SDK)
- 샘플 애플리케이션에서 X-Ray 추적(X-Ray SDK)

추적에 Powertools for AWS Lambda(Java) 및 AWS SAM 사용

다음 단계를 따라 AWS SAM를 사용하는 통합 Powertools for AWS Lambda(Java) 모듈을 사용하여 샘플 Hello World Java 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Java 11
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World Java 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

   > **Note**
   >
   > HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

   ```
   aws cloudformation describe-stacks --stack-name sam-app --query
    'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
   ```

6. API 엔드포인트 호출:

   ```
   curl -X GET <URL_FROM_PREVIOUS_STEP>
   ```

   성공하면 다음과 같은 결과가 응답됩니다.

   ```
   {"message":"hello world"}
   ```

7. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

   ```
   sam traces
   ```

   추적 출력은 다음과 같습니다.

   ```
   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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

추적에 Powertools for AWS Lambda(Java) 및 AWS CDK 사용

다음 단계를 따라 AWS CDK를 사용하는 통합 Powertools for AWS Lambda(Java) 모듈을 사용하여 샘플 Hello World Java 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 함수가 hello world 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Java 11
- AWS CLI 버전 2
- AWS CDK 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS CDK 애플리케이션 배포

1. 새 애플리케이션용 프로젝트 디렉터리를 생성합니다.

```
mkdir hello-world
cd hello-world
```

2. 앱을 초기화합니다.

```
cdk init app --language java
```

3. 다음 명령을 사용하여 maven 프로젝트를 생성합니다.

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. `hello-world\app\Function` 디렉터리에서 `pom.xml`을 열고 기존 코드를 Powertools에 대한 종속 구성 요소 및 maven 플러그인을 포함하는 다음 코드로 바꿉니다.

```
<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 디렉터리를 생성하고 로그 구성에 대한 log4j.xml을 생성합니다.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. log4j.xml을 열고 다음 코드를 추가합니다.

```
<?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 디렉터리에서 App.java를 열고 기존 코드를 다음 코드로 바꿉니다. Lambda 함수에 대한 코드입니다.

```
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 디렉터리에서 HelloWorldStack.java를 열고 기존 코드를 다음 코드로 바꿉니다. 이 코드는 Lambda Constructor와 ApiGatewayv2 Constructor를 사용하여 REST API와 Lambda 함수를 생성합니다.

```
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 디렉터리에서 pom.xml를 열고 기존 코드를 다음 코드로 바꿉니다.

```
<?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` 디렉터리에 있는지 확인하고 애플리케이션을 배포합니다.

```
cdk deploy
```

11. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. API 엔드포인트 호출:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

13. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

추적 출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
cdk destroy
```

ADOT를 사용하여 Java 함수 계측

ADOT는 OTel SDK를 사용하여 원격 측정 데이터를 수집하는 데 필요한 모든 것을 패키징할 수 있는 완전 관리형 Lambda 계층을 제공합니다. 이 계층을 사용하면 모든 함수 코드를 수정하지 않고도

Lambda 함수를 계측할 수 있습니다. 계층을 구성하여 OTel의 사용자 지정 초기화를 수행할 수도 있습니다. 자세한 내용은 ADOT 설명서의 Lambda에서 ADOT 컬렉터에 대한 사용자 지정 구성을 참조하세요.

Java 런타임의 경우 두 계층 중에서 선택하여 사용할 수 있습니다.

- AWS ADOT Java용 관리형 Lambda 계층(자동 계측 에이전트) - 이 계층은 시작 시 함수 코드를 자동으로 변환하여 추적 데이터를 수집합니다. ADOT Java 에이전트와 함께 이 계층을 사용하는 방법에 대한 자세한 지침은 ADOT 설명서의 AWS Distro for OpenTelemetry Lambda Support for Java (Auto-instrumentation Agent)(Java용 Distro for OpenTelemetry Lambda Support(자동 계측 에이전트))를 참조하세요.
- AWS ADOT Java용 관리형 Lambda 계층 - 이 계층은 Lambda 함수에 대한 기본 제공 계측도 제공하지만, OTel SDK를 초기화하려면 몇 가지 코드를 직접 변경해야 합니다. 이 계층을 사용하는 방법에 대한 자세한 지침은 ADOT 설명서의 AWS Distro for OpenTelemetry Lambda Support for Java(Java용 Distro for OpenTelemetry Lambda Support)를 참조하세요.

X-Ray SDK를 사용하여 Java 함수 계측

함수가 애플리케이션의 다른 리소스 및 서비스에 대해 수행하는 호출과 관련된 데이터를 기록하려면 Java용 X-Ray SDK를 빌드 구성에 추가합니다. 다음 예에서는 AWS SDK for Java 2.x 클라이언트의 자동 계측을 활성화하는 라이브러리를 포함하는 Gradle 빌드 구성을 보여줍니다.

Example build.gradle – 종속성 추적

```
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'
    ...
}
```

올바른 종속성을 추가하고 필요한 코드를 변경한 후 Lambda 콘솔 또는 API를 통해 함수의 구성에서 추적을 활성화합니다.

Lambda 콘솔을 사용하여 추적 활성화

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

Lambda API를 사용하여 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 Lambda 함수에 대한 추적을 구성하고 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation을 사용하여 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
```

```
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray 추적 해석

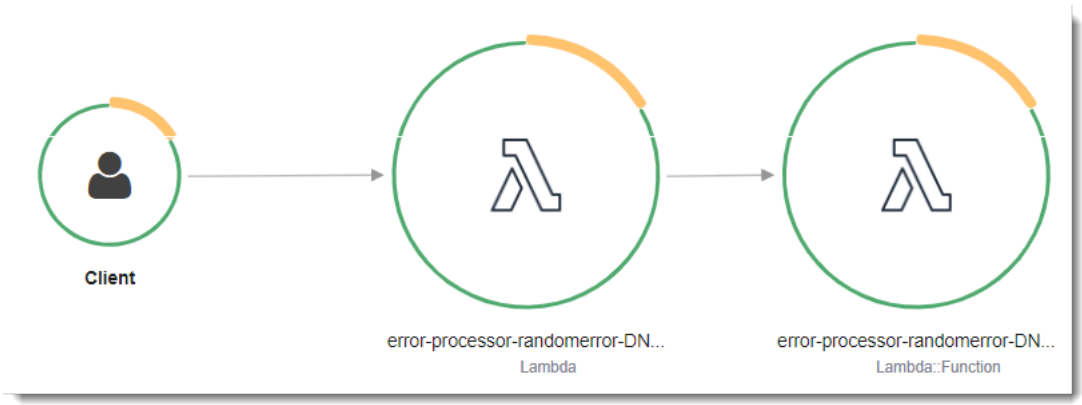
함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

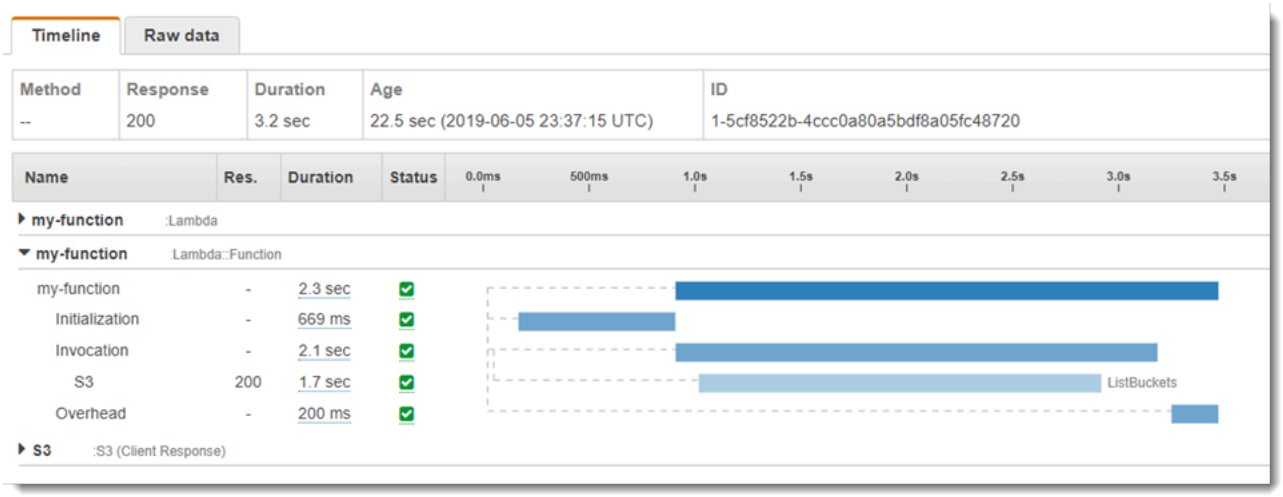
> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이

AWS::Lambda::Function입니다. AWS::Lambda 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. AWS::Lambda::Function 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 AWS::Lambda::Function 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

> **Note**
>
> Lambda SnapStart 함수에는 Restore 하위 세그먼트도 포함됩니다. Restore 하위 세그먼트는 Lambda가 스냅샷을 복원하고, 런타임(JVM)을 로드하고, afterRestore 런타임 후크를 실행하는 데 걸리는 시간을 보여줍니다. 스냅샷 복원 프로세스에는 microVM 외부 작업에 소요되는 시간이 포함될 수 있습니다. 이 시간은 Restore 하위 세그먼트에서 보고됩니다. 스냅샷을 복원하기 위해 microVM 외부에서 소요된 시간에 대한 요금은 부과되지 않습니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 AWS X-Ray SDK for Java을 참조하세요.

> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

계층에 런타임 종속성 저장(X-Ray SDK)

X-Ray SDK를 사용하여 AWS SDK 클라이언트를 계측하는 경우 함수 코드와 배포 패키지가 상당히 커질 수 있습니다. 함수 코드를 업데이트할 때마다 런타임 종속성을 업로드하지 않으려면 X-Ray SDK를 Lambda 계층에 패키징합니다.

다음 예에서는 AWS SDK for Java 및 Java용 X-Ray SDK를 저장하는 `AWS::Serverless::LayerVersion` 리소스를 보여줍니다.

Example template.yml – 종속성 계층

```
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
```

이 구성을 사용하면 런타임 종속성을 변경하는 경우 라이브러리 계층만 업데이트하면 됩니다. 함수 배포 패키지에는 코드만 포함되어 있으므로 이는 업로드 시간을 줄일 수 있습니다.

종속성 계층을 만들려면 배포 전에 계층 아카이브를 생성하기 위해 빌드 구성을 변경해야 합니다. 사용 가능한 예는 GitHub의 java-basic 샘플 애플리케이션을 참조하세요.

샘플 애플리케이션에서 X-Ray 추적(X-Ray SDK)

이 안내서의 GitHub 리포지토리에는 X-Ray 추적의 사용을 보여주는 샘플 애플리케이션이 들어 있습니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS SAM 템플릿 및 지원 리소스가 포함되어 있습니다.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.
- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

모든 샘플 애플리케이션은 Lambda 함수에 대해 활성 추적을 사용하도록 설정되어 있습니다. `s3-java` 애플리케이션은 AWS SDK for Java 2.x 클라이언트의 자동 계측, 테스트를 위한 세그먼트 관리, 사용자 지정 하위 세그먼트 및 런타임 종속성을 저장하기 위한 Lambda 계층의 사용을 보여줍니다.

AWS Lambda용 Java 샘플 애플리케이션

이 안내서의 GitHub 리포지토리는 AWS Lambda에서 Java를 사용하는 방법을 보여주는 샘플 애플리케이션을 제공합니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS CloudFormation 템플릿 및 지원 리소스가 포함되어 있습니다.

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.
- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

Lambda에서 인기 있는 Java 프레임워크 실행

- spring-cloud-function-samples – Spring Cloud 함수 프레임워크를 사용하여 AWS Lambda 함수를 생성하는 방법을 보여주는 Spring 예제입니다.
- 서버리스 Spring Boot 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 일반적인 Spring Boot 애플리케이션을 설정하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다.
- 서버리스 Micronaut 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 Micronaut을 사용하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다. Micronaut/Lambda 설명서에서 자세히 알아보세요.
- 서버리스 Quarkus 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 Quarkus를 사용하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다. Quarkus/Lambda 설명서 및 Quarkus/SnapStart 설명서에서 자세히 알아보세요.

Java에서 Lambda 함수를 처음 사용하는 경우 `java-basic` 예제부터 시작합니다. Lambda 이벤트 소스를 시작하려면 `java-events` 예제를 참조하세요. 이 두 예제 세트에서는 모두 Lambda의 Java 라이브러리, 환경 변수, AWS SDK 및 AWS X-Ray SDK의 사용을 보여줍니다. 이러한 예제에는 최소한의 설정이 필요하며 1분 이내에 명령줄에서 배포할 수 있습니다.

Go를 사용하여 Lambda 함수 빌드

Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 런타임(`provided` 런타임 패밀리)을 사용하여 Lambda에 Go 함수를 배포합니다.

주제

- Go 런타임 지원
- 도구 및 라이브러리
- Go에서 Lambda 함수 핸들러 정의
- AWS Lambda 컨텍스트 객체(Go)
- .zip 파일 아카이브를 사용하여 Go Lambda 함수 배포
- 컨테이너 이미지로 Go Lambda 함수 배포
- AWS Lambda 함수 로깅(Go)
- AWS Lambda에서 Go 코드 계측
- 환경 변수 사용

Go 런타임 지원

Lambda용 Go 1.x 관리형 런타임은 더 이상 사용되지 않습니다. Go 1.x 런타임을 사용하는 함수가 있는 경우 `provided.al2023` 또는 `provided.al2`로 마이그레이션해야 합니다. `provided.al2023` 및 `provided.al2` 런타임은 `go1.x`보다 뛰어난 몇 가지 이점을 제공하며, 여기에는 arm64 아키텍처(AWS Graviton2 프로세서), 더 작은 바이너리 및 약간 더 빠른 간접 호출 시간에 대한 지원이 해당됩니다.

이 마이그레이션에는 코드 변경이 필요 없습니다. 유일한 필수 변경 사항은 배포 패키지를 빌드하는 방법과 함수를 생성하는 데 사용하는 런타임과 관련이 있습니다. 자세한 내용은 AWS 컴퓨팅 블로그에서 Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2를 참조하세요.

OS 전용

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
OS 전용 런타임	`provided.al2023`	Amazon Linux 2023			
OS 전용 런타임	`provided.al2`	Amazon Linux 2			

도구 및 라이브러리

Lambda에서는 다음과 같은 Go 런타임용 도구 및 라이브러리를 제공합니다.

- AWS SDK for Go: Go 프로그래밍 언어용 공식 AWS SDK입니다.
- github.com/aws/aws-lambda-go/lambda: Go용 Lambda 프로그래밍 모델을 구현한 것입니다. 이 패키지는 AWS Lambda에서 핸들러를 호출하기 위해 사용됩니다.
- github.com/aws/aws-lambda-go/lambdacontext: 컨텍스트 객체에서 컨텍스트 정보에 액세스하기 위한 헬퍼입니다.
- github.com/aws/aws-lambda-go/events: 이 라이브러리는 일반적인 이벤트 소스 통합을 위한 유형 정의를 제공합니다.
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip: 이 도구를 사용하여 Windows에서 .zip 파일 아카이브를 생성할 수 있습니다.

자세한 내용은 GitHub의 aws-lambda-go를 참조하세요.

Lambda에서는 다음과 같은 Go 런타임용 샘플 애플리케이션을 제공합니다.

Go의 샘플 Lambda 애플리케이션

- go-al2 – 퍼블릭 IP 주소를 반환하는 hello world 함수입니다. 이 앱은 `provided.al2` 사용자 지정 런타임을 사용합니다.
- blank-go – Lambda의 Go 라이브러리, 로깅, 환경 변수 및 AWS SDK를 사용하는 방법을 보여주는 Go 함수입니다. 이 앱은 `go1.x` 런타임을 사용합니다.

Go에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

Go에서 작성된 Lambda 함수는 Go 실행 파일로 작성됩니다. Lambda 함수 코드에서는 github.com/aws/aws-lambda-go/lambda 패키지를 포함해야 합니다. 이 패키지는 Go용 Lambda 프로그래밍 모델을 구현합니다. 또한 핸들러 함수 코드와 `main()` 함수를 구현해야 합니다.

Example Lambda 함수로 이동

```
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)
}
```

다음은 이 함수에 대한 샘플 입력입니다.

```
{
  "name": "Jane"
}
```

유의할 사항:

- package main: Go에서 `func main()`을 포함하는 패키지의 이름은 항상 `main`으로 지정해야 합니다.
- import: 이 명령을 사용하면 Lambda 함수에 필요한 라이브러리를 포함시킬 수 있습니다. 이 인스턴스에서는 다음을 포함합니다.
  - context: AWS Lambda 컨텍스트 객체(Go).
  - fmt: 함수의 반환 값 형식을 지정하는 데 사용되는 Go 서식 객체입니다.
  - github.com/aws/aws-lambda-go/lambda: 앞서 언급한 것처럼 Go용 Lambda 프로그래밍 모델을 구현합니다.
- func HandleRequest(ctx context.Context, event *MyEvent) (*string, error): 이것은 Lambda 핸들러 서명입니다. Lambda 함수의 진입점으로 함수가 호출될 때 실행되는 로직을 포함합니다. 또한 포함된 파라미터들은 다음을 나타냅니다.
  - ctx context.Context: Lambda 함수 호출을 위한 런타임 정보를 제공합니다. `ctx`는 AWS Lambda 컨텍스트 객체(Go)를 통해 사용 가능한 정보를 활용하기 위해 선언하는 변수입니다.
  - event *MyEvent: `MyEvent`를 가리키는 `event`라는 이름의 파라미터입니다. Lambda 함수에 대한 입력을 나타냅니다.
  - *string, error: 핸들러가 두 개의 값을 반환합니다. 첫 번째는 Lambda 함수의 결과를 포함하는 문자열에 대한 포인터입니다. 두 번째는 오류 유형으로 오류가 없는 경우 `nil`이며 문제가 발생할 경우 표준 오류 정보를 포함합니다.
  - return &message, nil: 두 개의 값을 반환합니다. 첫 번째는 문자열 메시지에 대한 포인터로 입력 이벤트의 `Name` 필드를 사용하여 구성된 인사말입니다. 두 번째 값인 `nil`은 함수에 오류가 발생하지 않았음을 나타냅니다.
- func main(): Lambda 함수 코드를 실행하는 진입점입니다. 이 항목은 필수입니다.

  `func main(){}` 코드 대괄호 사이에 `lambda.Start(HandleRequest)`를 추가하면 Lambda 함수가 실행됩니다. Go 언어 표준에 따라 여는 대괄호({)는 `main` 함수 서명의 끝에 직접 배치해야 합니다.

이름 지정

provided.al2 및 provided.al2023 런타임

.zip 배포 패키지에서 `provided.al2` 또는 `provided.al2023` 런타임을 사용하는 Go 함수의 경우 함수 코드가 포함된 실행 파일의 이름을 `bootstrap`으로 지정해야 합니다. .zip 파일로 함

수를 배포하는 경우 bootstrap 파일이 .zip 파일의 루트에 있어야 합니다. 컨테이너 이미지에서 provided.al2 또는 provided.al2023 런타임을 사용하는 Go 함수의 경우 실행 파일에 모든 이름을 사용할 수 있습니다.

핸들러에는 아무 이름이나 사용할 수 있습니다. 코드에서 핸들러 값을 참조하려면 _HANDLER 환경 변수를 사용합니다.

go1.x runtime

go1.x 런타임을 사용하는 Go 함수의 경우 실행 파일과 핸들러는 어떤 이름이든 공유할 수 있습니다. 예를 들어 핸들러 값을 Handler로 설정하면 Lambda는 Handler 실행 파일에서 main() 함수를 호출합니다.

Lambda 콘솔에서 함수 핸들러 이름을 변경하려면 런타임 설정 창에서 편집을 선택합니다.

구조화된 유형을 이용한 Lambda 함수 핸들러

위 예제에서 입력 유형은 단순한 문자열이었습니다. 다만 구조화된 이벤트에서 다음과 같이 함수 핸들러로 전달할 수도 있습니다.

```
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)
}
```

다음은 이 함수에 대한 샘플 입력입니다.

```
{
    "What is your name?": "Jim",
    "How old are you?": 33
}
```

응답은 다음과 같습니다.

```
{
    "Answer": "Jim is 33 years old!"
}
```

내보내기를 위해서는 이벤트 구문의 필드 이름이 대문자로 표시되어야 합니다. AWS 이벤트 소스의 이벤트를 처리하는 것에 대한 자세한 내용은 aws-lambda-go/events를 참조하세요.

유효한 핸들러 서명

Go에서 Lambda 함수 핸들러를 빌드할 때 몇 가지 옵션들이 있으며 다만 다음과 같은 규칙들을 반드시 준수해야 합니다.

- 핸들러는 하나의 함수여야 합니다.
- 핸들러는 0~2개의 인수를 취할 수 있습니다. 인수가 2개일 경우, 첫 번째 인수는 `context.Context`를 구현해야 합니다.
- 핸들러는 0~2개의 인수를 반환할 수 있습니다. 반환 값이 1개일 경우, 이 값은 `error`를 구현해야 합니다. 반환 값이 2개일 경우, 두 번째 값은 `error`를 구현해야 합니다.

다음 목록은 유효한 핸들러 서명을 열거합니다. `TIn` 및 `TOut`은 encoding/json 표준 라이브러리와 호환 가능한 유형을 나타냅니다. 이러한 유형들이 역직렬화되는 방식에 관한 자세한 내용은 func Unmarshal을 참조하시기 바랍니다.

-
```
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)`

전역 상태 사용

Lambda 함수의 핸들러 코드와 무관한 전역 변수들을 선언하고 수정해야 합니다. 또한 핸들러는 그것이 로드될 때 실행되는 `init` 함수를 선언할 수 있습니다. 이 함수는 표준 Go 프로그램과 마찬가지로 AWS Lambda에서도 동일하게 동작합니다. Lambda 함수의 단일 인스턴스는 여러 이벤트를 동시에 처리하지 못합니다.

Example 글로벌 변수가 있는 Go 함수

> **Note**
>
> 이 코드는 AWS SDK for Go V2를 사용합니다. 자세한 내용은 AWS SDK for Go V2 사용 시작하기를 참조하세요.

```
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)
}
```

AWS Lambda 컨텍스트 객체(Go)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 메서드 및 속성에 제공합니다.

Lambda 컨텍스트 라이브러리는 다음과 같은 전역 변수, 메서드 및 속성들을 제공합니다.

전역 변수

- `FunctionName` – Lambda 함수의 이름입니다.
- `FunctionVersion` – 함수의 버전입니다.
- `MemoryLimitInMB` – 함수에 할당된 메모리의 양입니다.
- `LogGroupName` – 함수에 대한 로그 그룹입니다.
- `LogStreamName` – 함수 인스턴스에 대한 로그 스트림입니다.

컨텍스트 메서드

- `Deadline` – 실행 시간이 초과된 날짜를 Unix 시간 형식에 따른 밀리초 단위로 반환합니다.

컨텍스트 속성

- `InvokedFunctionArn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `AwsRequestID` – 호출 요청의 식별자입니다.
- `Identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
- `ClientContext` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.

호출 컨텍스트 정보 액세스

Lambda 함수들은 자신의 환경과 호출 요청에 관한 메타데이터에 액세스할 수 있습니다. 이는 Package context에서 액세스할 수 있습니다. 핸들러가 `context.Context`를 하나의 파라미터로 포함할 경우, Lambda는 함수에 관한 정보를 컨텍스트의 `Value` 속성에 삽입합니다. `lambdacontext` 객체의 콘텐츠에 액세스하려면 `context.Context` 라이브러리를 가져와야 합니다.

```
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)
}
```

위 예제에서 `lc` 은 컨텍스트 객체가 캡처한 정보를 소비하고 해당 정보 (이 경우에는 CognitoIdentityPool ID) 를 `log.Print(lc.Identity.CognitoIdentityPoolID)` 인쇄하는 데 사용되는 변수입니다.

다음 예제는 컨텍스트 객체를 사용하여 Lambda 함수가 완료되는 데 걸리는 시간을 모니터링하는 방법을 소개합니다. 이 방법을 활용하면 기대 성능을 분석할 수 있으며 필요에 따라 함수 코드를 조정할 수 있습니다.

```
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 파일 아카이브를 사용하여 Go Lambda 함수 배포

AWS Lambda 함수의 코드는 스크립트 또는 컴파일된 프로그램과 해당 종속 항목으로 구성됩니다. 함수 코드는 배포 패키지를 사용하여 Lambda에 배포합니다. Lambda는 컨테이너 이미지 및 .zip 파일 아카이브의 두 가지 배포 패키지를 지원합니다.

이 페이지에서는 Go 런타임의 배포 패키지로 .zip 파일을 생성한 다음 .zip 파일을 사용하여 AWS Management Console, AWS Command Line Interface(AWS CLI) 및 AWS Serverless Application Model(AWS SAM)에서 AWS Lambda에 함수 코드를 배포하는 방법을 설명합니다.

Lambda는 POSIX 파일 권한을 사용하므로 .zip 파일 아카이브를 생성하기 전에 배포 패키지 폴더에 대한 권한을 설정해야 할 수 있습니다.

Sections

macOS 및 Linux에서 .zip 파일 만들기

다음 단계에서는 `go build` 명령을 사용하여 실행 파일을 컴파일하고 Lambda용 .zip 파일 배포 패키지를 생성하는 방법을 보여줍니다. 코드를 컴파일하기 전에 GitHub에서 lambda 패키지를 설치했는지 확인하세요. 이 모듈은 Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스의 구현을 제공합니다. 이 라이브러리를 다운로드하려면 다음 명령을 실행합니다.

```
go get github.com/aws/aws-lambda-go/lambda
```

함수에서 AWS SDK for Go를 사용하는 경우 애플리케이션에 필요한 AWS 서비스 API 클라이언트와 함께 표준 SDK 모듈 세트를 다운로드하세요. SDK for Go를 설치하는 방법을 알아보려면 Getting Started with the AWS SDK for Go V2를 참조하세요.

제공된 런타임 패밀리 사용

Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 런타임(`provided` 런타임 패밀리)을 사용하여 Lambda에 Go 함수를 배포합니다.

.zip 배포 패키지 생성(macOS/Linux)

1. 애플리케이션의 `main.go` 파일이 들어 있는 프로젝트 디렉터리에서 실행 파일을 컴파일합니다. 유의할 사항:

   - 실행 파일의 이름은 `bootstrap`이어야 합니다. 자세한 내용은 이름 지정 단원을 참조하십시오.
   - 대상 명령 세트 아키텍처를 설정합니다. OS 전용 런타임은 arm64와 x86_64를 모두 지원합니다.
   - 선택 사항인 `lambda.norpc` 태그를 사용하여 lambda 라이브러리의 원격 절차 호출(RPC) 구성 요소를 제외할 수 있습니다. RPC 구성 요소는 지원 중단된 Go 1.x 런타임을 사용할 경우에만 필요합니다. RPC를 제외하면 배포 패키지의 크기가 줄어듭니다.

   arm64 아키텍처의 경우:

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   x86_64 아키텍처의 경우:

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (선택 사항) Linux에서 `CGO_ENABLED=0` 세트를 사용하여 패키지를 컴파일해야 할 수도 있습니다.

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   이 명령은 표준 C 라이브러리(`libc`) 버전에 대해 안정적인 바이너리 패키지를 만듭니다. 이 패키지는 다른 Lambda 및 다른 장치에서 다를 수 있습니다.

3. 실행 파일을 .zip 파일로 패키지하여 배포 패키지를 만듭니다.

   ```
   zip myFunction.zip bootstrap
   ```

   > **Note**
   >
   > `bootstrap` 파일은 .zip 파일의 루트에 있어야 합니다.

4. 함수를 생성합니다. 유의할 사항:

- 바이너리의 이름은 `bootstrap`이어야 하지만 핸들러 이름은 무엇이든 지정할 수 있습니다. 자세한 내용은 이름 지정 단원을 참조하십시오.
- `--architectures` 옵션은 arm64를 사용할 때만 필요합니다. 기본값은 x86_64입니다.
- `--role`에 실행 역할의 Amazon 리소스 이름(ARN)을 지정합니다.

```
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
```

Windows에서 .zip 파일 만들기

다음 단계에서는 GitHub에서 Windows용 build-lambda-zip 도구를 다운로드하고, 실행 파일을 컴파일하고, .zip 배포 패키지를 생성하는 방법을 보여줍니다.

> **Note**
>
> 설치를 아직 수행하지 않았다면 git를 설치한 후 `git` 실행 파일을 Windows `%PATH%` 환경 변수에 추가해야 합니다.

코드를 컴파일하기 전에 GitHub에서 lambda 라이브러리를 설치했는지 확인하세요. 이 라이브러리를 다운로드하려면 다음 명령을 실행합니다.

```
go get github.com/aws/aws-lambda-go/lambda
```

함수에서 AWS SDK for Go를 사용하는 경우 애플리케이션에 필요한 AWS 서비스 API 클라이언트와 함께 표준 SDK 모듈 세트를 다운로드하세요. SDK for Go를 설치하는 방법을 알아보려면 Getting Started with the AWS SDK for Go V2를 참조하세요.

제공된 런타임 패밀리 사용

Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 런타임(`provided` 런타임 패밀리)을 사용하여 Lambda에 Go 함수를 배포합니다.

.zip 배포 패키지 생성(Windows)

1. GitHub에서 build-lambda-zip 도구를 다운로드합니다.

```
go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
```

2. GOPATH의 도구를 사용하여 .zip 파일을 만듭니다. Go가 기본적으로 설치되어 있다면 도구는 보통 %USERPROFILE%\Go\bin에 있습니다. 그렇지 않다면 Go 런타임을 설치한 위치로 이동해 다음 중 하나를 수행합니다.

   cmd.exe

   cmd.exe에서 대상 명령 세트 아키텍처에 따라 다음 중 하나를 실행합니다. OS 전용 런타임은 arm64와 x86_64를 모두 지원합니다.

   선택 사항인 `lambda.norpc` 태그를 사용하여 lambda 라이브러리의 원격 절차 호출(RPC) 구성 요소를 제외할 수 있습니다. RPC 구성 요소는 지원 중단된 Go 1.x 런타임을 사용할 경우에만 필요합니다. RPC를 제외하면 배포 패키지의 크기가 줄어듭니다.

   Example - x86_64 아키텍처의 경우

```
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 아키텍처의 경우

```
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

   PowerShell에서 대상 명령 세트 아키텍처에 따라 다음 중 하나를 실행합니다. OS 전용 런타임은 arm64와 x86_64를 모두 지원합니다.

선택 사항인 `lambda.norpc` 태그를 사용하여 lambda 라이브러리의 원격 절차 호출(RPC) 구성 요소를 제외할 수 있습니다. RPC 구성 요소는 지원 중단된 Go 1.x 런타임을 사용할 경우에만 필요합니다. RPC를 제외하면 배포 패키지의 크기가 줄어듭니다.

x86_64 아키텍처의 경우:

```
$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 아키텍처의 경우:

```
$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. 함수를 생성합니다. 유의할 사항:

   - 바이너리의 이름은 `bootstrap`이어야 하지만 핸들러 이름은 무엇이든 지정할 수 있습니다. 자세한 내용은 이름 지정 단원을 참조하십시오.
   - `--architectures` 옵션은 arm64를 사용할 때만 필요합니다. 기본값은 x86_64입니다.
   - `--role`에 실행 역할의 Amazon 리소스 이름(ARN)을 지정합니다.

```
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 파일을 사용하여 Go Lambda 함수 생성 및 업데이트

.zip 배포 패키지를 생성한 후에는 이를 사용하여 새 Lambda 함수를 생성하거나 기존 함수를 업데이트할 수 있습니다. Lambda 콘솔, AWS Command Line Interface 및 Lambda API를 사용하여 .zip 패키지

를 배포할 수 있습니다. AWS Serverless Application Model(AWS SAM) 및 AWS CloudFormation을 사용하여 Lambda 함수를 생성하고 업데이트할 수도 있습니다.

Lambda용 .zip 배포 패키지의 최대 크기는 250MB(압축 해제됨)입니다. 이 제한은 Lambda 계층을 포함하여 업로드하는 모든 파일의 합산 크기에 적용됩니다.

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 chmod 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

콘솔을 사용하여.zip 파일로 함수 생성 및 업데이트

새 함수를 생성하려면 먼저 콘솔에서 함수를 생성한 다음.zip 아카이브를 업로드해야 합니다. 기존 함수를 업데이트하려면 함수에 대한 페이지를 연 다음 동일한 절차에 따라 업데이트된 .zip 파일을 추가합니다.

.zip 파일이 50MB 미만인 경우 로컬 컴퓨터에서 직접 파일을 업로드하여 함수를 생성하거나 업데이트할 수 있습니다. 50MB보다 큰 .zip 파일의 경우 먼저 패키지를 Amazon S3 버킷에 업로드해야 합니다. AWS Management Console을 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 Amazon S3 시작하기를 참조하세요. AWS CLI를 사용하여 파일을 업로드하려면 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> .zip 아카이브를 사용하도록 기존 컨테이너 이미지 함수를 변환할 수는 없습니다. 새로운 함수를 생성해야 합니다.

새 함수 생성(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수 생성을 선택합니다.
2. 새로 작성을 선택합니다.

3. 기본 정보에서 다음과 같이 합니다.

   a. 함수 이름에 함수 이름을 입력합니다.

   b. 런타임에서 `provided.al2023`를 선택합니다.

4. (선택 사항) 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 확장합니다. 새로운 실행 역할을 생성하거나 기존 실행 역할을 사용할 수 있습니다.

5. 함수 생성을 선택합니다. Lambda에서 선택한 런타임을 사용하여 기본 'Hello World' 함수를 생성합니다.

로컬 시스템에서 .zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서.zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. .zip 파일을 선택합니다.
5. .zip 파일을 업로드하려면 다음을 수행합니다.

   a. 업로드를 선택한 다음 파일 선택기에서.zip 파일을 선택합니다.

   b. Open을 선택합니다.

   c. Save(저장)를 선택합니다.

Amazon S3 버킷에서.zip 아카이브 업로드(콘솔)

1. Lambda 콘솔의 함수 페이지에서 새 .zip 파일을 업로드할 함수를 선택합니다.
2. 코드 탭을 선택합니다.
3. 코드 소스 창에서 에서 업로드를 선택합니다.
4. Amazon S3 위치를 선택합니다.
5. .zip 파일의 Amazon S3 링크 URL을 붙여 넣고 저장을 선택합니다.

AWS CLI를 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CLI를 사용하여 새 함수를 생성하거나.zip 파일로 기존 함수를 업데이트할 수 있습니다. create-function 및 update-function-code 명령을 사용하여 .zip 패키지를 배포합니다. .zip 파일이 50MB보다 작은 경우 로컬 빌드 시스템의 파일 위치에서 .zip 패키지를 업로드할 수 있습니다. 더 큰 파일의 경우

Amazon S3 버킷에서 .zip 패키지를 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

> **Note**
>
> AWS CLI를 사용하여 Amazon S3 버킷에서 .zip 파일을 업로드하는 경우 버킷은 함수와 동일한 AWS 리전에 있어야 합니다.

AWS CLI에서 .zip 파일을 사용하여 새 함수를 생성하려면 다음을 지정해야 합니다.

- 함수의 이름(`--function-name`)
- 함수의 런타임(`--runtime`)
- 함수의 실행 역할(`--role`)의 Amazon 리소스 이름(ARN)
- 함수 코드에 있는 핸들러 메서드의 이름(`--handler`)

.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
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 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--code` 옵션을 사용합니다. 버전이 지정된 객체에만 `S3ObjectVersion` 파라미터를 사용해야 합니다.

```
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
```

CLI를 사용하여 기존 함수를 업데이트하려면 `--function-name` 파라미터를 사용하여 함수 이름을 지정합니다. 함수 코드를 업데이트하는 데 사용할.zip 파일의 위치도 지정해야 합니다. .zip 파일이 로컬 빌드 시스템의 폴더에 있는 경우 다음 예제 명령과 같이 `--zip-file` 옵션을 사용하여 파일 경로를 지정합니다.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

Amazon S3 버킷에서 .zip 파일의 위치를 지정하려면 다음 예제 명령과 같이 `--s3-bucket` 및 `--s3-key` 옵션을 사용합니다. 버전이 지정된 객체에만 `--s3-object-version` 파라미터를 사용해야 합니다.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket DOC-EXAMPLE-BUCKET --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Lambda API를 사용하여.zip 파일로 함수 생성 및 업데이트

.zip 파일 아카이브를 사용하여 함수를 생성하고 업데이트하려면 다음 API 작업을 사용합니다.

- CreateFunction
- UpdateFunctionCode

AWS SAM을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS Serverless Application Model(AWS SAM)은 AWS에서 서버리스 애플리케이션을 빌드하고 실행하는 프로세스를 간소화하는 데 도움이 되는 도구 키트입니다. YAML 또는 JSON 템플릿에서 애플리케이션의 리소스를 정의하고 AWS SAM Command Line Interface(AWS SAM CLI)를 사용하여 애플리케이션을 빌드, 패키징 및 배포합니다. AWS SAM 템플릿에서 Lambda 함수를 빌드하면 AWS SAM은 함수 코드와 사용자가 지정하는 종속 항목을 사용하여 .zip 배포 패키지 또는 컨테이너 이미지를 자동으로 생성합니다. AWS SAM을 사용하여 Lambda 함수를 빌드하고 배포하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Getting started with AWS SAM을 참조하세요.

AWS SAM을 사용하여 기존 .zip 파일 아카이브로 Lambda 함수를 생성할 수도 있습니다. AWS SAM을 사용하여 Lambda 함수를 생성하려면 Amazon S3 버킷 또는 빌드 시스템의 로컬 폴더에 .zip 파일을 저장할 수 있습니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS SAM 템플릿에서 `AWS::Serverless::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `CodeUri` - 함수 코드의 Amazon S3 URI, 로컬 폴더 경로 또는 FunctionCode 객체로 설정됨
- `Runtime` - 선택한 런타임으로 설정됨

AWS SAM을 사용하면 .zip 파일이 50MB보다 큰 경우 Amazon S3 버킷에 먼저 파일을 업로드할 필요가 없습니다. AWS SAM은 로컬 빌드 시스템의 위치에서 허용되는 최대 크기 250MB(압축 해제)까지 .zip 패키지를 업로드할 수 있습니다.

AWS SAM에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS SAM 개발자 안내서의 AWS::Serverless::Function을 참조하세요.

예제: AWS SAM을 사용하여 provided.al2023로 Go 함수 구축

1. 다음 속성을 사용하여 AWS SAM 템플릿을 생성합니다.

   - BuildMethod: 애플리케이션의 컴파일러를 지정합니다. `go1.x`를 사용합니다.
   - Runtime: `provided.al2023`를 사용합니다.
   - CodeUri: 코드 경로를 입력합니다.
   - Architectures: arm64 아키텍처의 경우 `[arm64]`를 사용하고, x86_64 명령 세트 아키텍처의 경우 `[amd64]`를 사용하거나 `Architectures` 속성을 제거합니다.

   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 명령을 사용하여 실행 파일을 컴파일합니다.

   ```
   sam build
   ```

3. sam deploy 명령을 사용하여 함수를 Lambda에 배포합니다.

   ```
   sam deploy --guided
   ```

AWS CloudFormation을 사용하여.zip 파일로 함수 생성 및 업데이트

AWS CloudFormation을 사용하여 .zip 파일 아카이브로 Lambda 함수를 생성할 수 있습니다. .zip 파일에서 Lambda 함수를 생성하려면 먼저 Amazon S3 버킷에 파일을 업로드해야 합니다. AWS CLI를 사용하여 Amazon S3 버킷에 파일을 업로드하는 방법에 대한 지침은 AWS CLI 사용 설명서의 객체 이동을 참조하세요.

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스는 Lambda 함수를 지정합니다. 이 리소스에서 다음 속성을 설정하여 .zip 파일 아카이브로 함수를 생성합니다.

- `PackageType` - `Zip`으로 설정됨
- `Code` - `S3Bucket` 및 `S3Key` 필드에 Amazon S3 버킷 이름과 .zip 파일 이름을 입력합니다.
- `Runtime` - 선택한 런타임으로 설정됨

AWS CloudFormation에서 생성하는 .zip 파일은 4MB를 초과할 수 없습니다. AWS CloudFormation에서 .zip 파일을 사용하여 함수를 배포하는 방법에 대해 자세히 알아보려면 AWS CloudFormation 사용 설명서의 AWS::Lambda::Function을 참조하세요.

종속 항목을 위한 Go 계층 생성

> **Note**
>
> 계층을 Go와 같은 컴파일된 언어의 함수와 함께 사용하면 Python과 같은 해석된 언어의 함수와 함께 사용할 때와 같은 이점을 얻지 못할 수 있습니다. Go는 컴파일된 언어이므로 함수가 초기화 단계에서 수동으로 공유 어셈블리를 메모리로 로드해야 하기 때문에 콜드 시간 시간이 늘어날 수 있습니다. 대신 컴파일 시간에 모든 공유 코드를 포함하여 기본 제공 최적화를 활용하는 것이 좋습니다.

이 섹션의 지침은 계층에 종속 항목을 포함하는 방법을 보여줍니다.

또한 Lambda는 `/opt/lib` 디렉터리의 모든 라이브러리와 `/opt/bin` 디렉터리의 모든 바이너리를 자동으로 감지합니다. Lambda가 계층 콘텐츠를 제대로 찾을 수 있도록 다음 구조로 계층을 생성하세요.

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
```

```
    | bin_1
    | bin_2
```

계층을 패키징한 후 the section called “계층 생성 및 삭제” 및 the section called “계층 추가”을 참조하여 계층 설정을 완료합니다.

컨테이너 이미지로 Go Lambda 함수 배포

Go Lambda 함수의 컨테이너 이미지를 빌드하는 두 가지 방법이 있습니다.

- AWS OS 전용 기본 이미지 사용

  Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 기본 이미지를 사용하여 Lambda용 Go 이미지를 빌드합니다. 이미지가 Lambda와 호환되도록 하려면 이미지에 `aws-lambda-go/lambda` 패키지를 포함해야 합니다.
- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 이미지에 `aws-lambda-go/lambda` 패키지를 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

Go 함수 배포를 위한 AWS 기본 이미지

Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 기본 이미지를 사용하여 Lambda에 Go 함수를 배포합니다.

OS 전용

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
OS 전용 런타임	`provided.al2023`	Amazon Linux 2023			

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
OS 전용 런타임	`provided.al2`	Amazon Linux 2			

Amazon Elastic Container Registry 퍼블릭 갤러리: gallery.ecr.aws/lambda/provided

Go 런타임 인터페이스 클라이언트

`aws-lambda-go/lambda` 패키지에는 런타임 인터페이스의 구현이 포함되어 있습니다. 이미지에 `aws-lambda-go/lambda`를 사용하는 방법의 예는 AWS OS 전용 기본 이미지 사용 또는 비AWS 기본 이미지 사용 섹션을 참조하세요.

AWS OS 전용 기본 이미지 사용

Go는 다른 관리형 런타임과 다른 방법으로 구현됩니다. Go는 기본적으로 실행 가능한 바이너리로 컴파일되므로 전용 언어 런타임이 필요하지 않습니다. OS 전용 기본 이미지를 사용하여 Go 함수용 컨테이너 이미지를 빌드합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
al2023	OS 전용 런타임	Amazon Linux 2023	GitHub의 OS 전용 런타임용 Dockerfile	
al2	OS 전용 런타임	Amazon Linux 2	GitHub의 OS 전용 런타임용 Dockerfile	

이러한 기본 이미지에 대한 자세한 내용은 Amazon ECR 퍼블릭 갤러리의 provided를 참조하세요.

Go 핸들러에 aws-lambda-go/lambda 패키지를 포함해야 합니다. 이 패키지는 런타임 인터페이스를 포함하여 Go용 프로그래밍 모델을 구현합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Go

- Docker
- AWS Command Line Interface(AWS CLI) 버전 2

provided.al2023 기본 이미지에서 이미지 생성

`provided.al2023` 기본 이미지로 Go 함수를 빌드하고 배포합니다.

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir hello
cd hello
```

2. 새 Go 모듈을 초기화합니다.

```
go mod init example.com/hello-world
```

3. lambda 라이브러리를 새 모듈의 종속 구성 요소로 추가합니다.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 이름이 `main.go`인 파일을 생성하여 텍스트 편집기에서 엽니다. Lambda 함수에 대한 코드입니다. 다음 샘플 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.

```
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. 텍스트 편집기를 사용하여 프로젝트 디렉터리에 Dockerfile을 생성합니다. 다음 예제 Dockerfile은 다단계 빌드를 사용합니다. 이렇게 하면 각 단계에서 다른 기본 이미지를 사용할 수 있습니다. Go 기본 이미지와 같은 하나의 이미지를 사용하여 코드를 컴파일하고 실행 가능한 바이너리를 빌드할 수 있습니다. 그런 다음 최종 FROM 문에서 provided.al2023 등의 다른 이미지를 사용하여 Lambda에 배포할 이미지를 정의할 수 있습니다. 빌드 프로세스는 최종 배포 이미지와 분리되어 있으므로 최종 이미지에는 애플리케이션을 실행하는 데 필요한 파일만 포함됩니다.

   선택 사항인 lambda.norpc 태그를 사용하여 lambda 라이브러리의 원격 절차 호출(RPC) 구성 요소를 제외할 수 있습니다. RPC 구성 요소는 지원 중단된 Go 1.x 런타임을 사용할 경우에만 필요합니다. RPC를 제외하면 배포 패키지의 크기가 줄어듭니다.

   Example - 다단계 빌드 Dockerfile

   > **Note**
   >
   > Dockerfile에서 지정한 Go 버전(예: golang:1.20)이 애플리케이션을 생성하는 데 사용한 Go 버전과 동일한지 확인합니다.

```
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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 docker-image로 지정하고 test 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

> **Note**
>
> 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. `provided.al2023` 기본 이미지에는 런타임 인터페이스 에뮬레이터가 포함됩니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 실행

1. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `./main`은 Dockerfile의 `ENTRYPOINT`입니다.

   ```
   docker run -d -p 9000:8080 \
   --entrypoint /usr/local/bin/aws-lambda-rie \
   docker-image:test ./main
   ```

   이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.
2. 새 터미널 창에서 curl 명령을 사용하여 다음 엔드포인트에 이벤트를 게시합니다.

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 일부 함수에는 JSON 페이로드가 필요할 수 있습니다. 예제

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```
3. 컨테이너 ID를 가져옵니다.

```
docker ps
```

4. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.
   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.
   - 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:
   - `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
   - `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

비AWS 기본 이미지 사용

비 AWS 기본 이미지에서 Go용 컨테이너 이미지를 빌드할 수 있습니다. 다음 단계의 예제 Dockerfile에서는 Alpine 기본 이미지를 사용합니다.

Go 핸들러에 aws-lambda-go/lambda 패키지를 포함해야 합니다. 이 패키지는 런타임 인터페이스를 포함하여 Go용 프로그래밍 모델을 구현합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- Go
- Docker
- AWS Command Line Interface(AWS CLI) 버전 2

대체 기본 이미지에서 이미지 생성

Alpine 기본 이미지로 Go 함수 빌드 및 배포

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir hello
cd hello
```

2. 새 Go 모듈을 초기화합니다.

```
go mod init example.com/hello-world
```

3. lambda 라이브러리를 새 모듈의 종속 구성 요소로 추가합니다.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 이름이 `main.go`인 파일을 생성하여 텍스트 편집기에서 엽니다. Lambda 함수에 대한 코드입니다. 다음 샘플 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.

```
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. 텍스트 편집기를 사용하여 프로젝트 디렉터리에 Dockerfile을 생성합니다. 다음 예제 Dockerfile에서는 Alpine 기본 이미지를 사용합니다.

   Example Dockerfile

   > **Note**
   >
   > Dockerfile에서 지정한 Go 버전(예: `golang:1.20`)이 애플리케이션을 생성하는 데 사용한 Go 버전과 동일한지 확인합니다.

```
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 build 명령으로 도커 이미지를 빌드합니다. 다음 예제에서는 이미지 이름을 `docker-image`로 지정하고 `test` 태그를 지정합니다.

```
docker build --platform linux/amd64 -t docker-image:test .
```

   > **Note**
   >
   > 이 명령은 빌드 머신의 아키텍처에 관계없이 컨테이너가 Lambda 실행 환경과 호환되는지 확인하기 위해 `--platform linux/amd64` 옵션을 지정합니다. ARM64 명령 세트 아키

텍처를 사용하여 Lambda 함수를 생성하려는 경우 `--platform linux/arm64` 옵션을 대신 사용하도록 명령을 변경해야 합니다.

(선택 사항) 로컬에서 이미지 테스트

런타임 인터페이스 에뮬레이터를 사용하여 이미지를 로컬로 테스트합니다. 에뮬레이터를 이미지에 빌드하거나 다음 절차를 사용하여 로컬 시스템에 설치할 수 있습니다.

로컬 시스템에 런타임 인터페이스 에뮬레이터 설치 및 실행

1. 프로젝트 디렉터리에서 다음 명령을 실행하여 GitHub에서 런타임 인터페이스 에뮬레이터(x86-64 아키텍처)를 다운로드하고 로컬 시스템에 설치합니다.

   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 에뮬레이터를 설치하려면 이전 명령의 GitHub 리포지토리 URL을 다음과 같이 바꿉니다.

   ```
   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 에뮬레이터를 설치하려면 `$downloadLink`을(를) 다음과 같이 바꿉니다.

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. docker run 명령을 사용하여 Docker 이미지를 시작합니다. 유의할 사항:

   - `docker-image`는 이미지 이름이고 `test`는 태그입니다.
   - `/main`은 Dockerfile의 `ENTRYPOINT`입니다.

   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
   ```

   이 명령은 이미지를 컨테이너로 실행하고 `localhost:9000/2015-03-31/functions/function/invocations`에 로컬 엔드포인트를 생성합니다.

   > **Note**
   >
   > ARM64 명령 세트 아키텍처를 위한 도커 이미지를 빌드한 경우 `--platform linux/arm64` 옵션을 `--platform linux/amd64` 대신 사용해야 합니다.

3. 로컬 엔드포인트에 이벤트를 게시합니다.

   Linux/macOS

   Linux 및 macOS에서 다음 `curl` 명령을 실행합니다.

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

PowerShell에서 다음 `Invoke-WebRequest` 명령을 실행합니다.

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

이 명령은 빈 이벤트와 함께 함수를 호출하고 응답을 반환합니다. 샘플 함수 코드가 아닌 자체 함수 코드를 사용하는 경우 JSON 페이로드로 함수를 호출할 수 있습니다. 예제

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 컨테이너 ID를 가져옵니다.

```
docker ps
```

5. docker kill 명령을 사용하여 컨테이너를 중지합니다. 이 명령에서 3766c4ab331c를 이전 단계의 컨테이너 ID로 바꿉니다.

```
docker kill 3766c4ab331c
```

이미지 배포

Amazon ECR에 이미지 배포 및 Lambda 함수 생성

1. get-login-password 명령을 실행하여 Amazon ECR 레지스트리에 대해 Docker CLI를 인증합니다.

   - `--region` 값을 Amazon ECR 리포지토리를 생성하려는 AWS 리전으로 설정합니다.

- 111122223333를 사용자의 AWS 계정 ID로 바꿉니다.

```
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 명령을 사용하여 Amazon ECR에 리포지토리를 생성합니다.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 리포지토리는 Lambda 함수와 동일한 AWS 리전 내에 있어야 합니다.

성공하면 다음과 같은 응답이 표시됩니다.

```
{
    "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. 이전 단계의 출력에서 `repositoryUri`를 복사합니다.
4. docker tag 명령을 실행하여 로컬 이미지를 Amazon ECR 리포지토리에 최신 버전으로 태깅합니다. 이 명령에서:

- `docker-image:test`를 도커 이미지의 이름과 태그로 바꿉니다.
- `<ECRrepositoryUri>`를 복사한 `repositoryUri`로 바꿉니다. URI 끝에 `:latest`를 포함해야 합니다.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

예제

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. docker push 명령을 실행하여 Amazon ECR 리포지토리에 로컬 이미지를 배포합니다. 리포지토리 URI 끝에 `:latest`를 포함해야 합니다.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 함수에 대한 실행 역할이 아직 없는 경우 하나 생성합니다. 다음 단계에서는 역할의 Amazon 리소스 이름(ARN)이 필요합니다.
7. Lambda 함수를 생성합니다. `ImageUri`의 경우 이전의 리포지토리 URI를 지정합니다. URI 끝에 `:latest`를 포함해야 합니다.

```
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**
>
> 이미지가 Lambda 함수와 동일한 리전에 있는 한 다른 AWS 계정의 이미지를 사용하여 함수를 생성할 수 있습니다. 자세한 내용은 Amazon ECR 교차 계정 권한 단원을 참조하십시오.

8. 함수를 호출합니다.

```
aws lambda invoke --function-name hello-world response.json
```

다음과 같은 응답이 표시되어야 합니다.

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

9. 함수의 출력을 보려면 `response.json` 파일을 확인합니다.

함수 코드를 업데이트하려면 이미지를 다시 빌드하고 Amazon ECR 리포지토리에 새 이미지를 업로드한 다음 update-function-code 명령을 사용하여 이미지를 Lambda 함수에 배포해야 합니다.

Lambda는 이미지 태그를 특정 이미지 다이제스트로 확인합니다. 즉, 함수를 배포하는 데 사용된 이미지 태그가 Amazon ECR의 새 이미지로 가리키는 경우 Lambda는 새 이미지를 사용하도록 함수를 자동으로 업데이트하지 않습니다. 새 이미지를 동일한 Lambda 함수에 배포하려면 Amazon ECR의 이미지 태그가 동일하게 유지되더라도 `update-function-code` 명령을 사용해야 합니다.

AWS Lambda 함수 로깅(Go)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그를 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 호출에 관한 세부 정보를 로그 스트림에 전송하며, 함수 코드에서 로그 및 그 외 출력을 중계합니다. 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 단원을 참조하십시오.

이 페이지에서는 AWS Command Line Interface, Lambda 콘솔 또는 CloudWatch 콘솔을 사용하여 Lambda 함수 코드의 로그 출력이나 액세스 로그를 생성하는 방법에 대해 설명합니다.

단원

- 로그를 반환하는 함수 생성
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제

로그를 반환하는 함수 생성

함수 코드의 로그를 출력하려면, fmt 패키지에서 메서드를 사용하거나 `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용합니다. 다음 예제에서는 로그 패키지를 사용합니다.

Example main.go – 로깅

```
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 로그 형식

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 줄을 로깅합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.

- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - `aws configure`을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 LogResult이라는 함수의 my-function 필드에서 로그 ID를 검색하는 방법을 보여 줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 base64 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 my-function에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```


로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

AWS Lambda에서 Go 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 두 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS Distro for OpenTelemetry(ADOT) - 안전하게 프로덕션 준비가 된 AWS에서 지원하는 OpenTelemetry(OTEL) SDK의 배포입니다.
- AWS Go용 X-Ray SDK – 추적 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 어떤 솔루션을 사용하든 X-Ray에서 엔드 투 엔드 추적 기능을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

ADOT를 사용하여 Go 함수 계측

ADOT는 OTel SDK를 사용하여 원격 측정 데이터를 수집하는 데 필요한 모든 것을 패키징할 수 있는 완전 관리형 Lambda 계층을 제공합니다. 이 계층을 사용하면 모든 함수 코드를 수정하지 않고도 Lambda 함수를 계측할 수 있습니다. 계층을 구성하여 OTel의 사용자 지정 초기화를 수행할 수도 있습니다. 자세한 내용은 ADOT 설명서의 Lambda에서 ADOT 컬렉터에 대한 사용자 지정 구성을 참조하세요.

Go 런타임의 경우 ADOT Go용 AWS 관리형 Lambda 계층을 추가하여 함수를 자동으로 계측할 수 있습니다. 이 계층을 추가하는 방법에 대한 자세한 지침은 ADOT 설명서의 AWS Distro for OpenTelemetry Lambda Support for Go를 참조하세요.

X-Ray Go를 사용하여 Java 함수 계측

Lambda 함수가 애플리케이션의 다른 리소스에 대해 수행하는 호출에 대한 세부 정보를 기록하려면 AWS X-Ray Go용 SDK를 사용할 수도 있습니다. SDK를 가져오려면 `go get`을 사용하여 해당 GitHub 리포지토리에서 SDK를 다운로드합니다.

```
go get github.com/aws/aws-xray-sdk-go
```

AWS SDK 클라이언트를 계측하려면 클라이언트를 `xray.AWS()` 메서드에 전달합니다. 그런 다음 메서드의 `WithContext` 버전을 사용하여 호출을 추적할 수 있습니다.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

올바른 종속성을 추가하고 필요한 코드를 변경한 후 Lambda 콘솔 또는 API를 통해 함수의 구성에서 추적을 활성화합니다.

Lambda 콘솔을 사용하여 추적 활성화

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.

3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

Lambda API를 사용하여 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 Lambda 함수에 대한 추적을 구성하고 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation을 사용하여 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray 추적 해석

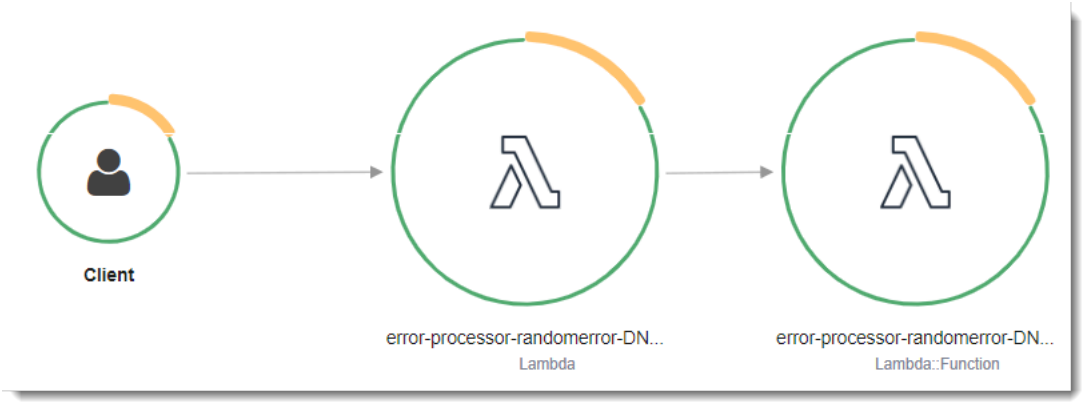
함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

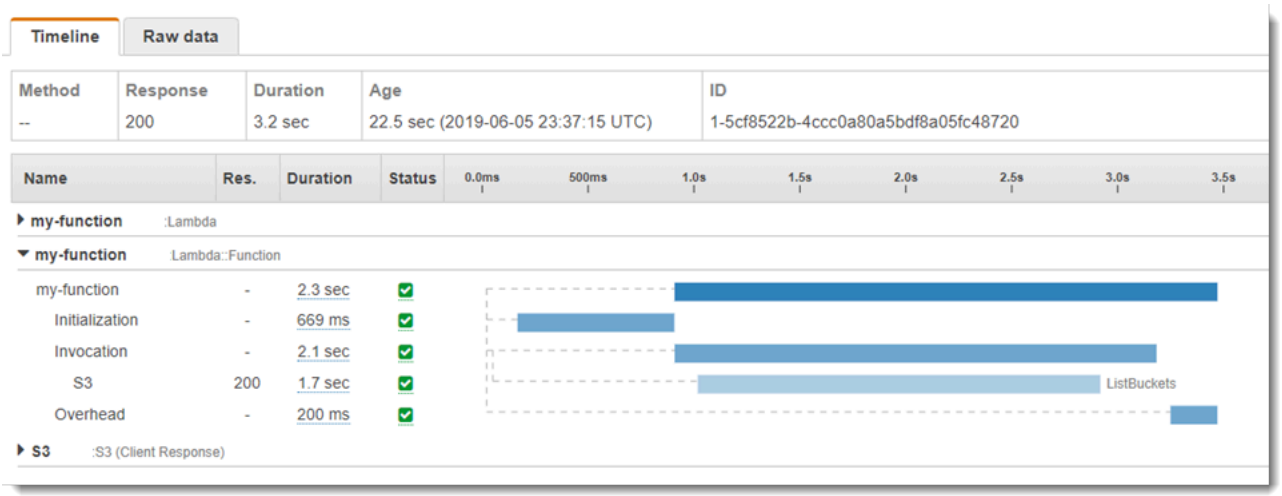
X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. `AWS::Lambda::Function` 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 `AWS::Lambda::Function` 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.
- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 AWS X-Ray SDK for Go를 참조하세요.

> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

환경 변수 사용

Go의 환경 변수에 액세스하려면 Getenv 함수를 사용합니다.

다음 단원에서는 그 방법에 대해 설명합니다. 해당 함수는 출력된 결과들의 형식을 지정하기 위한 fmt 패키지를 가져오며, 환경 변수에 액세스할 수 있도록 플랫폼에 독립적인 시스템 인터페이스인 os 패키지도 가져옵니다.

```
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 런타임에 의해 설정된 환경 변수 목록은 정의된 런타임 환경 변수 단원을 참조하세요.

C#을 사용하여 Lambda 함수 빌드

관리형 .NET 6 또는 .NET 8 런타임, 사용자 지정 런타임 또는 컨테이너 이미지를 사용하여 Lambda에서 .NET 애플리케이션을 실행할 수 있습니다. 애플리케이션 코드가 컴파일된 후에는 .zip 파일 또는 컨테이너 이미지로 Lambda에 배포할 수 있습니다. Lambda는 .NET 언어에 대해 다음과 같은 런타임을 제공합니다.

.NET

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
.NET 8	`dotnet8`	Amazon Linux 2023			
.NET 6	`dotnet6`	Amazon Linux 2	2024년 11월 12일	2025년 2월 28일	2025년 3월 31일

.NET 개발 환경 설정

Lambda 함수를 개발하고 빌드하려면 Microsoft Visual Studio, Visual Studio Code 및 JetBrains Rider를 비롯한 일반적으로 사용 가능한 .NET 통합 개발 환경(IDE)을 아무거나 이용하면 됩니다. 개발 환경을 단순화하기 위해.AWS은(는) NET 프로젝트 템플릿 세트와 `Amazon.Lambda.Tools` 명령줄 인터페이스(CLI)를 제공합니다.

다음.NET CLI 명령을 실행하여 이 프로젝트 템플릿과 명령줄 도구를 설치합니다.

.NET 프로젝트 템플릿 설치

프로젝트 템플릿(.NET 8)을 설치하려면 다음을 수행하세요.

```
dotnet new install Amazon.Lambda.Templates
```

프로젝트 템플릿(.NET 6) 설치하기:

```
dotnet new --install Amazon.Lambda.Templates
```

> **Note**
>
> .NET 6 관리형 Lambda 런타임을 사용하는 경우 .NET 8을 사용하도록 업그레이드하는 것이 좋습니다. 자세히 알아보려면 AWS Compute Blog의 Managing AWS Lambda runtime upgrades 및 Introducing the .NET 8 runtime for AWS Lambda를 참조하세요.

CLI 도구 설치 및 업데이트

다음 명령을 실행하여 `Amazon.Lambda.Tools` CLI를 설치, 업데이트 및 제거합니다.

명령줄 도구 설치하기:

```
dotnet tool install -g Amazon.Lambda.Tools
```

명령줄 도구 업데이트:

```
dotnet tool update -g Amazon.Lambda.Tools
```

명령줄 도구 제거:

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

C#에서 Lambda 함수 핸들러 정의

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

함수가 간접적으로 호출되고 Lambda가 함수의 핸들러 메서드를 실행하면 함수에 두 인수를 전달합니다. 첫 번째 인수는 `event` 객체입니다. 다른 AWS 서비스이(가) 함수를 간접적으로 호출하면 `event` 객체에는 함수 간접 호출을 유발한 이벤트에 대한 데이터가 포함됩니다. 예를 들어 API Gateway의 `event` 객체에는 경로, HTTP 메서드, HTTP 헤더에 대한 정보가 포함되어 있습니다. 정확한 이벤트 구조는 AWS 서비스 함수 간접 호출에 따라 달라집니다. 개별 서비스의 이벤트 형식에 대한 자세한 내용은 다른 서비스 통합을(를) 참조하십시오.

또한 Lambda는 `context` 객체를 함수에 전달합니다. 이 객체에는 호출, 함수 및 실행 환경에 대한 정보가 포함되어 있습니다. 자세한 내용은 the section called “컨텍스트” 단원을 참조하십시오.

모든 Lambda 이벤트의 기본 형식은 JSON 형식의 이벤트를 나타내는 바이트 스트림입니다. 함수 입력 및 출력 파라미터는 `System.IO.Stream` 유형인 경우를 제외하고 직렬화해야 합니다. `LambdaSerializer` 어셈블리 속성을 설정하여 사용하려는 시리얼라이저를 지정하십시오. 자세한 내용은 the section called “Lambda 함수의 직렬화” 단원을 참조하십시오.

주제

- 람다용 .NET 실행 모델
- 클래스 라이브러리 핸들러
- 실행 가능한 어셈블리 핸들러
- Lambda 함수의 직렬화
- Lambda 주석 프레임워크를 사용하여 함수 코드를 간소화합니다
- Lambda 함수 핸들러 제한 사항

람다용 .NET 실행 모델

.NET에서 Lambda 함수를 실행하기 위한 두 가지 실행 모델, 즉 클래스 라이브러리 접근 방식과 실행 가능한 어셈블리 접근 방식이 있습니다.

클래스 라이브러리 접근 방식에서는 간접적으로 호출할 함수의 `AssemblyName`, `ClassName` 및 `Method`을(를) 나타내는 문자열이 있는 Lambda를 제공합니다. 이 문자열의 형식에 대한 자세한 내용

은 the section called “클래스 라이브러리 핸들러”을(를) 참조하십시오. 함수 초기화 단계에서 함수 클래스가 초기화되고 생성자의 코드가 실행됩니다.

실행 가능한 어셈블리 접근 방식에서는 C# 9의 최상위 명령문 기능을 사용합니다. 이 접근 방식은 함수에 대한 간접 호출 명령을 수신할 때마다 Lambda가 실행하는 실행 가능한 어셈블리를 생성합니다. 실행할 실행 가능한 어셈블리의 이름만 Lambda에 제공합니다.

다음 섹션에서는 이 두 가지 접근 방식에 대한 예제 함수 코드를 제공합니다.

클래스 라이브러리 핸들러

다음 Lambda 함수 코드는 클래스 라이브러리 접근 방식을 사용하는 Lambda 함수에 대한 핸들러 메서드(`FunctionHandler`)의 예를 보여줍니다. 이 예제 함수에서 Lambda는 함수를 간접적으로 호출하는 API Gateway로부터 이벤트를 수신합니다. 이 함수는 데이터베이스에서 레코드를 읽고 API Gateway 응답의 일부로 레코드를 반환합니다.

```
[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 함수를 생성할 때는 함수 핸들러에 대한 정보를 핸들러 문자열 형태로 Lambda에 제공해야 합니다. 이를 통해 Lambda는 함수가 간접적으로 호출될 때 코드의 어떤 메서드를 실행할지 알 수 있습니다. C#에서 클래스 라이브러리 접근 방식을 사용할 때의 핸들러 문자열 형식은 다음과 같습니다.

`ASSEMBLY::TYPE::METHOD`, 여기서 각 항목은 다음과 같습니다.

- `ASSEMBLY`은(는) 애플리케이션에 대한 .NET 어셈블리 파일의 이름입니다. `Amazon.Lambda.Tools` CLI를 사용하여 애플리케이션을 빌드하고 .csproj 파일의 `AssemblyName` 속성을 사용하여 어셈블리 이름을 설정하지 않은 경우, `ASSEMBLY`은(는) 단지.csproj 파일의 이름이 됩니다.
- `TYPE`은(는) 핸들러 유형의 전체 이름으로,`Namespace` 및 `ClassName`(으)로 돼 있습니다.
- `METHOD`은(는) 코드에 있는 함수 핸들러 메서드의 이름입니다.

표시된 예제 코드에서 어셈블리 이름이 `GetProductHandler`(으)로 지정되면 핸들러 문자열은 `GetProductHandler::GetProductHandler.Function::FunctionHandler`이(가) 됩니다.

실행 가능한 어셈블리 핸들러

다음 예제에서 Lambda 함수는 실행 가능한 어셈블리로 정의됩니다. 이 코드의 핸들러 메서드는 `Handler`(이)라는 이름으로 지정됩니다. 실행 가능한 어셈블리를 사용하는 경우 Lambda 런타임을 부트스트랩해야 합니다. 그렇게 하려면 `LambdaBootstrapBuilder.Create` 메서드를 사용합니다. 이 메서드는 함수가 핸들러로 사용하는 메서드와 사용할 Lambda 시리얼라이저를 입력으로 사용합니다.

최상위 명령문 사용에 대한 자세한 내용은 AWS compute 블로그에서.AWS Lambda의 NET 6 런타임 소개를 참조하십시오.

```
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)
    };
};
```

실행 가능한 어셈블리를 사용할 때 Lambda에 코드 실행 방법을 알려주는 핸들러 문자열이 어셈블리의 이름입니다. 이 예시에서는 `GetProductHandler`이(가) 될 것입니다.

Lambda 함수의 직렬화

Lambda 함수가 `Stream` 객체 이외의 입력 또는 출력 유형을 사용하는 경우, 애플리케이션에 직렬화 라이브러리를 추가해야 합니다. `System.Text.Json` 및 `Newtonsoft.Json`에서 제공하는 표준 리플렉션 기반 직렬화를 사용하거나 소스 생성 직렬화를 사용하여 직렬화를 구현할 수 있습니다.

소스 생성 직렬화 사용

소스 생성 직렬화는 컴파일 시간에서 직렬화 코드를 생성할 수 있는 .NET 버전 6 이상의 기능입니다. 이를 통해 리플렉션이 필요하지 않으며 함수의 성능을 향상시킬 수 있습니다. 함수에서 소스 생성 직렬화를 사용하려면 다음과 같이 하십시오.

- `JsonSerializerContext`에서 상속되는 새 부분 클래스를 생성하면서 직렬화 또는 역직렬화가 필요한 모든 유형에 대한 `JsonSerializable` 속성을 추가합니다.
- `SourceGeneratorLambdaJsonSerializer<T>`을(를) 사용하여 `LambdaSerializer`을(를) 구성합니다.
- 새로 생성한 클래스를 사용하도록 애플리케이션 코드의 수동 직렬화 또는 역직렬화를 업데이트하십시오.

소스 생성 직렬화를 사용하는 예제 함수는 다음 코드에 나와 있습니다.

```
[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**
>
> Lambda를 통해 네이티브 사전 컴파일(AOT)을 사용하려면 소스 생성 직렬화를 사용해야 합니다.

리플렉션 기반 직렬화 사용

AWS은(는) 애플리케이션에 직렬화를 빠르게 추가할 수 있도록 사전에 빌드된 라이브러리를 제공합니다. `Amazon.Lambda.Serialization.SystemTextJson` 또는 `Amazon.Lambda.Serialization.Json` NuGet 패키지를 사용하여 이

를 구성합니다. Amazon.Lambda.Serialization.SystemTextJson은(는) 백그라운드에서 System.Text.Json을(를) 사용하여 직렬화 작업을 수행하고, Amazon.Lambda.Serialization.Json은(는) Newtonsoft.Json 패키지를 사용합니다.

ILambdaSerializer 라이브러리의 일부로 사용할 수 있는 Amazon.Lambda.Core 인터페이스를 구현하여 자체 직렬화 라이브러리를 생성할 수도 있습니다. 이 인터페이스는 다음과 같은 두 가지 메서드를 정의합니다.

- T Deserialize<T>(Stream requestStream);

  이 메서드를 구현하여 Invoke API에서 Lambda 함수 핸들러로 전달되는 객체로 요청 페이로드를 역직렬화합니다.
- T Serialize<T>(T response, Stream responseStream);

  이 메서드를 구현하여 Lambda 함수 핸들러에서 반환된 결과를 Invoke API 작업에서 반환된 응답 페이로드로 직렬화합니다.

Lambda 주석 프레임워크를 사용하여 함수 코드를 간소화합니다

Lambda 주석은 C#을 사용하여 Lambda 함수 작성을 간소화하는 .NET 6 및 .NET 8용 프레임워크입니다. 주석 프레임워크를 사용하면 일반 프로그래밍 모델을 사용하여 작성된 Lambda 함수의 코드 대부분을 대체할 수 있습니다. 프레임워크를 사용하여 작성된 코드는 더 간단한 식을 사용하므로 비즈니스 로직에 집중할 수 있습니다.

다음 예제 코드는 주석 프레임워크를 사용하여 Lambda 함수 작성을 간소화하는 방법을 보여줍니다. 첫 번째 예제는 일반 Lambda 프로그램 모델을 사용하여 작성된 코드를 보여주고, 두 번째 예제는 주석 프레임워크를 사용하여 동일한 코드를 보여줍니다.

```
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 주석을 사용하여 코드를 간소화하는 방법에 대한 또 다른 예는 awsdocs/aws-doc-sdk-examples GitHub 리포지토리의 이 크로스 서비스 예제 애플리케이션을 참조하십시오. 이 폴더 PamApiAnnotations은(는) 주요 function.cs 파일의 Lambda 주석을 사용합니다. PamApi 폴더에는 비교를 위해 일반 Lambda 프로그래밍 모델을 사용하여 작성된 동일한 파일이 있습니다.

주석 프레임워크는 소스 생성기를 사용하여 Lambda 프로그래밍 모델을 두 번째 예제에 표시된 코드로 변환하는 코드를 생성합니다.

NET용으로 Lambda 주석을 사용하는 방법에 대한 자세한 내용은 다음 리소스를 참조하십시오.

- aws/aws-lambda-dotnet GitHub 리포지토리.
- AWS개발자 도구 블로그에서 .NET 주석 Lambda 프레임워크(미리 보기)를 소개합니다.
- Amazon.Lambda.Annotations NuGet 패키지.

Lambda 주석 프레임워크를 사용한 종속성 주입

또한 Lambda 주석 프레임워크를 사용하면 익숙한 구문을 활용하여 Lambda 함수에 종속성 주입을 추가할 수 있습니다. `Startup.cs` 파일에 `[LambdaStartup]` 속성을 추가하면 Lambda 주석 프레임워크가 컴파일 타임에 필요한 코드를 생성합니다.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Lambda 함수는 생성자 주입을 사용하여 서비스를 주입하거나 `[FromServices]` 속성을 사용하여 개별 메서드에 주입할 수 있습니다.

```
[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 함수 핸들러 제한 사항

핸들러 서명에는 몇 가지 제한이 있습니다.

- `unsafe`이어서는 안 되고, 핸들러 메서드 및 종속 프로그램 내에서 `unsafe` 콘텍스트가 사용될 수는 있지만 핸들러 서명에서 포인터 유형을 사용해서는 안 됩니다. 자세한 내용은 Microsoft Docs 웹 사이트에서 unsafe(C# Reference)를 참조하세요.
- `params` 키워드를 사용하여 다수의 파라미터를 전달하거나 `ArgIterator`를 입력으로 사용하거나 다수의 파라미터를 지원하는 데 사용되는 파라미터를 반환하지 않을 수 있습니다.
- 핸들러는 일반 메서드(예: IList<T> Sort<T>(IList<T> input))가 아닙니다.
- 서명 `async void`이 있는 비동기식 핸들러는 지원되지 않습니다.

.zip 파일 아카이브를 사용하여 C# Lambda 함수를 빌드 및 배포

.NET 배포 패키지(.zip 파일 아카이브)에는 함수의 컴파일된 어셈블리와 해당 어셈블리의 모든 종속 항목이 함께 포함되어 있습니다. 패키지는 *proj*`.deps.json` 파일도 포함합니다. 이 패키지는 함수의 모든 종속성과 *proj*`.runtimeconfig.json` 파일을 .NET 런타임으로 전송하며, 해당 파일은 런타임을 구성하는 데 사용됩니다.

개별 Lambda 함수를 배포하려면`Amazon.Lambda.Tools` NET Lambda 글로벌 CLI를 사용하면 됩니다. `dotnet lambda deploy-function` 명령을 사용하면.zip 배포 패키지가 자동으로 생성되어 Lambda에 배포됩니다. 하지만 AWS Serverless Application Model(AWS SAM) 또는 AWS Cloud Development Kit (AWS CDK)와(과) 같은 프레임워크를 사용하여 .NET 애플리케이션을 AWS에 배포하는 것이 좋습니다.

서버리스 애플리케이션은 일반적으로 특정 비즈니스 작업을 수행하기 위해 함께 작동하는 Lambda 함수와 기타 관리형 AWS 서비스의 조합으로 구성됩니다. AWS SAM과 AWS CDK는 규모에 따라 다른 AWS 서비스와 함께 Lambda 함수를 빌드하고 배포하는 것을 단순화합니다. AWS SAM 템플릿 사양에서는 서버리스 애플리케이션을 구성하는 Lambda 함수, API, 권한, 구성 및 기타 AWS 리소스를 설명하는 간단하고 깔끔한 구문을 제공합니다. AWS CDK을(를) 사용하여 클라우드 인프라를 코드로 정의하면 .NET과 같은 최신 프로그래밍 언어와 프레임워크를 활용하여 클라우드에서 신뢰할 수 있고, 확장 가능하며, 비용 대비 효율적인 애플리케이션을 빌드하는 데 도움이 됩니다. AWS CDK 및 AWS SAM은(는) 모두 .NET Lambda 글로벌 CLI를 사용하여 함수를 패키징합니다.

.NET Core CLI를 사용하면 C#의 함수와 함께 Lambda 계층을 사용할 수 있지만 이는 권장되지 않습니다. 계층을 사용하는 C#의 함수는 초기화 단계 중 공유 어셈블리를 메모리에 수동으로 로드하므로 콜드 시작 시간을 늘릴 수 있습니다. 대신 컴파일 시간에 모든 공유 코드를 포함하여 .NET 컴파일러의 기본 제공 최적화를 활용하세요.

AWS SAM, AWS CDK 및 .NET Lambda 글로벌 CLI를 사용하여.NET Lambda 함수를 빌드하고 배포하는 지침은 다음 섹션에서 확인할 수 있습니다.

주제

- .NET Lambda 글로벌 CLI 사용
- AWS SAM을 사용하여 C# Lambda 함수 배포
- AWS CDK을 사용하여 C# Lambda 함수 배포
- ASP.NET 애플리케이션 배포

.NET Lambda 글로벌 CLI 사용

.NET CLI와 .NET Lambda 글로벌 도구 확장(`Amazon.Lambda.Tools`)은.NET 기반 Lambda 애플리케이션을 생성하고, 패키징하고, Lambda에 배포하는 크로스 플랫폼 방법을 제공합니다. 이 섹션에서는.NET CLI 및 Amazon Lambda 템플릿을 사용하여 새 Lambda .NET 프로젝트를 생성하고 `Amazon.Lambda.Tools`을(를) 사용하여 패키징하고 배포하는 방법을 알아봅니다

주제

필수 조건

.NET 8 SDK

아직 하지 않았다면 .NET 8 SDK 및 런타임을 설치하세요.

AWS Amazon.Lambda.Templates .NET프로젝트 템플릿

Lambda 함수 코드를 생성하려면 Amazon.Lambda.Templates NuGet 패키지를 사용합니다. 이 템플릿 패키지를 설치하려면 다음 명령을 실행합니다.

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET 글로벌 CLI 도구

Lambda 함수를 생성하려면 Amazon.Lambda.Tools .NET 글로벌 도구 확장을 사용합니다. Amazon.Lambda.Tools를 설치하려면 다음 명령을 실행합니다.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 확장에 대한 자세한 내용은 GitHub에서 AWS Extensions for .NET CLI 리포지토리를 참조하세요.

.NET CLI를 사용하여 .NET 프로젝트 생성

.NET CLI에서는 `dotnet new` 명령을 사용해 한 명령줄에서 .NET 프로젝트를 생성합니다. Lambda는 `Amazon.Lambda.Templates` NuGet 패키지를 사용하여 추가 템플릿을 제공합니다.

이 패키지를 설치한 후 다음 명령을 실행하여 사용 가능한 템플릿 목록을 확인합니다.

```
dotnet new list
```

템플릿에 대한 세부 정보를 검토하려면 `help` 옵션을 사용하세요. 예를 들어 `lambda.EmptyFunction` 템플릿에 대한 세부 정보를 확인하려면 다음 명령을 실행합니다.

```
dotnet new lambda.EmptyFunction --help
```

.NET Lambda 함수의 기본 템플릿을 생성하려면 `lambda.EmptyFunction` 템플릿을 사용하십시오. 그러면 문자열을 입력으로 받아 `ToUpper` 메서드를 사용하여 대문자로 변환하는 간단한 함수가 생성됩니다. 이 템플릿은 다음 옵션을 지원합니다.

- `--name` – 함수의 이름입니다.
- `--region` - 함수를 생성할 AWS 리전입니다.
- `--profile` – 사용자의 AWS SDK for .NET 자격 증명에 있는 프로필의 이름입니다. .NET의 보안 인증 프로필에 대한 자세한 내용은 .NET 개발자 안내서용 AWS SDK에서 AWS 보안 인증 구성을 참조하십시오.

이 예제에서는 기본 프로필 및 AWS 리전 설정을 사용하여 `myDotnetFunction`(이)라는 빈 함수를 새로 만듭니다.

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

이 명령은 프로젝트 디렉터리에 다음 파일 및 디렉터리를 생성합니다.

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
```

```
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

src/myDotnetFunction 디렉터리 아래에서 다음과 같은 파일들을 검사합니다.

- aws-lambda-tools-defaults.json: Lambda 함수를 배포할 때 이 파일에 명령줄 옵션을 지정합니다. 예:

```
  "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 핸들러 함수 코드입니다. 이 코드는 기본 Amazon.Lambda.Core 라이브러리와 기본 LambdaSerializer 속성을 포함하는 C# 템플릿에 속합니다. 직렬화 요구 사항들과 옵션들에 관한 자세한 내용은 Lambda 함수의 직렬화 단원을 참조하세요. 이 단원은 Lambda 함수 코드를 적용하기 위해 편집할 수 있는 하나의 샘플 함수도 포함하고 있습니다.

```
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: 애플리케이션을 구성하는 파일 및 어셈블리들의 목록을 열거한 MSBuild 파일입니다.

```
<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: 이 파일을 사용하면 Lambda 함수를 문서화할 수 있습니다.

`myfunction/test` 디렉터리 아래에서 다음과 같은 파일들을 검사합니다.

- myDotnetFunction.Tests.csproj: 앞서 언급한 것처럼 이것은 테스트 프로젝트를 구성하는 여러 파일 및 어셈블리를 열거한 MSBuild 파일입니다. 이 파일은 `Amazon.Lambda.Core` 라이브러리도 포함하고 있기 때문에 함수를 테스트하는 데 필요한 Lambda 템플릿을 원활하게 통합할 수 있습니다.

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
```

```
    ...
```

- FunctionTest.cs: src 디렉터리에 포함된 것과 똑같은 C# 코드 템플릿 파일입니다. Lambda 함수를 프로덕션 환경에 업로드하기 전에 함수의 프로덕션 코드를 미러링하고 테스트할 수 있도록 이 파일을 편집합니다.

```
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 CLI를 사용하여 .NET 프로젝트 배포하기

배포 패키지를 빌드하고 Lambda에 배포하려면 Amazon.Lambda.Tools CLI 도구를 사용합니다. 이전 단계에서 생성한 파일로 함수를 배포하려면 먼저 함수의 .csproj 파일이 포함된 폴더로 이동하십시오.

```
cd myDotnetFunction/src/myDotnetFunction
```

코드를 Lambda에.zip 배포 패키지로 배포하려면 다음 명령을 실행합니다. 자체 함수 이름을 선택하십시오.

```
dotnet lambda deploy-function myDotnetFunction
```

배포 중에 마법사는 the section called “실행 역할(함수가 다른 리소스에 액세스할 수 있는 권한)”을 선택하라고 요청합니다. 이 예제에서는 lambda_basic_role을(를) 선택합니다.

함수를 배포한 후 dotnet lambda invoke-function 명령을 사용하여 클라우드에서 테스트할 수 있습니다. lambda.EmptyFunction 템플릿에 있는 예제 코드의 경우 --payload 옵션을 사용하여 문자열을 전달하여 함수를 테스트할 수 있습니다.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

함수가 성공적으로 배포된 경우 다음과 유사한 출력이 표시될 것입니다.

```
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 CLI와 함께 Lambda 계층 사용

> **Note**
>
> 계층을 C#과 같은 컴파일된 언어의 함수와 함께 사용하면 Python과 같은 해석된 언어의 함수와 함께 사용할 때와 같은 이점을 얻지 못할 수 있습니다. C#은 컴파일된 언어이므로 함수가 초기화 단계에서 수동으로 공유 어셈블리를 메모리로 로드해야 하기 때문에 콜드 시간 시간이 늘어날 수 있습니다. 대신 컴파일 시간에 모든 공유 코드를 포함하여 기본 제공 최적화를 활용하는 것이 좋습니다.

.NET CLI는 계층 게시 및 계층을 사용하는 C# 함수 배포에 도움이 되는 명령을 지원합니다. 지정된 Amazon S3 버킷에 계층을 게시하려면 `..csproj` 파일과 동일한 디렉터리에서 다음 명령을 실행합니다.

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

그런 다음 .NET CLI를 사용하여 함수를 배포할 때 다음 명령에서 사용할 계층 ARN을 지정합니다.

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

Hello World 함수의 전체 예제는 blank-csharp-with-layer 샘플을 참조하세요.

AWS SAM을 사용하여 C# Lambda 함수 배포

AWS Serverless Application Model(AWS SAM)은 AWS에서 서버리스 애플리케이션을 빌드하고 실행하는 프로세스를 간소화하는 데 도움이 되는 도구 키트입니다. YAML 또는 JSON 템플릿에서 애플리케이션의 리소스를 정의하고 AWS SAM Command Line Interface(AWS SAM CLI)를 사용하여 애플리케이션을 빌드, 패키징 및 배포합니다. AWS SAM 템플릿에서 Lambda 함수를 빌드하면 AWS SAM은(는) 함수 코드와 사용자가 지정하는 종속 항목을 사용하여 .zip 배포 패키지 또는 컨테이너 이미지를 자동으로 생성합니다. AWS SAM은(는) AWS CloudFormation 스택을 사용하여 함수를 배포합니다. AWS SAM을 사용하여 Lambda 함수를 빌드하고 배포하는 방법에 대해 자세히 알아보려면 AWS Serverless Application Model 개발자 안내서의 Getting started with AWS SAM을 참조하세요.

다음 단계에서 AWS SAM을(를) 사용하여 샘플 .NET Hello World 애플리케이션을 다운로드, 빌드 및 배포하는 방법을 보여줍니다. 이 샘플 애플리케이션은 Lambda 함수와 Amazon API Gateway 엔드포인트를 사용하여 기본 API 백엔드를 구현합니다. API Gateway 엔드포인트에 HTTP GET 요청을 보내면 API Gateway가 Lambda 함수를 호출합니다. 함수는 요청을 처리하는 Lambda 함수 인스턴스의 IP 주소와 함께 “hello world” 메시지를 반환합니다.

AWS SAM을(를) 사용하여 애플리케이션을 빌드하고 배포하면 AWS SAM CLI는 백그라운드에서 `dotnet lambda package` 명령을 사용하여 개별 Lambda 함수 코드 번들을 패키징합니다.

필수 조건

.NET 8 SDK

.NET 8 SDK 및 런타임을 설치합니다.

AWS SAM CLI 버전 1.39 이상

AWS SAM CLI의 최신 버전을 설치하는 방법에 대해 알아보려면 AWS SAM CLI 설치를 확인하십시오.

샘플 AWS SAM 애플리케이션 배포

1. 다음 명령을 활용하는 Hello world .NET 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

이 명령은 프로젝트 디렉터리에 다음 파일 및 디렉터리를 생성합니다.

```
### 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`이(가) 포함된 디렉토리로 이동합니다. 이 파일은 Lambda 함수 및 API Gateway API를 포함하여 애플리케이션의 AWS 리소스를 정의하는 템플릿입니다.

```
cd sam-app
```

3. 애플리케이션의 소스를 빌드하려면 다음 명령을 실행합니다.

```
sam build
```

4. AWS에 애플리케이션을 배포하려면 다음 명령을 실행합니다.

```
sam deploy --guided
```

이 명령은 다음과 같은 일련의 프롬프트와 함께 애플리케이션을 패키징하고 배포합니다. 기본 옵션을 그대로 사용하려면 입력 키를 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수도 있습니다. 괜찮습니까?에 대해 y을(를) 입력합니다.

- 스택 이름: AWS CloudFormation에 배포할 스택의 이름입니다. 이 이름은 AWS 계정 및 AWS 리전에 고유한 것이어야 합니다.
- AWS 리전: 앱을 배포하려는 AWS 리전입니다.
- 배포 전 변경 사항 확인: AWS SAM이(가) 애플리케이션 변경 사항을 배포하기 전에 변경 세트를 수동으로 검토하려면 예를 선택합니다. 아니요를 선택하면 AWS SAM CLI가 애플리케이션 변경 사항을 자동으로 배포합니다.
- SAM CLI IAM 역할 생성 허용: 이 예제의 Hello world 템플릿을 비롯한 많은 AWS SAM 템플릿은 AWS Identity and Access Management (IAM) 역할을 생성하여 Lambda 함수를 다른 AWS 서비스에 액세스할 수 있는 권한을 부여합니다. IAM 역할을 생성하거나 수정하는 AWS CloudFormation 스택을 배포할 권한을 제공하려면 예를 선택합니다.
- 롤백 비활성화: 기본적으로 AWS SAM이(가) 스택을 생성하거나 배포하는 동안 오류가 발생하면 스택을 이전 버전으로 롤백합니다. 이 기본값을 허용하려면 아니요를 선택합니다.
- HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니까: y을(를) 입력합니다.
- samconfig.toml에 인수 저장: 구성 선택 사항을 저장하려면 예를 선택합니다. 앞으로는 파라미터 `sam deploy` 없이 다시 실행하여 변경 내용을 애플리케이션에 배포할 수 있습니다.

5. 애플리케이션 배포가 완료되면 CLI는 Hello World Lambda 함수의 Amazon 리소스 이름(ARN)과 이 함수에 대해 생성된 IAM 역할을 반환합니다. 또한 API Gateway API의 엔드포인트가 표시됩니다. 애플리케이션을 테스트하려면 브라우저에서 엔드포인트를 엽니다. 다음과 유사한 응답이 나타납니다.

```
{"message":"hello world","location":"34.244.135.203"}
```

6. 다음 명령을 실행하여 리소스를 삭제합니다. 참고로 생성한 API 엔드포인트는 인터넷을 통해 액세스할 수 있는 퍼블릭 엔드포인트입니다. 테스트 후에는 이 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

다음 단계

.NET을 사용하여 Lambda 함수를 빌드하고 배포하기 위해 AWS SAM을(를) 사용하는 방법에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

- AWS Serverless Application Model (AWS SAM) 개발자 안내서가이드
- AWS Lambda 및 SAM CLI를 사용하여 서버리스 .NET 애플리케이션 빌드하기

AWS CDK을 사용하여 C# Lambda 함수 배포

AWS Cloud Development Kit (AWS CDK)은(는) .NET과 같은 최신 프로그래밍 언어 및 프레임워크를 사용하여 클라우드 인프라를 코드로 정의하는 오픈 소스 소프트웨어 개발 프레임워크입니다. AWS CDK프로젝트를 실행하여 AWS CloudFormation 템플릿을 생성한 다음 이를 사용하여 코드를 배포합니다.

AWS CDK을(를) 사용하여 예제 Hello world .NET 애플리케이션을 빌드하고 배포하려면 다음 섹션의 지침을 따르십시오. 샘플 애플리케이션은 API Gateway 엔드포인트와 Lambda 함수로 구성된 기본 API 백엔드를 구현합니다. 엔드포인트에 HTTP GET 요청을 보내면 API Gateway는 Lambda 함수를 호출합니다. 함수는 요청을 처리하는 Lambda 인스턴스의 IP 주소와 함께 Hello world 메시지를 반환합니다.

필수 조건

.NET 8 SDK

.NET 8 SDK 및 런타임을 설치합니다.

AWS CDK 버전 2

AWS CDK의 최신 버전을 설치하는 방법을 알아보려면 AWS Cloud Development Kit (AWS CDK) v2 개발자 안내서의 AWS CDK(으)로 시작하기를 확인하십시오.

샘플 AWS CDK 애플리케이션 배포

1. 샘플 애플리케이션을 위한 프로젝트 디렉터리를 생성하고 탐색합니다.

```
mkdir hello-world
cd hello-world
```

2. 다음 명령을 실행하여 새 AWS CDK 애플리케이션을 초기화합니다.

```
cdk init app --language csharp
```

이 명령은 프로젝트 디렉터리에 다음과 같은 파일 및 디렉터리를 생성합니다

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. `src` 디렉터리를 열고 .NET CLI를 사용하여 새 Lambda 함수를 생성합니다. AWS CDK을(를) 사용하여 배포할 함수입니다. 이 예제에서는 `lambda.EmptyFunction` 템플릿을 사용하여 `HelloWorldLambda`(이)라는 Hello world 함수를 생성합니다.

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

이 단계 후에 프로젝트 디렉터리 내부의 디렉터리 구조는 다음과 같아야 합니다.

```
### 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 디렉터리에서 HelloWorldStack.cs 파일을 엽니다. 파일의 내용을 다음 코드로 바꿉니다.

```
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
                }),
            });
        }
    }
}
```

Lambda 함수 자체의 정의뿐만 아니라 애플리케이션 코드를 컴파일하고 번들링하는 코드입니다. BundlingOptions 객체를 사용하면 zip 파일의 콘텐츠를 생성하는 데 사용되는 명령 집합과 함께 zip 파일을 생성할 수 있습니다. 이 경우 `dotnet lambda package` 명령은 zip 파일을 컴파일하고 생성하는 데 사용됩니다.

5. 애플리케이션을 배포하려면 다음 명령을 입력합니다.

```
cdk deploy
```

6. .NET Lambda CLI를 사용하여 배포된 Lambda 함수를 호출하십시오.

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. 테스트를 마친 후 생성한 리소스를 보관하고 싶지 않다면 삭제해도 됩니다. 다음 명령을 실행하여 리소스를 삭제합니다.

```
cdk destroy
```

다음 단계

.NET을 사용하여 Lambda 함수를 빌드하고 배포하기 위해 AWS CDK을(를) 사용하는 방법에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

- C#에서 AWS CDK 작업
- AWS CDK를 사용하여.NET C# Lambda 함수를 빌드, 패키징 및 게시합니다

ASP.NET 애플리케이션 배포

이벤트 기반 함수를 호스팅할 뿐만 아니라 Lambda와 함께.NET을 사용하여 가벼운 ASP.NET 애플리케이션을 호스팅할 수도 있습니다. `Amazon.Lambda.AspNetCoreServer` NuGet 패키지를 사용하여 ASP.NET 애플리케이션을 빌드하고 배포할 수 있습니다. 이 섹션에서는.NET Lambda CLI 도구를 사용하여 Lambda에 ASP.NET 웹 API를 배포하는 방법을 알아봅니다.

주제

- 필수 조건
- Lambda에 ASP.NET 웹 API 배포하기
- Lambda에 ASP.NET 최소 API 배포하기

필수 조건

.NET 8 SDK

.NET 8 SDK 및 ASP.NET Core 런타임을 설치합니다.

Amazon.Lambda.Tools

Lambda 함수를 생성하려면 Amazon.Lambda.Tools .NET 글로벌 도구 확장을 사용합니다. Amazon.Lambda.Tools를 설치하려면 다음 명령을 실행합니다.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 확장에 대한 자세한 내용은 GitHub에서 AWS Extensions for .NET CLI 리포지토리를 참조하세요.

Amazon.Lambda.Templates

Lambda 함수 코드를 생성하려면 Amazon.Lambda.Templates NuGet 패키지를 사용합니다. 이 템플릿 패키지를 설치하려면 다음 명령을 실행합니다.

```
dotnet new --install Amazon.Lambda.Templates
```

Lambda에 ASP.NET 웹 API 배포하기

ASP.NET을 사용하여 웹 API를 배포하기 위해.NET Lambda 템플릿을 사용하여 새 웹 API 프로젝트를 생성할 수 있습니다. 다음 명령을 사용하여 새 ASP.NET 웹 API 프로젝트를 초기화합니다. 예제 명령에서는 프로젝트 AspNetOnLambda의 이름을 지정합니다.

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

이 명령은 프로젝트 디렉터리에 다음 파일 및 디렉터리를 생성합니다.

```
.
### 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
```

Lambda가 함수를 호출할 때 사용하는 진입점 LambdaEntryPoint.cs 파일입니다. .NET Lambda 템플릿으로 생성된 파일에는 다음 코드가 포함되어 있습니다.

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
```

```
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

Lambda에서 사용하는 진입점은 `Amazon.Lambda.AspNetCoreServer` 패키지의 세 가지 기본 클래스 중 하나를 상속해야 합니다. 이 세 가지 기본 클래스는 다음과 같습니다.

- `APIGatewayProxyFunction`
- `APIGatewayHttpApiV2ProxyFunction`
- `ApplicationLoadBalancerFunction`

제공된.NET Lambda 템플릿을 사용하여 `LambdaEntryPoint.cs` 파일을 생성할 때 사용되는 기본 클래스는 `APIGatewayProxyFunction`입니다. 함수에서 사용하는 기본 클래스는 Lambda 함수 앞에 있는 API 계층에 따라 달라집니다.

세 기본 클래스 각각에는 `FunctionHandlerAsync`(이)라는 퍼블릭 메서드가 포함되어 있습니다. 이 메서드의 이름은 Lambda가 함수를 호출하는 데 사용하는 핸들러 문자열에 속합니다. 이 `FunctionHandlerAsync` 메서드는 인바운드 이벤트 페이로드를 올바른 ASP.NET 형식으로 변환하고 ASP.NET 응답을 Lambda 응답 페이로드로 다시 변환합니다. 표시된 예제 `AspNetOnLambda` 프로젝트의 경우 핸들러 문자열은 다음과 같습니다.

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

Lambda에 API를 배포하려면 다음 명령을 실행하여 소스 코드 파일이 포함된 디렉토리로 이동하고 AWS CloudFormation을(를) 사용하여 함수를 배포하십시오.

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> **`dotnet lambda deploy-serverless`** 명령을 사용하여 API를 배포하는 경우 AWS CloudFormation에서는 배포 중에 지정한 스택 이름을 기반으로 Lambda 함수에 이름을 지정합

니다. Lambda 함수에 사용자 지정 이름을 지정하려면 `serverless.template` 파일을 편집하여 `AWS::Serverless::Function` 리소스에 `FunctionName` 속성을 추가하십시오. 자세한 내용은 AWS CloudFormation 사용 설명서에서 이름 유형을 참조하세요.

Lambda에 ASP.NET 최소 API 배포하기

Lambda에 ASP.NET 최소 API를 배포하려면.NET Lambda 템플릿을 사용하여 새로운 최소 API 프로젝트를 생성하시면 됩니다. 다음 명령을 사용하여 새 최소 API 프로젝트를 초기화합니다. 이 예제에서는 프로젝트 `MinimalApiOnLambda`의 이름을 지정합니다.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

이 명령은 프로젝트 디렉터리에서 다음 파일 및 디렉터리를 생성합니다.

```
### 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` 파일에는 다음 코드가 포함되어 있습니다.

```
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();
```

Lambda에서 실행하도록 최소 API를 구성하려면 Lambda와 ASP.NET Core 간의 요청 및 응답이 제대로 변환되도록 이 코드를 편집해야 할 수 있습니다. 기본적으로 함수는 REST API 이벤트 소스에 맞게 구성됩니다. HTTP API 또는 Application Load Balancer의 경우 다음 옵션 중 하나를 사용하여 (`LambdaEventSource.RestApi`)을(를) 대체합니다.

- (`LambdaEventSource.HttpAPi`)
- (`LambdaEventSource.ApplicationLoadBalancer`)

Lambda에 최소 API를 배포하려면 다음 명령을 실행하여 소스 코드 파일이 포함된 디렉토리로 이동하고 AWS CloudFormation을(를) 사용하여 함수를 배포하십시오.

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

컨테이너 이미지를 사용하여 .NET Lambda 함수 배포

.NET Lambda 함수의 컨테이너 이미지를 빌드하는 세 가지 방법이 있습니다.

- .NET용 AWS 기본 이미지 사용

  AWS 기본 이미지에는 언어 런타임, Lambda와 함수 코드 간의 상호 작용을 관리하는 런타임 인터페이스 클라이언트 및 로컬 테스트를 위한 런타임 인터페이스 에뮬레이터가 미리 로드되어 있습니다.

- AWS OS 전용 기본 이미지 사용

  AWS OS 전용 기본 이미지는 Amazon Linux 배포판 및 런타임 인터페이스 에뮬레이터를 포함합니다. 이러한 이미지는 일반적으로 Go 및 Rust와 같은 컴파일된 언어의 컨테이너 이미지와 Lambda가 기본 이미지를 제공하지 않는 언어 또는 언어 버전(예: Node.js 19)의 컨테이너 이미지를 생성하는 데 사용됩니다. OS 전용 기본 이미지를 사용하여 사용자 지정 런타임을 구현할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 .NET용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

- 비AWS 기본 이미지 사용

  Alpine Linux, Debian 등의 다른 컨테이너 레지스트리의 대체 기본 이미지를 사용할 수 있습니다. 조직에서 생성한 사용자 지정 이미지를 사용할 수도 있습니다. 이미지가 Lambda와 호환되도록 하려면 .NET용 런타임 인터페이스 클라이언트를 이미지에 포함해야 합니다.

> **Tip**
>
> Lambda 컨테이너 함수가 활성 상태가 되는 데 걸리는 시간을 줄이려면 Docker 설명서의 다단계 빌드 사용을 참조하세요. 효율적인 컨테이너 이미지를 빌드하려면 Dockerfile 작성 모범 사례를 따르세요.

이 페이지에서는 Lambda용 컨테이너 이미지를 빌드, 테스트 및 배포하는 방법을 설명합니다.

주제

- .NET용 AWS 기본 이미지
- .NET용 AWS 기본 이미지 사용
- 런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

.NET용 AWS 기본 이미지

AWS는 .NET에 대한 다음과 같은 기본 이미지를 제공합니다.

태그	런타임	운영 체제	Dockerfile	사용 중단
8	.NET 8	Amazon Linux 2023	GitHub의 .NET 8용 Dockerfile	
6	.NET 6	Amazon Linux 2	GitHub의 .NET 6용 Dockerfile	2024년 11월 12일

Amazon ECR 리포지토리: gallery.ecr.aws/lambda/dotnet

.NET용 AWS 기본 이미지 사용

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- .NET SDK - 다음 단계에서는 .NET 8 기본 이미지를 사용합니다. .NET 버전이 Dockerfile에 지정한 기본 이미지의 버전과 일치하는지 확인하세요.
- Docker

기본 이미지를 사용하여 이미지 생성 및 배포

다음 단계에서는 Amazon.Lambda.Templates와 Amazon.Lambda.Tools를 사용하여 .NET 프로젝트를 생성합니다. 그런 다음 도커 이미지를 빌드하고 Amazon ECR에 업로드하고 Lambda 함수에 배포합니다.

1. Amazon.Lambda.Templates NuGet 패키지를 설치합니다.

```
dotnet new install Amazon.Lambda.Templates
```

2. `lambda.image.EmptyFunction` 템플릿을 사용하여.NET 프로젝트를 생성합니다.

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

3. *MyFunction*/src/*MyFunction* 디렉터리로 이동합니다. 여기에 프로젝트 파일이 저장됩니다. 다음 파일을 검사합니다.

   - aws-lambda-tools-defaults.json – Lambda 함수를 배포할 때 이 파일에 명령줄 옵션을 지정합니다.
   - Function.cs - Lambda 핸들러 함수 코드입니다. 기본 `Amazon.Lambda.Core` 라이브러리와 기본 `LambdaSerializer` 속성을 포함하는 C# 템플릿입니다. 직렬화 요구 사항들과 옵션에 대한 자세한 내용은 Lambda 함수의 직렬화 섹션을 참조하세요. 제공된 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.
   - MyFunction.csproj - 애플리케이션을 구성하는 파일과 어셈블리가 나열된 .NET 프로젝트 파일입니다.
   - Readme.md - 이 파일에는 샘플 Lambda 함수에 대한 자세한 정보가 들어 있습니다.

4. `src/`*MyFunction* 디렉터리에서 Dockerfile을 검사합니다. 제공된 Dockerfile을 테스트에 사용하거나 사용자 고유의 Dockerfile로 바꿀 수 있습니다. 사용자 고유의 Dockerfile을 사용하는 경우 다음 사항을 확인하세요.

   - `FROM` 속성을 기본 이미지의 URI로 설정합니다. .NET 버전이 기본 이미지의 버전과 일치해야 합니다.
   - `CMD` 인수를 Lambda 함수 핸들러로 설정합니다. `aws-lambda-tools-defaults.json`의 `image-command`와 일치해야 합니다.

   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 .NET 글로벌 도구를 설치합니다.

   ```
   dotnet tool install -g Amazon.Lambda.Tools
   ```

   Amazon.Lambda.Tools가 이미 설치되어 있는 경우 최신 버전이 설치되어 있는지 확인합니다.

```
dotnet tool update -g Amazon.Lambda.Tools
```

6. 아직 이동하지 않은 경우 디렉터리를 *MyFunction*/src/*MyFunction*으로 변경합니다.

```
cd src/MyFunction
```

7. Amazon.Lambda.Tools를 사용하여 도커 이미지를 빌드하고 새 Amazon ECR 리포지토리로 푸시하고 Lambda 함수를 배포합니다.

   `--function-role`로 함수에 대한 실행 역할의 Amazon 리소스 이름(ARN)이 아닌 역할 이름을 지정합니다. 예를 들면 `lambda-role`입니다.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

   Amazon.Lambda.Tools .NET 글로벌에 대한 자세한 내용은 GitHub의 AWS .NET CLI용 확장 리포지토리를 참조하세요.

8. 함수를 호출합니다.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

   모든 것이 성공하면 다음을 볼 수 있습니다.

```
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 함수를 삭제합니다.

```
dotnet lambda delete-function MyFunction
```

런타임 인터페이스 클라이언트에서 대체 기본 이미지 사용

OS 전용 기본 이미지나 대체 기본 이미지를 사용하는 경우 이미지에 런타임 인터페이스 클라이언트를 포함해야 합니다. 런타임 인터페이스 클라이언트는 Lambda와 함수 코드 간의 상호 작용을 관리하는 Lambda 런타임 API을 확장합니다.

다음 예제에서는 비 AWS 기본 이미지를 사용하여.NET용 컨테이너 이미지를 빌드하는 방법과 .NET용 Lambda 런타임 인터페이스 클라이언트인 Amazon.Lambda.RuntimeSupport 패키지를 추가하는 방법을 보여줍니다. 예제 Dockerfile에서는 Microsoft .NET 8 기본 이미지를 사용합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- .NET SDK - 다음 단계에서는 .NET 8 기본 이미지를 사용합니다. .NET 버전이 Dockerfile에 지정한 기본 이미지의 버전과 일치하는지 확인하세요.
- Docker

대체 기본 이미지를 사용하여 이미지 생성 및 배포

1. Amazon.Lambda.Templates NuGet 패키지를 설치합니다.

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. `lambda.CustomRuntimeFunction` 템플릿을 사용하여.NET 프로젝트를 생성합니다. 이 템플릿에는 Amazon.Lambda.RuntimeSupport 패키지가 포함되어 있습니다.

   ```
   dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
   ```

3. *MyFunction*`/src/`*MyFunction* 디렉터리로 이동합니다. 여기에 프로젝트 파일이 저장됩니다. 다음 파일을 검사합니다.

   - aws-lambda-tools-defaults.json – Lambda 함수를 배포할 때 이 파일에 명령줄 옵션을 지정합니다.
   - Function.cs - 이 코드에는 `Amazon.Lambda.RuntimeSupport` 라이브러리를 부트스트랩으로 초기화하는 `Main` 메서드가 있는 클래스가 포함되어 있습니다. `Main` 메서드는 함수 프로세스의 진입점입니다. `Main` 메서드는 부트스트랩이 작업할 수 있는 래퍼로 함수 처리기를 래핑합니다. 자세한 내용은 GitHub 리포지토리의 Using Amazon.Lambda.RuntimeSupport as a class library를 참조하세요.

- MyFunction.csproj - 애플리케이션을 구성하는 파일과 어셈블리가 나열된 .NET 프로젝트 파일입니다.
- Readme.md - 이 파일에는 샘플 Lambda 함수에 대한 자세한 정보가 들어 있습니다.

4. `aws-lambda-tools-defaults.json` 파일을 열고 다음 줄을 추가합니다.

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

- package-type: 배포 패키지를 컨테이너 이미지로 정의합니다.
- docker-host-build-output-dir: 빌드 프로세스의 출력 디렉터리를 설정합니다.

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* 디렉터리에 Dockerfile을 생성합니다. 다음 예제 Dockerfile은 AWS 기본 이미지 대신 Microsoft .NET 기본 이미지를 사용합니다.

- FROM 속성을 기본 이미지 식별자로 설정합니다. .NET 버전이 기본 이미지의 버전과 일치해야 합니다.

- COPY 명령을 사용하여 함수를 /var/task 디렉터리에 복사합니다.
- Docker 컨테이너가 시작될 때 실행할 모듈로 ENTRYPOINT를 설정합니다. 이 경우 모듈은 Amazon.Lambda.RuntimeSupport 라이브러리를 초기화하는 부트스트랩입니다.

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 .NET 글로벌 도구 확장을 설치합니다.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools가 이미 설치되어 있는 경우 최신 버전이 설치되어 있는지 확인합니다.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Amazon.Lambda.Tools를 사용하여 도커 이미지를 빌드하고 새 Amazon ECR 리포지토리로 푸시하고 Lambda 함수를 배포합니다.

--function-role로 함수에 대한 실행 역할의 Amazon 리소스 이름(ARN)이 아닌 역할 이름을 지정합니다. 예를 들면 lambda-role입니다.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

Amazon.Lambda.Tools .NET CLI 확장에 대한 자세한 내용은 GitHub의 AWS Extensions for .NET CLI 리포지토리를 참조하세요.

8. 함수를 호출합니다.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

모든 것이 성공하면 다음을 볼 수 있습니다.

```
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 함수를 삭제합니다.

```
dotnet lambda delete-function MyFunction
```

네이티브 런타임 형식으로 .NET Lambda 함수 코드 컴파일

.NET 8은 네이티브 사전(AOT) 컴파일을 지원합니다. 네이티브 AOT를 사용하면 Lambda 함수 코드를 네이티브 런타임 형식으로 컴파일할 수 있으므로 런타임에 .NET 코드를 컴파일할 필요가 없습니다. 네이티브 AOT 컴파일은 .NET에서 작성하는 Lambda 함수의 콜드 스타트 시간을 줄일 수 있습니다. 자세한 내용은 AWS Compute Blog의 Introducing the .NET 8 runtime for AWS Lambda를 참조하세요.

Sections

Lambda 런타임

네이티브 AOT 컴파일을 사용하여 Lambda 함수 빌드를 배포하려면 관리형 .NET 8 Lambda 런타임을 사용합니다. 이 런타임은 x86_64 및 arm64 아키텍처의 사용을 모두 지원합니다.

AOT를 사용하지 않고 .NET Lambda 함수를 배포할 경우 애플리케이션이 먼저 중간 언어(IL) 코드로 컴파일됩니다. 런타임의 Just-in-Time(JIT) 컴파일러는 Lambda 런타임에 IL 코드를 가져와 필요에 따라 기계 코드로 컴파일합니다. 네이티브 AOT로 미리 컴파일된 Lambda 함수를 사용하면 함수를 배포할 때 코드를 머신 코드로 컴파일하므로 실행 전에 코드를 컴파일하기 위해 Lambda 런타임의 .NET 런타임이나 SDK에 의존하지 않습니다.

AOT의 한 가지 제한 사항은 .NET 8 런타임이 사용하는 것과 동일한 Amazon Linux 2023(AL2023) 운영 체제가 있는 환경에서 애플리케이션 코드를 컴파일해야 한다는 것입니다. .NET Lambda CLI는 AL2023 이미지를 사용하여 Docker 컨테이너에서 애플리케이션을 컴파일하는 기능을 제공합니다.

크로스 아키텍처 호환성과 관련된 잠재적인 문제를 방지하려면 함수에 대해 구성한 것과 동일한 프로세서 아키텍처를 사용하는 환경에서 코드를 컴파일하는 것이 좋습니다. 크로스 아키텍처 컴파일의 제한 사항에 대해 자세히 알아보려면 Microsoft .NET 설명서의 교차 컴파일을 참조하세요.

필수 조건

Docker

네이티브 AOT를 사용하려면 .NET 8 런타임과 동일한 AL2023 운영 체제가 있는 환경에서 함수 코드를 컴파일해야 합니다. 다음 섹션의 .NET CLI 명령은 도커를 사용하여 AL2023 환경에서 Lambda 함수를 개발하고 빌드합니다.

.NET 8 SDK

네이티브 AOT 컴파일은 .NET 8의 기능입니다. 빌드 머신에 런타임뿐만 아니라 .NET 8 SDK도 설치해야 합니다.

Amazon.Lambda.Tools

Lambda 함수를 생성하려면 Amazon.Lambda.Tools .NET 글로벌 도구 확장을 사용합니다. Amazon.Lambda.Tools를 설치하려면 다음 명령을 실행합니다.

```
dotnet tool install -g Amazon.Lambda.Tools
```

Amazon.Lambda.Tools .NET CLI 확장에 대한 자세한 내용은 GitHub에서 AWS Extensions for .NET CLI 리포지토리를 참조하세요.

Amazon.Lambda.Templates

Lambda 함수 코드를 생성하려면 Amazon.Lambda.Templates NuGet 패키지를 사용합니다. 이 템플릿 패키지를 설치하려면 다음 명령을 실행합니다.

```
dotnet new install Amazon.Lambda.Templates
```

시작하기

.NET 글로벌 CLI와 AWS Serverless Application Model(AWS SAM) 모두 네이티브 AOT를 사용하여 애플리케이션을 빌드하기 위한 시작 템플릿을 제공합니다. 첫 번째 네이티브 AOT Lambda 함수를 빌드하려면 다음 지침의 단계를 수행하십시오.

네이티브 AOT 컴파일된 Lambda 함수를 초기화하고 배포하기

1. 네이티브 AOT 템플릿을 사용하여 새 프로젝트를 초기화한 다음 생성된 `.cs` 및 `.csproj` 파일이 포함된 디렉토리로 이동합니다. 이 예제에서는 함수 `NativeAotSample`의 이름을 지정합니다.

```
dotnet new lambda.NativeAOT -n NativeAotSample
cd ./NativeAotSample/src/NativeAotSample
```

네이티브 AOT 템플릿으로 생성한 Function.cs 파일에는 다음 함수 코드가 포함되어 있습니다.

```
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
```

네이티브 AOT는 애플리케이션을 단일 네이티브 바이너리로 컴파일합니다. 해당 바이너리의 엔트리포인트는 `static Main` 메서드입니다. `static Main` 내에서 Lambda 런타임이 부트스트랩되고 `FunctionHandler` 메서드가 설정됩니다. 런타임 부트스트랩의 일부로 소스 생성 시리얼라이저는 `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()`을(를) 사용하여 구성됩니다

2. Lambda에 애플리케이션을 배포하려면 도커가 로컬 환경에서 실행되고 있는지 확인하고 다음 명령을 실행하십시오.

```
dotnet lambda deploy-function
```

.NET 글로벌 CLI는 백그라운드에서 AL2023 도커 이미지를 다운로드하고 실행 중인 컨테이너 내에서 애플리케이션 코드를 컴파일합니다. 컴파일된 바이너리는 Lambda에 배포되기 전에 로컬 파일 시스템으로 다시 출력됩니다.

3. 다음 명령을 실행하여 함수를 테스트하십시오. `<FUNCTION_NAME>`을(를) 배포 마법사에서 함수에 선택한 이름으로 바꾸십시오.

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

CLI의 응답에는 콜드 시작(초기화 기간)에 대한 성과 세부 정보와 함수 간접 호출의 총 실행 시간이 포함됩니다.

4. 이전 단계에 따라 생성한 AWS 리소스를 삭제하려면 다음 명령을 실행하십시오. `<FUNCTION_NAME>`을(를) 배포 마법사에서 함수에 선택한 이름으로 바꾸십시오. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 청구되는 것을 방지할 수 있습니다.

```
dotnet lambda delete-function <FUNCTION_NAME>
```

직렬화

네이티브 AOT를 사용하여 Lambda에 함수를 배포하려면 함수 코드가 소스 생성 직렬화를 사용해야 합니다. 소스 생성기는 런타임 리플렉션을 사용하여 직렬화를 위해 객체 속성에 액세스하는 데 필요한 메타데이터를 수집하는 대신 애플리케이션을 빌드할 때 컴파일되는 C# 소스 파일을 생성합니다. 소스 생성 시리얼라이저를 올바르게 구성하려면 함수에서 사용하는 입력 및 출력 객체와 사용자 지정 유형을

포함해야 합니다. 예를 들어 API Gateway REST API에서 이벤트를 수신하고 사용자 지정 `Product` 유형을 반환하는 Lambda 함수에는 다음과 같이 정의된 시리얼라이저가 포함됩니다.

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

트리밍

네이티브 AOT는 컴파일 과정에서 애플리케이션 코드를 트리밍하여 바이너리가 최대한 작아지도록 합니다. .NET 8 for Lambda는 이전 버전의 .NET에 비해 향상된 트리밍 지원을 제공합니다. Lambda 런타임 라이브러리, AWS .NET SDK, .NET Lambda Annotations 및 .NET 8 자체에 대한 지원이 추가되었습니다.

이러한 개선 사항을 통해 빌드 시간 트리밍 경고를 제거할 수 있지만 .NET이 트림으로부터 완전히 안전하다고 할 수는 없습니다. 즉, 함수가 사용되는 라이브러리의 일부는 일부 컴파일 단계에서 제거될 수도 있습니다. 다음 예제와 같이 `TrimmerRootAssemblies`를 `.csproj` 파일의 일부로 정의하여 이를 관리할 수 있습니다.

```
<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>
```

트리밍 경고를 받은 경우 `TrimmerRootAssembly`에 경고를 생성하는 클래스를 추가해도 문제가 해결되지 않을 수 있습니다. 트리밍 경고는 클래스가 런타임까지 확인할 수 없는 다른 클래스에 액세스하려고 함을 나타냅니다. 런타임 오류를 방지하려면 `TrimmerRootAssembly`에 이 두 번째 클래스를 추가합니다.

트리밍 경고 관리에 대해 자세히 알아보려면 Microsoft .NET 설명서의 트리밍 경고 소개를 참조하세요.

문제 해결

오류: OS 간 네이티브 컴파일은 지원되지 않습니다.

Amazon.Lambda.Tools .NET Core 글로벌 도구 버전이 오래되었습니다. 최신 버전으로 업데이트한 후 다시 시도하세요.

도커: 이 플랫폼에서는 이미지 운영 체제 'linux'를 사용할 수 없습니다.

시스템의 도커는 Windows 컨테이너를 사용하도록 구성되어 있습니다. 네이티브 AOT 빌드 환경을 실행하려면 Linux 컨테이너로 전환하세요.

일반적인 오류에 대한 자세한 내용은 GitHub에서 AWS NativeAOT for .NET 리포지토리를 참조하세요.

AWS Lambda 컨텍스트 객체(C#)

Lambda는 함수를 실행할 때 컨텍스트 객체를 핸들러에 전달합니다. 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 속성에 제공합니다.

컨텍스트 속성

- `FunctionName` – Lambda 함수의 이름입니다.
- `FunctionVersion` – 함수의 버전입니다.
- `InvokedFunctionArn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `MemoryLimitInMB` – 함수에 할당된 메모리의 양입니다.
- `AwsRequestId` – 호출 요청의 식별자입니다.
- `LogGroupName` – 함수에 대한 로그 그룹입니다.
- `LogStreamName` – 함수 인스턴스에 대한 로그 스트림입니다.
- `RemainingTime` (`TimeSpan`) – 실행 시간이 초과되기 전에 남은 시간(밀리초)입니다.
- `Identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
- `ClientContext` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.
- `Logger` 함수에 대한 로거 객체입니다.

`ILambdaContext` 객체의 정보를 사용하여 모니터링 목적으로 함수의 간접 호출에 대한 정보를 출력할 수 있습니다. 다음 코드는 구조화된 로깅 프레임워크에 컨텍스트 정보를 추가하는 방법의 예입니다. 이 예제에서는 함수가 `AwsRequestId`을(를) 로그 출력에 추가합니다. 또한 함수는 Lambda 함수 제한 시간에 도달하려는 경우 `RemainingTime` 속성을 사용하여 진행 중인 작업도 취소합니다.

```
[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)
            };
        }
    }
}
```

Lambda 함수 로깅(C#)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그 항목을 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 간접 호출에 대한 세부 정보와 함수 코드의 기타 출력을 로그 스트림으로 전송합니다. CloudWatch Logs에 대한 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 섹션을 참조하세요.

단원

- 로그를 반환하는 함수 생성
- 도구 및 라이브러리
- 구조화된 로깅에 Powertools forAWS Lambda(.NET) 및 AWS SAM 사용
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제

로그를 반환하는 함수 생성

함수 코드의 로그를 출력하려면 콘솔 클래스에서 메서드를 사용하거나, `stdout` 또는 `stderr`에 쓰는 로깅 라이브러리를 사용합니다.

.NET 런타임은 각 호출에 대해 `START`, `END` 및 `REPORT` 줄을 로깅합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.
- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.

- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

도구 및 라이브러리

Powertools for AWS Lambda(.NET)는 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다. Logging 유틸리티는 JSON으로 구조화된 출력과 함께 모든 함수의 함수 컨텍스트에 대한 추가 정보를 포함하는 Lambda 최적화 로거를 제공합니다. 이 유틸리티를 사용하여 다음을 수행합니다.

- Lambda 컨텍스트, 콜드 스타트 및 구조 로깅 출력에서 JSON으로 주요 필드 캡처
- 지시 시 Lambda 호출 이벤트 로깅(기본적으로 비활성화됨)
- 로그 샘플링을 통해 호출 비율에 대해서만 모든 로그 인쇄(기본적으로 비활성화됨)
- 언제든지 구조화된 로그에 추가 키 추가
- 사용자 지정 로그 포맷터(Bring Your Own Formatter)를 사용하여 조직의 로깅 RFC와 호환되는 구조로 로그 출력

구조화된 로깅에 Powertools forAWS Lambda(.NET) 및 AWS SAM 사용

다음 단계를 따라 AWS SAM을 사용하는 통합 Powertools for AWS Lambda(.NET) 모듈을 사용하여 샘플 Hello World C# 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 이 함수는 `hello world` 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- .NET 6 또는 .NET 8
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World TypeScript 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 로그를 가져오려면 sam logs를 실행합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서에서 로그 관련 작업을 참조하세요.

```
sam logs --stack-name sam-app
```

출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

로그 보존 관리

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 경과 후 CloudWatch가 로그를 자동으로 삭제하는 보존 기간을 구성하세요. 로그 보존을 설정하려면 AWS SAM 템플릿에 다음을 추가합니다.

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - aws configure을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 `base64` 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 `my-function`에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한

내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```


로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

AWS Lambda에서 C# 코드 계측

Lambda는 AWS X-Ray와 통합되어 Lambda 애플리케이션을 추적, 디버깅 및 최적화할 수 있습니다. Lambda 함수와 기타 AWS 서비스를 포함할 수 있는 애플리케이션의 리소스를 탐색할 때 X-Ray를 사용하여 요청을 추적할 수 있습니다.

추적 데이터를 X-Ray로 전송하려면 다음 세 SDK 라이브러리 중 하나를 사용할 수 있습니다.

- AWS Distro for OpenTelemetry(ADOT) - 안전하게 프로덕션 준비가 된 AWS에서 지원하는 OpenTelemetry(OTEL) SDK의 배포입니다.
- AWS X-Ray SDK for .NET — 추적 데이터를 생성하고 X-Ray에 전송하는 SDK입니다.
- Powertools for AWS Lambda(.NET) – 서버리스 모범 사례를 구현하고 개발자 속도를 높이기 위한 개발자 도구 키트입니다.

각 SDK는 텔레메트리 데이터를 X-Ray 서비스로 전송하는 방법을 제공합니다. X-Ray를 사용하여 애플리케이션의 성능 지표를 확인하고, 필터링하고, 인사이트를 얻어 문제와 최적화 기회를 식별할 수 있습니다.

> **Important**
>
> X-Ray와 Powertools for AWS Lambda SDK는 AWS에서 제공하는 긴밀하게 통합된 계측 솔루션의 일부입니다. ADOT Lambda Layer는 일반적으로 더 많은 데이터를 수집하는 추적 계측기에 대한 전체 업계 표준의 일부이지만 모든 사용 사례에 적합하지는 않을 수 있습니다. 어떤 솔루션을 사용하든 X-Ray에서 엔드 투 엔드 추적 기능을 구현할 수 있습니다. 둘 중 하나를 선택하는 방법에 대해 자세히 알아보려면 AWS Distro for Open Telemetry와 X-Ray SDK 중에서 선택하기를 참조하세요.

Sections

추적에 Powertools for AWS Lambda(.NET) 및 AWS SAM 사용

다음 단계를 따라 AWS SAM을 사용하는 통합 Powertools for AWS Lambda(.NET) 모듈을 사용하여 샘플 Hello World C# 애플리케이션을 다운로드, 빌드 및 배포합니다. 이 애플리케이션은 기본 API 백엔드를 구현하고 Powertools를 사용하여 로그, 지표 및 추적을 내보냅니다. 이 구성에는 Amazon API Gateway 엔드포인트와 Lambda 함수가 포함됩니다. API Gateway 엔드포인트로 GET 요청을 전송하면 Lambda 함수가 호출되고 Embedded Metric Format을 사용하여 로그 및 지표를 CloudWatch로 전송하고 기록을 AWS X-Ray로 전송합니다. 함수가 hello world 메시지를 반환합니다.

필수 조건

이 섹션의 단계를 완료하려면 다음이 필요합니다.

- .NET 6 또는 .NET 8
- AWS CLI 버전 2
- AWS SAM CLI 버전 1.75 이상. 이전 버전의 AWS SAM CLI가 있는 경우 AWS SAM CLI 업그레이드를 참조하세요.

샘플 AWS SAM 애플리케이션 배포

1. Hello World TypeScript 템플릿을 사용하여 애플리케이션을 초기화합니다.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 앱을 빌드합니다.

```
cd sam-app && sam build
```

3. 앱을 배포합니다.

```
sam deploy --guided
```

4. 화면에 표시되는 프롬프트를 따릅니다. 대화형 환경에서 제공되는 기본 옵션을 수락하려면 `Enter`을 누릅니다.

> **Note**
>
> HelloWorldFunction에 권한 부여가 정의되어 있지 않을 수 있습니다. 괜찮습니다?에 대해 y를 입력합니다.

5. 배포된 애플리케이션의 URL을 가져옵니다.

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. API 엔드포인트 호출:

```
curl <URL_FROM_PREVIOUS_STEP>
```

성공하면 다음과 같은 결과가 응답됩니다.

```
{"message":"hello world"}
```

7. 함수에 대한 트레이스를 가져오려면 sam traces를 실행합니다.

```
sam traces
```

추적 출력은 다음과 같습니다.

```
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. 이는 인터넷을 통해 액세스할 수 있는 퍼블릭 API 엔드포인트입니다. 테스트 후에는 엔드포인트를 삭제하는 것이 좋습니다.

```
sam delete
```

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray SDK를 사용하여 .NET 함수 계측

함수 코드를 계측하여 메타데이터를 기록하고 다운스트림 호출을 추적할 수 있습니다. 함수가 다른 리소스 및 서비스에 대해 수행하는 호출과 관련된 세부 정보를 기록하려면 AWS X-Ray SDK for .NET을(를) 사용합니다. SDK를 가져오려면 프로젝트 파일에 `AWSXRayRecorder` 패키지를 추가합니다.

```
<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, 엔터티 프레임워크 및 HTTP 요청에 대한 자동 계측을 제공하는 다양한 Nuget 패키지가 있습니다. 전체 구성 옵션 세트를 확인하려면 AWS X-Ray 개발자 안내서의 .NET용 AWS X-Ray SDK를 참조하십시오.

원하는 Nuget 패키지를 추가한 후 자동 계측을 구성하십시오. 함수의 핸들러 함수 외부에서 이 구성을 수행하는 것이 좋습니다. 실행 환경 재사용을 활용하여 함수 성능을 향상시킬 수 있습니다. 다음 코드 예제에서는 함수 생성자에서 `RegisterXRayForAllServices` 메서드를 호출하여 모든 AWS SDK 호출에 대한 계측을 추가합니다.

```
[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 콘솔을 사용하여 추적 활성화

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

Lambda API를 사용하여 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 Lambda 함수에 대한 추적을 구성하고 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation을 사용하여 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

X-Ray 추적 해석

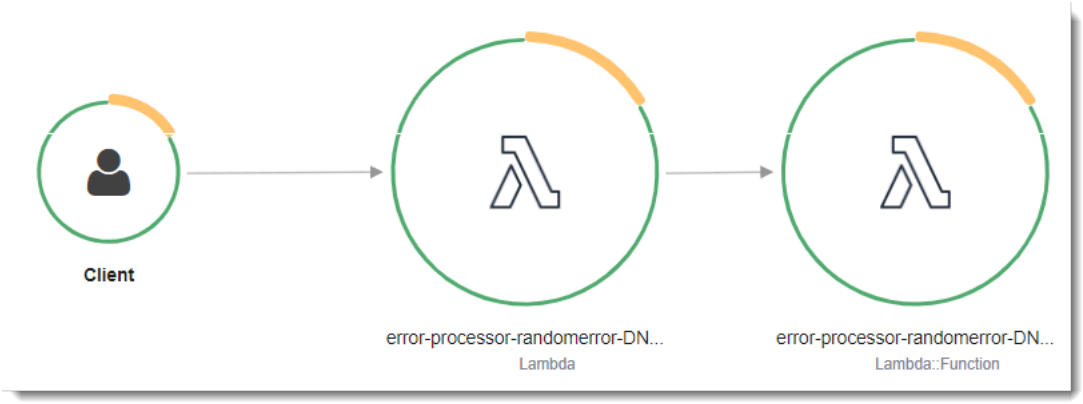
함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

활성 추적을 구성하면 애플리케이션을 통해 특정 요청을 관찰할 수 있습니다. X-Ray 서비스 그래프는 애플리케이션 및 모든 구성 요소에 대한 정보를 보여줍니다. 다음 이미지에서는 두 가지 함수와 함께 애플리케이션을 보여줍니다. 기본 함수는 이벤트를 처리하고 때로는 오류를 반환합니다. 맨 위의 두 번째 함수는 첫 번째의 로그 그룹에 나타나는 오류를 처리하고 AWS SDK를 사용하여 X-Ray, Amazon Simple Storage Service(Amazon S3), Amazon CloudWatch Logs를 호출합니다.

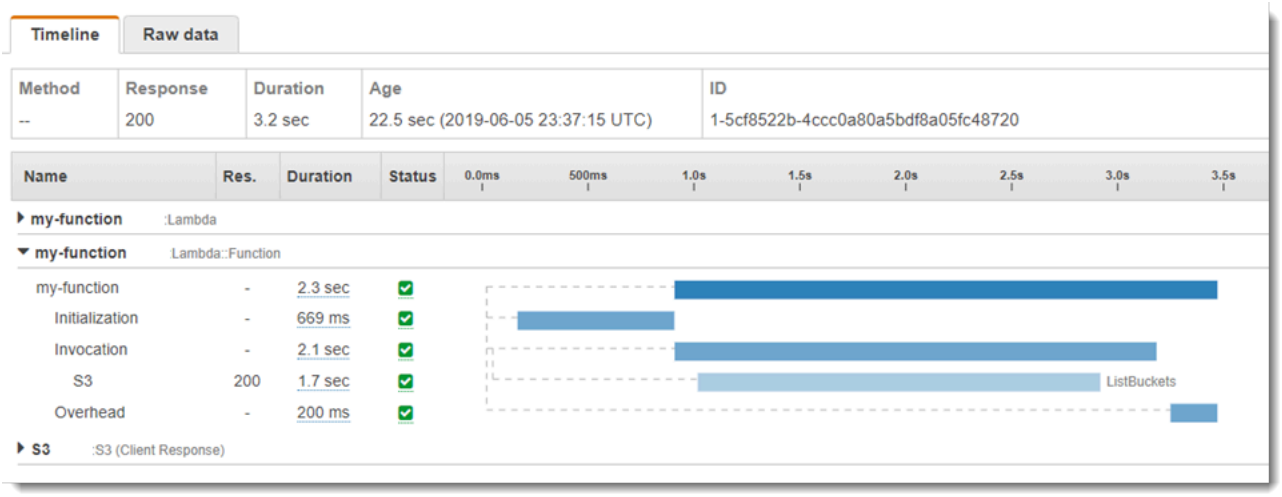
X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에 문제가 있는 것입니다. `AWS::Lambda::Function` 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



이 예에서는 3개의 하위 세그먼트를 표시하도록 `AWS::Lambda::Function` 세그먼트를 확장합니다.

- 초기화 – 함수를 로드하고 초기화 코드를 실행하는 데 소요된 시간을 나타냅니다. 이 하위 세그먼트는 함수의 각 인스턴스에서 처리하는 첫 번째 이벤트에 대해서만 표시됩니다.
- 호출— 핸들러 코드를 실행하는 데 소요된 시간을 나타냅니다.

- 오버헤드 – Lambda 런타임이 다음 이벤트를 처리하기 위해 준비하는 데 소비하는 시간을 나타냅니다.

HTTP 클라이언트를 계측하고, SQL 쿼리를 기록하고, 주석 및 메타데이터가 있는 사용자 지정 하위 세그먼트를 생성할 수도 있습니다. 자세한 내용은 AWS X-Ray 개발자 안내서의 AWS X-Ray SDK for .NET를 참조하십시오.

> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

C#의 AWS Lambda 함수 테스트

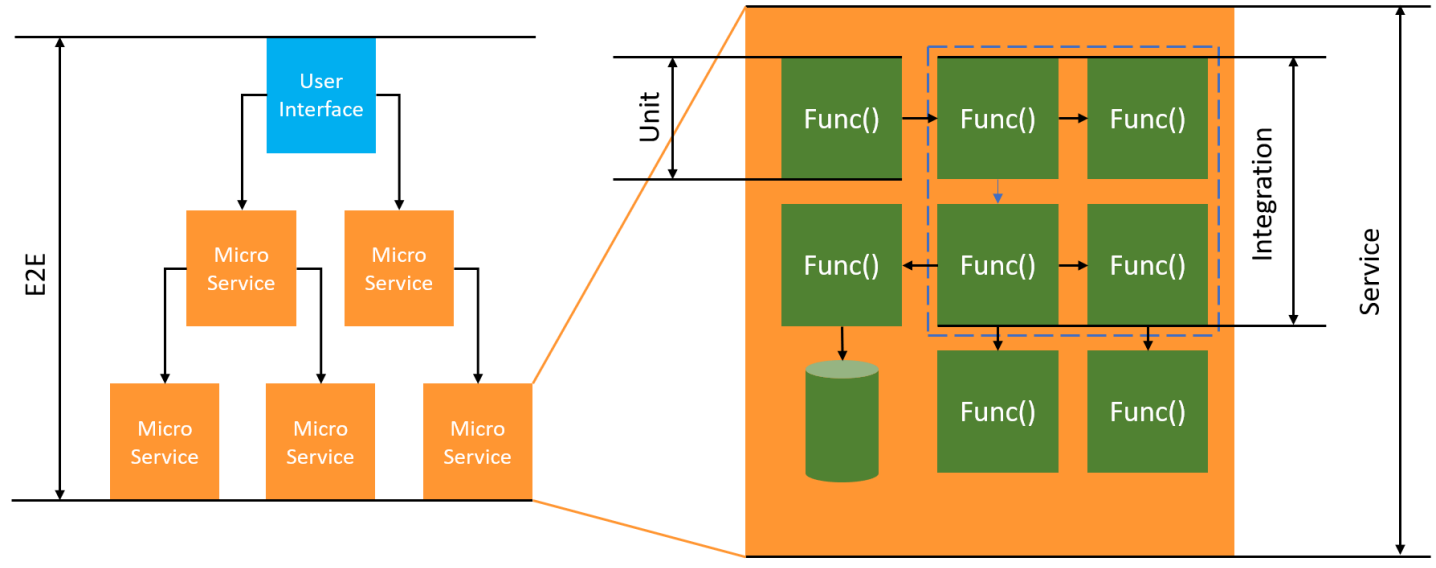
> **Note**
>
> 서버리스 솔루션 테스트를 위한 기술 및 모범 사례에 대한 전체 소개는 함수 테스트 장을 참조하세요.

서버리스 함수 테스트는 기존 테스트 유형과 기법을 사용하지만, 서버리스 애플리케이션을 전체적으로 테스트하는 것도 고려해야 합니다. 클라우드 기반 테스트는 함수와 서버리스 애플리케이션 모두의 품질을 가장 정확하게 측정합니다.

서버리스 애플리케이션 아키텍처에는 API 호출을 통해 중요한 애플리케이션 기능을 제공하는 관리형 서비스가 포함됩니다. 따라서 개발 주기에는 함수와 서비스가 상호 작용할 때 기능을 확인하는 자동화된 테스트가 포함되어야 합니다.

클라우드 기반 테스트를 생성하지 않으면 로컬 환경과 배포된 환경 간의 차이로 인해 문제가 발생할 수 있습니다. 지속적 통합 프로세스는 QA, 스테이징 또는 프로덕션과 같은 다음 배포 환경으로 코드를 승격하기 전에 클라우드에서 프로비저닝되는 리소스 제품군을 대상으로 테스트를 실행해야 합니다.

이 짧은 안내서를 계속 읽고 서버리스 애플리케이션의 테스트 전략에 대해 알아보거나 Serverless Test Samples 리포지토리를 방문하여 선택한 언어 및 런타임과 관련된 실제 예제를 자세히 살펴보세요.



서버리스 테스트의 경우 여전히 단위, 통합 및 end-to-end테스트를 작성해야 합니다.

- 단위 테스트 - 격리된 코드 블록에 대해 실행되는 테스트입니다. 예를 들어, 특정 항목과 대상에 대한 배송료를 계산하는 비즈니스 로직을 확인합니다.
- 통합 테스트 - 일반적으로 클라우드 환경에서 상호 작용하는 둘 이상의 구성 요소 또는 서비스를 포함하는 테스트입니다. 예를 들어, 함수가 대기열에서 이벤트를 처리하는지 확인합니다.
- E nd-to-end 테스트 - 전체 애플리케이션의 동작을 확인하는 테스트입니다. 예를 들어, 인프라가 올바르게 설정되어 있고 고객의 주문 기록을 위해 예상대로 서비스 간에 이벤트가 흐르는지 확인합니다.

서버리스 애플리케이션 테스트

일반적으로 다양한 접근 방식을 사용하여 클라우드에서 테스트, 모의 객체로 테스트, 에뮬레이터로 테스트 등의 서버리스 애플리케이션 코드 테스트를 수행합니다.

클라우드에서 테스트

클라우드에서의 테스트는 단위 테스트, 통합 테스트, 테스트를 포함한 테스트의 모든 단계에서 유용합니다. end-to-end 클라우드에 배포되고 클라우드 기반 서비스와 상호 작용하는 코드에 대해 테스트를 실행합니다. 이 접근 방식은 코드 품질을 가장 정확하게 측정합니다.

클라우드에서 Lambda 함수를 디버깅하는 편리한 방법은 테스트 이벤트와 콘솔을 이용하는 것입니다. 테스트 이벤트는 함수에 대한 JSON 입력입니다. 함수에 입력이 필요하지 않은 경우 이벤트는 빈 JSON 문서(({}))가 될 수 있습니다. 콘솔은 다양한 서비스 통합을 위한 샘플 이벤트를 제공합니다. 콘솔에서 이벤트를 생성한 후 팀과 공유하여 테스트를 더 쉽고 일관성 있게 만들 수 있습니다.

> **Note**
>
> 콘솔에서 함수를 테스트하는 것이 빠르게 시작할 수 있는 방법이지만 테스트 주기를 자동화하면 애플리케이션 품질과 개발 속도가 보장됩니다.

테스트 도구

개발 주기를 가속화하기 위해 함수를 테스트할 때 사용할 수 있는 다양한 도구와 기법이 있습니다. 예를 들어, AWS SAM 가속화와 AWS CDK 감시 모드 모두 클라우드 환경을 업데이트하는 데 필요한 시간을 줄입니다.

Lambda 함수 코드를 정의하는 방법에 따라 단위 테스트를 간단하게 추가할 수 있습니다. Lambda는 클래스를 초기화하기 위해 파라미터가 없는 퍼블릭 생성자가 필요합니다. 두 번째 내부 생성자를 도입하면 애플리케이션이 사용하는 종속성을 제어할 수 있습니다.

```
[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)
        };
    }
}
```

이 함수에 대한 테스트를 작성하려면 `Function` 클래스의 새 인스턴스를 초기화하고 `IDatabaseRepository`의 모의 구현을 전달하면 됩니다. 아래 예제에서는 `XUnit`, `Moq`, 및 `FluentAssertions`을(를) 사용하여 `FunctionHandler`이(가) 200 상태 코드를 반환하는지 확인하는 간단한 테스트를 작성합니다.

```
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);
    }
}
```

비동기 테스트의 예를 비롯한 자세한 예제는 의 .NET 테스트 샘플 리포지토리를 참조하십시오. GitHub

PowerShell을 사용하여 Lambda 함수 빌드

다음 단원에서는 일반적인 프로그래밍 패턴 및 핵심 개념이 PowerShell로 Lambda 함수 코드를 작성할 때 어떻게 적용되는지 설명합니다.

Lambda는 PowerShell에 대해 다음과 같은 샘플 애플리케이션을 제공합니다.

- blank-powershell – 로깅, 환경 변수 및 AWS SDK를 사용하는 방법을 보여주는 PowerShell 함수입니다.

시작하기 전에 PowerShell 개발 환경을 먼저 설정해야 합니다. 작업 방법에 대한 지침은 PowerShell 개발 환경 설정 단원을 참조하십시오.

AWSLambdaPSCore 모듈을 사용해 템플릿에서 샘플 PowerShell 프로젝트를 다운로드하고, PowerShell 배포 패키지를 생성하고, AWS 클라우드에 PowerShell 함수를 배포하는 방법을 알아보려면 .zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포 단원을 참조하세요.

Lambda는 .NET 언어에 대해 다음과 같은 런타임을 제공합니다.

.NET

명칭	식별자	운영 체제	사용 중단 날짜	블록 함수 생성	블록 함수 업데이트
.NET 8	dotnet8	Amazon Linux 2023			
.NET 6	dotnet6	Amazon Linux 2	2024년 11월 12일	2025년 2월 28일	2025년 3월 31일

주제

- PowerShell 개발 환경 설정
- .zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포
- PowerShell에서 Lambda 함수 핸들러 정의
- AWS Lambda컨텍스트 객체 입력 PowerShell
- AWS Lambda 함수 로깅(PowerShell)

PowerShell 개발 환경 설정

Lambda는 런타임을 위한 도구 및 라이브러리 세트를 제공합니다. PowerShell 설치 지침은 on용 Lambda 도구를 참조하십시오. PowerShell GitHub

이 AWSLambdaPSCore 모듈에는 Lambda PowerShell 함수를 작성하고 게시하는 데 도움이 되는 다음과 같은 cmdlet이 포함되어 있습니다.

- Get- AWSPowerShellLambdaTemplate - 시작 템플릿 목록을 반환합니다.
- 신규- AWSPowerShellLambda - 템플릿을 기반으로 초기 PowerShell 스크립트를 만듭니다.
- 게시- AWSPowerShellLambda - 지정된 PowerShell 스크립트를 Lambda에 게시합니다.
- 신규- AWSPowerShellLambdaPackage — CI/CD 시스템에서 배포에 사용할 수 있는 Lambda 배포 패키지를 생성합니다.

.zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포

PowerShell 런타임용 배포 패키지에는 PowerShell 스크립트, 스크립트에 필요한 PowerShell 모듈, Core를 PowerShell 호스팅하는 데 필요한 어셈블리가 포함되어 있습니다. PowerShell

Lambda 함수 생성

Lambda로 PowerShell 스크립트 작성 및 호출을 시작하려면 cmdlet을 사용하여 New-AWSPowerShellLambda 템플릿을 기반으로 시작 스크립트를 생성할 수 있습니다. Publish-AWSPowerShellLambda cmdlet을 사용해 스크립트를 Lambda에 배포할 수 있습니다. 그런 다음 명령줄 또는 Lambda 콘솔에서 스크립트를 테스트할 수 있습니다.

새 PowerShell 스크립트를 생성하고 업로드하고 테스트하려면 다음을 수행하십시오.

1. 사용 가능한 템플릿 목록을 보려면 다음 명령을 실행합니다.

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template                Description
--------                -----------
Basic                   Bare bones script
CodeCommitTrigger       Script to process AWS CodeCommit Triggers
...
```

2. Basic 템플릿을 토대로 샘플 스크립트를 생성하려면 다음 명령을 실행합니다.

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

현재 디렉터리의 새 하위 디렉터리에 MyFirstPSScript.ps1이라는 새 파일이 생성됩니다. 이 디렉터리의 이름은 -ScriptName 파라미터를 기반으로 지정됩니다. -Directory 파라미터를 사용하면 대체 디렉터리를 선택할 수 있습니다.

새 파일의 내용은 다음과 같습니다.

```
# 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. 스크립트의 로그 메시지가 Amazon CloudWatch Logs로 전송되는 방식을 확인하려면 샘플 PowerShell 스크립트에서 해당 `Write-Host` 줄의 주석 처리를 제거하십시오.

   Lambda 함수에서 데이터를 다시 반환하는 방법을 확인하려면 `$PSVersionTable`을 사용해 스크립트 끝에 새 행을 추가합니다. 이렇게 하면 PowerShell 파이프라인에 가 `$PSVersionTable` 추가됩니다. PowerShell 스크립트가 완료된 후 PowerShell 파이프라인의 마지막 객체는 Lambda 함수의 반환 데이터입니다. `$PSVersionTable`실행 환경에 대한 정보도 제공하는 PowerShell 글로벌 변수입니다.

   이와 같이 변경한 후 샘플 스크립트의 마지막 두 개 행은 다음과 같습니다.

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. `MyFirstPSScript.ps1` 파일 편집 후 디렉터리를 스크립트가 있는 위치로 변경합니다. 다음 명령을 실행하여 스크립트를 Lambda에 게시합니다.

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   `-Name` 파라미터는 Lambda 콘솔에 나타나는 Lambda 함수 이름을 지정합니다. 이 함수를 사용해 스크립트를 수동으로 호출할 수 있습니다.

5. AWS Command Line Interface (AWS CLI) `invoke` 명령을 사용하여 함수를 호출합니다.

   ```
   > aws lambda invoke --function-name MyFirstPSScript out
   ```

PowerShell에서 Lambda 함수 핸들러 정의

Lambda 함수가 호출되면 Lambda 핸들러가 PowerShell 스크립트를 호출합니다.

PowerShell 스크립트가 호출되면 다음 변수가 미리 정의되어 있습니다.

- *$LambdaInput* – 핸들러에 대한 입력이 포함된 PSObject입니다. 이 입력은 이벤트 데이터(이벤트 소스에서 게시)가 되거나, 문자열이나 사용자 지정 데이터 객체 같은 사용자 지정 입력이 될 수 있습니다.
- *$LambdaContext* – 현재 호출에 대한 정보(예: 현재 함수의 이름, 메모리 제한, 남아 있는 실행 시간, 로깅 등)에 액세스하는 데 사용할 수 있는 Amazon.Lambda.Core.ILambdaContext 객체입니다.

예를 들어 다음과 같은 PowerShell 예제 코드를 고려해 보세요.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

이 스크립트는 $LambdaContext 변수에서 얻은 FunctionName 속성을 반환합니다.

> **Note**
>
> 스크립트에서 사용하는 모듈을 나타내기 위해서는 PowerShell 스크립트 내에서 `#Requires` 문을 사용해야 합니다. 이 문은 두 가지 중요한 작업을 수행하는데 바로, 1) 다른 개발자에게 스크립트가 사용하는 모듈을 알리는 작업과 2) 배포의 일부로 스크립트를 패키징하는 데 AWS PowerShell 도구에 필요한 종속 모듈을 식별하는 작업입니다. PowerShell의 `#Requires` 문에 대한 자세한 내용은 Requires 정보를 참조하세요. PowerShell 배포 패키지에 대한 자세한 내용은 .zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포 단원을 참조하세요.
> PowerShell Lambda 함수가 AWS PowerShell cmdlet을 사용하는 경우 Windows PowerShell만 지원하는 `AWSPowerShell` 모듈이 아니라 PowerShell Core를 지원하는 `AWSPowerShell.NetCore` 모듈을 참조하는 `#Requires` 문을 설정해야 합니다. cmdlet 가져오기 프로세스를 최적화하는 `AWSPowerShell.NetCore` 버전 3.3.270.0 이상을 사용해야 합니다. 이전 버전을 사용하는 경우 콜드 부팅 시간이 더 길어집니다. 자세한 내용은 AWS Tools for PowerShell 페이지를 참조하세요.

데이터 반환

일부 Lambda 호출은 호출자에게 데이터가 반환됨을 의미합니다. 예를 들어, API Gateway의 웹 요청에 대한 응답으로 호출이 발생한 경우 Lambda 함수는 해당 응답을 다시 반환해야 합니다. PowerShell Lambda의 경우 PowerShell 파이프라인에 추가된 마지막 객체는 Lambda 호출의 반환 데이터입니다. 객체가 문자열인 경우 데이터는 있는 그대로 반환됩니다. 그렇지 않으면 `ConvertTo-Json` cmdlet을 사용해 객체가 JSON으로 변환됩니다.

예를 들어, PowerShell 파이프라인에 `$PSVersionTable`을 추가하는 다음 PowerShell 문에 대해 생각해 보세요.

```
$PSVersionTable
```

PowerShell 스크립트가 완성되면 PowerShell 파이프라인의 마지막 객체가 Lambda 함수의 반환 데이터입니다. `$PSVersionTable`은 실행 환경에 대한 정보도 제공하는 PowerShell 전역 변수입니다.

AWS Lambda컨텍스트 객체 입력 PowerShell

Lambda가 함수를 실행하면 `$LambdaContext` 변수를 핸들러에 제공하여 컨텍스트 정보를 전달합니다. 이 변수는 호출, 함수 및 실행 환경에 관한 정보를 메서드 및 속성에 제공합니다.

컨텍스트 속성

- `FunctionName` – Lambda 함수의 이름입니다.
- `FunctionVersion` – 함수의 버전입니다.
- `InvokedFunctionArn` – 함수를 호출할 때 사용하는 Amazon 리소스 이름(ARN)입니다. 호출자가 버전 번호 또는 별칭을 지정했는지 여부를 나타냅니다.
- `MemoryLimitInMB` – 함수에 할당된 메모리의 양입니다.
- `AwsRequestId` – 호출 요청의 식별자입니다.
- `LogGroupName` – 함수에 대한 로그 그룹입니다.
- `LogStreamName` – 함수 인스턴스에 대한 로그 스트림입니다.
- `RemainingTime` – 실행 시간이 초과되기 전에 남은 시간(밀리초)입니다.
- `Identity` – (모바일 앱) 요청을 승인한 Amazon Cognito 자격 증명에 대한 정보입니다.
- `ClientContext` – (모바일 앱) 클라이언트 애플리케이션이 Lambda에게 제공한 클라이언트 컨텍스트입니다.
- `Logger` - 함수에 대한 로거 객체입니다.

다음 PowerShell 코드 스니펫은 일부 컨텍스트 정보를 인쇄하는 간단한 핸들러 함수를 보여줍니다.

```
#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
```

AWS Lambda 함수 로깅(PowerShell)

AWS Lambda는 자동으로 Lambda 함수를 모니터링하고 로그를 Amazon CloudWatch로 보냅니다. Lambda 함수는 함수의 각 인스턴스에 대한 CloudWatch Logs 로그 그룹 및 로그 스트림과 함께 제공됩니다. Lambda 런타임 환경은 각 호출에 관한 세부 정보를 로그 스트림에 전송하며, 함수 코드에서 로그 및 그 외 출력을 중계합니다. 자세한 내용은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 단원을 참조하십시오.

이 페이지에서는 AWS Command Line Interface, Lambda 콘솔 또는 CloudWatch 콘솔을 사용하여 Lambda 함수 코드의 로그 출력이나 액세스 로그를 생성하는 방법에 대해 설명합니다.

단원

- 로그를 반환하는 함수 생성
- Lambda 콘솔 사용
- CloudWatch 콘솔 사용
- AWS Command Line Interface(AWS CLI) 사용
- 로그 삭제

로그를 반환하는 함수 생성

함수 코드에서 로그를 출력하려면 Microsoft.PowerShell.Utility에서 cmdlets를 사용하거나, `stdout` 또는 `stderr`에 쓰는 로깅 모듈을 사용합니다. 다음 예에는 `Write-Host`가 사용됩니다.

Example function/Handler.ps1 – 로깅

```
#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 로그 형식

```
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 런타임은 각 호출에 대해 START, END 및 REPORT 줄을 로깅합니다. 보고서 행은 다음과 같은 세부 정보를 제공합니다.

REPORT 행 데이터 필드

- RequestId – 호출의 고유한 요청 ID입니다.
- 지속시간 – 함수의 핸들러 메서드가 이벤트를 처리하는 데 걸린 시간입니다.
- 청구 기간 – 호출에 대해 청구된 시간입니다.
- 메모리 크기 - 함수에 할당된 메모리 양입니다.
- 사용된 최대 메모리 – 함수에서 사용한 메모리 양입니다.

- 초기화 기간 – 제공된 첫 번째 요청의 경우 런타임이 핸들러 메서드 외부에서 함수를 로드하고 코드를 실행하는 데 걸린 시간입니다.
- XRAY TraceId – 추적된 요청의 경우 AWS X-Ray 추적 ID입니다.
- SegmentId - 추적된 요청의 경우 X-Ray 세그먼트 ID입니다.
- 샘플링 완료(Sampled) – 추적된 요청의 경우 샘플링 결과입니다.

Lambda 콘솔 사용

Lambda 함수를 호출한 후 Lambda 콘솔을 사용하여 로그 출력을 볼 수 있습니다.

포함된 코드 편집기에서 코드를 테스트할 수 있는 경우 실행 결과에서 로그를 찾을 수 있습니다. 콘솔 테스트 기능을 사용하여 함수를 간접적으로 호출하면 세부 정보 섹션에서 로그 출력을 찾을 수 있습니다.

CloudWatch 콘솔 사용

Amazon CloudWatch 콘솔을 사용하여 모든 Lambda 함수 호출에 대한 로그를 볼 수 있습니다.

CloudWatch 콘솔에서 로그를 보려면

1. CloudWatch 콘솔에서 로그 그룹 페이지를 엽니다.
2. 함수(/aws/lambda/***your-function-name***)에 대한 로그 그룹을 선택합니다.
3. 로그 스트림을 선택합니다.

각 로그 스트림은 함수의 인스턴스에 해당합니다. 로그 스트림은 Lambda 함수를 업데이트할 때, 그리고 여러 동시 호출을 처리하기 위해 추가 인스턴스가 생성될 때 나타납니다. 특정 호출에 대한 로그를 찾으려면 AWS X-Ray로 함수를 계측하는 것이 좋습니다. X-Ray는 요청 및 로그 스트림에 대한 세부 정보를 기록합니다.

AWS Command Line Interface(AWS CLI) 사용

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - `aws configure`을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 `base64` 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 `my-function`에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다:

```
{
    "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"
}
```

로그 삭제

함수를 삭제해도 로그 그룹이 자동으로 삭제되지 않습니다. 로그를 무기한 저장하지 않으려면 로그 그룹을 삭제하거나 로그가 자동으로 삭제되는 보존 기간을 구성하세요.

Rust를 사용하여 Lambda 함수 빌드

Rust는 네이티브 코드로 컴파일되므로 Lambda에서 Rust 코드를 실행하는 데 전용 런타임이 필요하지 않습니다. 대신 Rust 런타임 클라이언트를 사용하여 로컬에서 프로젝트를 빌드한 다음 `provided.al2023` 또는 `provided.al2` 런타임을 사용하여 Lambda에 배포하세요. `provided.al2023` 또는 `provided.al2`를 사용하면 Lambda는 최신 패치를 통해 자동으로 운영 체제를 최신 상태로 유지합니다.

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

Rust용 도구 및 라이브러리

- AWS SDK for Rust: AWS SDK for Rust는 Amazon Web Services 인프라 서비스와 상호 작용할 수 있는 Rust API를 제공합니다.
- Lambda용 Rust 런타임 클라이언트: Rust 런타임 클라이언트는 실험용 패키지입니다. 주요 변경 사항이 있을 수 있으며 프로덕션에 권장되지 않습니다.
- Cargo Lambda: 이 라이브러리는 Rust로 빌드된 Lambda 함수와 함께 작동하는 명령줄 애플리케이션을 제공합니다.
- Lambda HTTP: 이 라이브러리는 HTTP 이벤트와 함께 작동하는 래퍼를 제공합니다.
- Lambda 확장: 이 라이브러리는 Rust로 Lambda 확장을 작성할 수 있도록 지원합니다.
- AWS Lambda 이벤트: 이 라이브러리는 일반적인 이벤트 소스 통합을 위한 유형 정의를 제공합니다.

Rust용 샘플 Lambda 애플리케이션

- 기본 Lambda 함수: 기본 이벤트를 처리하는 방법을 보여주는 Rust 함수입니다.
- 오류 처리가 포함된 Lambda 함수: Lambda에서 사용자 지정 Rust 오류를 처리하는 방법을 보여주는 Rust 함수입니다.
- 공유 리소스가 포함된 Lambda 함수: Lambda 함수를 생성하기 전에 공유 리소스를 초기화하는 Rust 프로젝트입니다.
- Lambda HTTP 이벤트: HTTP 이벤트를 처리하는 Rust 함수입니다.

- CORS 헤더가 포함된 Lambda HTTP 이벤트: Tower를 사용하여 CORS 헤더를 삽입하는 Rust 함수입니다.
- Lambda REST API: Axum과 Diesel을 사용하여 PostgreSQL 데이터베이스에 연결하는 REST API입니다.
- 서버리스 Rust 데모: Lambda의 Rust 라이브러리, 로깅, 환경 변수, AWS SDK의 사용법을 보여주는 Rust 프로젝트입니다.
- 기본 Lambda 확장: 기본 확장 이벤트를 처리하는 방법을 보여주는 Rust 확장입니다.
- Lambda 로그 Amazon Data Firehose 확장: Lambda 로그를 Firehose로 전송하는 방법을 보여주는 Rust 확장입니다.

주제

- Rust에서 Lambda 함수 핸들러 정의
- Rust의 Lambda 컨텍스트 객체
- Rust로 HTTP 이벤트 처리
- .zip 파일 아카이브를 사용하여 Rust Lambda 함수 배포
- Rust에서 Lambda 함수 로깅

Rust에서 Lambda 함수 핸들러 정의

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

Lambda 함수의 핸들러는 이벤트를 처리하는 함수 코드의 메서드입니다. 함수가 호출되면 Lambda는 핸들러 메서드를 실행합니다. 함수는 핸들러가 응답을 반환하거나 종료하거나 제한 시간이 초과될 때까지 실행됩니다.

Lambda 함수 코드를 Rust 실행 파일로 작성하세요. 핸들러 함수 코드와 main 함수를 구현하고 다음을 포함시키세요.

- Rust에 대한 Lambda 프로그래밍 모델을 구현하는 crates.io의 lambda_runtime 크레이트.
- 종속 구성 요소에 Tokio를 포함시키세요. Lambda용 Rust 런타임 클라이언트는 Tokio를 사용하여 비동기 호출을 처리합니다.

Example - JSON 이벤트를 처리하는 Rust 핸들러

다음 예제에서는 serde_json 크레이트를 사용하여 기본 JSON 이벤트를 처리합니다.

```
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
}
```

유의할 사항:

- use: Lambda 함수에 필요한 라이브러리를 가져옵니다.

- `async fn main`: Lambda 함수 코드를 실행하는 진입점입니다. Rust 런타임 클라이언트는 Tokio를 비동기 런타임으로 사용하므로 main 함수에 `#[tokio::main]`으로 주석을 달아야 합니다.
- `async fn handler(event: LambdaEvent<Value>) -> Result<Value,Error>`: Lambda 핸들러 서명입니다. 이 함수가 호출될 때 실행되는 코드가 여기에 포함됩니다.
  - `LambdaEvent<Value>`: Lambda 런타임과 Lambda 함수 컨텍스트에서 수신한 이벤트를 설명하는 일반 유형입니다.
  - `Result<Value, Error>`: 함수가 `Result` 유형을 반환합니다. 함수가 성공하면 결과는 JSON 값입니다. 함수가 실패하면 결과는 오류입니다.

공유 상태 사용

Lambda 함수의 핸들러 코드와 무관한 공유 변수를 선언할 수 있습니다. 이러한 변수는 함수가 이벤트를 수신하기 전에 초기화 단계에서 상태 정보를 로드하는 데 도움이 될 수 있습니다.

Example - 함수 인스턴스 간에 Amazon S3 클라이언트 공유

유의할 사항:

- `use aws_sdk_s3::Client`: 이 예제에서는 `aws-sdk-s3 = "0.26.0"`을 `Cargo.toml` 파일의 종속 구성 요소 목록에 추가해야 합니다.
- `aws_config::from_env`: 이 예제에서는 `aws-config = "0.55.1"`을 `Cargo.toml` 파일의 종속 구성 요소 목록에 추가해야 합니다.

```
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
}
```

Rust의 Lambda 컨텍스트 객체

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

Lambda는 함수를 실행할 때 핸들러가 수신하는 컨텍스트 객체에 컨텍스트 객체를 추가합니다 LambdaEvent . 이 객체는 호출, 함수 및 실행 환경에 관한 정보를 속성에 제공합니다.

컨텍스트 속성

- `request_id`: Lambda 서비스에서 생성한 AWS 요청 ID입니다.
- `deadline`: 현재 호출의 실행 기한(밀리초)입니다.
- `invoked_function_arn`: 호출 중인 Lambda 함수의 Amazon 리소스 이름(ARN)입니다.
- `xray_trace_id`: 현재 호출에 대한 AWS X-Ray 기록 ID입니다.
- `client_content`: AWS 모바일 SDK에서 전송한 클라이언트 컨텍스트 객체입니다. AWS 모바일 SDK를 사용하여 함수를 호출하지 않는 한 이 필드는 비어 있습니다.
- `identity`: 함수를 호출한 Amazon Cognito ID입니다. Lambda API에 대한 호출 요청이 Amazon Cognito 자격 증명 풀에서 발급한 AWS 자격 증명을 사용하여 이루어지지 않은 경우 이 필드는 비어 있습니다.
- `env_config`: 로컬 환경 변수의 Lambda 함수 구성입니다. 이 속성에는 함수 이름, 메모리 할당, 버전 및 로그 스트림과 같은 정보가 포함됩니다.

호출 컨텍스트 정보 액세스

Lambda 함수들은 자신의 환경과 호출 요청에 관한 메타데이터에 액세스할 수 있습니다. 함수 핸들러가 수신하는 `LambaEvent` 객체에는 `context` 메타데이터가 포함됩니다.

```
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
}
```

Rust로 HTTP 이벤트 처리

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

Amazon API Gateway APIs, Application Load Balancer 및 Lambda 함수 URL에서 Lambda에 HTTP 이벤트를 전송할 수 있습니다. crates.io의 aws_lambda_events 크레이트를 사용하여 이러한 소스의 이벤트를 처리할 수 있습니다.

Example - API Gateway 프록시 요청 처리

유의할 사항:

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: aws_lambda_events 크레이트에는 많은 Lambda 이벤트가 포함되어 있습니다. 컴파일 시간을 줄이려면 기능 플래그를 사용하여 필요한 이벤트를 활성화하세요. 예: `aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- `use http::HeaderMap`: 이 가져오기를 수행하려면 종속 구성 요소에 http 크레이트를 추가해야 합니다.

```
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
}
```

Lambda용 Rust 런타임 클라이언트는 이벤트를 전송하는 서비스에 관계없이 기본 HTTP 유형으로 작업할 수 있도록 하는 이러한 이벤트 유형에 대한 추상화도 제공합니다. 다음 코드는 이전 예제와 동일하며 Lambda 함수 URL, Application Load Balancer 및 API Gateway와 함께 즉시 작동합니다.

> **Note**
>
> lambda_http 크레이트는 아래에 있는 lambda_runtime 크레이트를 사용합니다. `lambda_runtime`을 별도로 가져올 필요가 없습니다.

Example - HTTP 요청 처리

```
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` 사용 방법의 또 다른 예는 AWS Labs GitHub 리포지토리의http-axum 코드 샘플을 참조하세요.

Rust용 샘플 HTTP Lambda 이벤트

- Lambda HTTP 이벤트: HTTP 이벤트를 처리하는 Rust 함수입니다.
- CORS 헤더가 포함된 Lambda HTTP 이벤트: Tower를 사용하여 CORS 헤더를 삽입하는 Rust 함수입니다.
- 공유 리소스가 포함된 Lambda HTTP 이벤트: 함수 핸들러가 생성되기 전에 초기화된 공유 리소스를 사용하는 Rust 함수입니다.

.zip 파일 아카이브를 사용하여 Rust Lambda 함수 배포

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

이 페이지에서는 Rust 함수를 컴파일한 다음 Cargo Lambda를 사용하여 컴파일된 바이너리를 AWS Lambda에 배포하는 방법을 설명합니다. 또한 AWS Command Line Interface 및 AWS Serverless Application Model CLI를 사용하여 컴파일된 바이너리를 배포하는 방법도 보여줍니다.

Sections

사전 조건

- Rust
- AWS Command Line Interface(AWS CLI) 버전 2

macOS, Windows 또는 Linux에서 Rust 함수 빌드

다음 단계에서는 Rust를 사용하여 첫 번째 Lambda 함수에 대한 프로젝트를 생성하고 Cargo Lambda로 컴파일하는 방법을 보여줍니다.

1. macOS, Windows 및 Linux에서 Lambda용 Rust 함수를 컴파일하는 Cargo 하위 명령인 Cargo Lambda를 설치합니다.

   Python 3이 설치된 시스템에 Cargo Lambda를 설치하려면 pip를 사용합니다.

   ```
   pip3 install cargo-lambda
   ```

   macOS 또는 Linux에 Cargo Lambda를 설치하려면 Homebrew를 사용합니다.

```
brew tap cargo-lambda/cargo-lambda
brew install cargo-lambda
```

Windows에 Cargo Lambda를 설치하려면 Scoop을 사용합니다.

```
scoop bucket add cargo-lambda
scoop install cargo-lambda/cargo-lambda
```

다른 옵션은 Cargo Lambda 설명서의 설치를 참조하세요.

2. 패키지 구조를 생성합니다. 이 명령은 `src/main.rs`에 몇 가지 기본 함수 코드를 생성합니다. 이 코드를 테스트에 사용하거나 사용자 고유의 코드로 바꿀 수 있습니다.

```
cargo lambda new my-function
```

3. 패키지의 루트 디렉터리 내에서 build 하위 명령을 실행하여 함수의 코드를 컴파일합니다.

```
cargo lambda build --release
```

(선택 사항) Lambda에서 AWS Graviton2를 사용하려면 `--arm64` 플래그를 추가하여 ARM CPU용 코드를 컴파일합니다.

```
cargo lambda build --release --arm64
```

4. Rust 함수를 배포하기 전에 머신에서 AWS 자격 증명을 구성합니다.

```
aws configure
```

Cargo Lambda로 Rust 함수 바이너리 배포

deploy 하위 명령을 사용하여 컴파일된 바이너리를 Lambda에 배포합니다. 이 명령은 실행 역할을 생성한 다음 Lambda 함수를 생성합니다. 기존 실행 역할을 지정하려면 --iam-role 플래그를 사용합니다.

```
cargo lambda deploy my-function
```

AWS CLI로 Rust 함수 바이너리 배포

AWS CLI로 바이너리를 배포할 수도 있습니다.

1. build 하위 명령을 사용하여 .zip 배포 패키지를 빌드합니다.

```
cargo lambda build --release --output-format zip
```

2. Lambda에 .zip 패키지를 배포합니다. `--role`로 실행 역할의 ARN을 지정합니다.

```
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 CLI로 Rust 함수 바이너리 배포

AWS SAM CLI로 바이너리를 배포할 수도 있습니다.

1. 리소스와 속성 정의를 사용하여 AWS SAM 템플릿을 생성합니다. 자세한 내용은 AWS Serverless Application Model 개발자 안내서의 AWS::Serverless::Function을 참조하세요.

   Example Rust 바이너리에 대한 SAM 리소스 및 속성 정의

```
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 하위 명령을 사용하여 함수를 컴파일합니다.

```
cargo lambda build --release
```

3. sam deploy 명령을 사용하여 함수를 Lambda에 배포합니다.

```
sam deploy --guided
```

AWS SAM CLI를 사용하여 Rust 함수를 빌드하는 방법에 대한 자세한 내용은 AWS Serverless Application Model 개발자 안내서의 Cargo Lambda로 Rust Lambda 함수 빌드를 참조하세요.

Cargo Lambda로 Rust 함수 호출

invoke 하위 명령을 사용하여 페이로드로 함수를 테스트합니다.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

AWS CLI로 Rust 함수 호출

AWS CLI를 사용하여 함수를 호출할 수도 있습니다.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

Rust에서 Lambda 함수 로깅

> **Note**
>
> Rust 런타임 클라이언트는 실험용 패키지입니다. 변경될 수 있으며 평가 목적으로만 사용됩니다.

AWS Lambda사용자를 대신하여 Lambda 함수를 자동으로 모니터링하고 Amazon에 로그를 전송합니다. CloudWatch Lambda 함수는 로그 로그 그룹과 함수의 각 인스턴스에 대한 로그 스트림과 함께 CloudWatch 제공됩니다. Lambda 런타임 환경은 각 호출에 관한 세부 정보를 로그 스트림에 전송하며, 함수 코드에서 로그 및 그 외 출력을 중계합니다. 자세한 설명은 AWS Lambda과 함께 Amazon CloudWatch Logs 사용 섹션을 참조하세요. 이 페이지에서는 Lambda 함수의 코드에서 로그 출력을 생성하는 방법을 설명합니다.

로그를 작성하는 함수 생성

함수 코드에서 로그를 출력하려면 `println!` 매크로와 같이 `stdout` 또는 `stderr`에 기록하는 모든 로깅 함수를 사용할 수 있습니다. 다음 예제에서는 `println!`을 사용하여 함수 처리기가 시작될 때와 완료되기 전에 메시지를 인쇄합니다.

```
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
}
```

Tracing 크레이트를 사용한 고급 로깅

Tracing은 구조화된 이벤트 기반 진단 정보를 수집하기 위해 Rust 프로그램을 계측하기 위한 프레임워크입니다. 이 프레임워크는 구조화된 JSON 로그 메시지 생성과 같은 로깅 출력 수준 및 형식을 사

용자 지정하는 유틸리티를 제공합니다. 이 프레임워크를 사용하려면 함수 핸들러를 구현하기 전에 subscriber를 초기화해야 합니다. 그런 다음 debug, info, error 등의 추적 매크로를 사용하여 각 시나리오에 대해 원하는 로깅 수준을 지정할 수 있습니다.

Example - Tracing 크레이트 사용

다음을 참고합니다.

- tracing_subscriber::fmt().json(): 이 옵션을 포함하면 로그 형식이 JSON으로 지정됩니다. 이 옵션을 사용하려면 tracing-subscriber 종속 구성 요소에 json 기능을 포함해야 합니다(예: tracing-subscriber = { version = "0.3.11", features = ["json"] }).
- #[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]: 이 주석은 핸들러가 호출될 때마다 스팬을 생성합니다. 스팬은 각 로그 라인에 요청 ID를 추가합니다.
- { %first_name }: 이 구성은 first_name 필드를 사용되는 로그 라인에 추가합니다. 이 필드의 값은 같은 이름의 변수에 해당합니다.

```
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
}
```

이 Rust 함수가 호출되면 다음과 유사한 2개의 로그 라인이 인쇄됩니다.

```
{"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"}]}
```

다른 AWS 서비스의 이벤트로 Lambda 간접 호출

일부 AWS 서비스는 트리거를 사용하여 직접 Lambda 함수를 간접적으로 호출할 수 있습니다. 이러한 서비스는 Lambda로 이벤트를 푸시하고, 지정된 이벤트가 발생하면 함수가 즉시 간접적으로 호출됩니다. 트리거는 개별 이벤트와 실시간 처리에 적합합니다. Lambda 콘솔을 사용하여 트리거를 생성하면 콘솔은 해당 AWS 서비스와 상호 작용하여 서비스에 대한 이벤트 알림을 구성합니다. 트리거는 실제로 Lambda가 아니라 이벤트를 생성하는 서비스에 의해 저장되고 관리됩니다.

이벤트는 JSON 형식으로 구조화된 데이터입니다. JSON 구조는 JSON 구조를 생성하는 서비스와 이벤트 유형에 따라 다르지만, 모두 함수가 이벤트를 처리하는 데 필요한 데이터를 포함합니다.

함수에는 여러 개의 트리거가 있을 수 있습니다. 각 트리거는 함수를 독립적으로 호출하는 클라이언트 역할을 하며 Lambda가 함수에 전달하는 각 이벤트에는 단일 트리거의 데이터만 있습니다. Lambda는 이벤트 문서를 객체로 변환한 후 함수 핸들러에 전달합니다.

서비스에 따라 이벤트 기반 간접 호출은 동기식 또는 비동기식으로 수행될 수 있습니다.

- 동기식 호출의 경우 이벤트를 생성하는 서비스는 함수로부터 응답을 기다립니다. 이 서비스는 함수가 응답에 반환되어야 하는 데이터를 정의합니다. 이 서비스는 오류에 대해 재시도할지 여부와 같은 오류 전략을 제어합니다.
- 비동기 호출의 경우 Lambda는 이벤트를 함수에 전달하기 전에 대기열에 추가합니다. Lambda가 이벤트를 대기열에 넣으면 이벤트를 생성한 서비스에 즉시 성공 응답을 보냅니다. 함수가 이벤트를 처리한 후 Lambda는 이벤트 생성 서비스에 대한 응답을 반환하지 않습니다.

트리거 생성

트리거를 생성하는 가장 쉬운 방법은 Lambda 콘솔을 사용하는 것입니다. 콘솔을 사용하여 트리거를 생성하면 Lambda는 함수의 리소스 기반 정책에 필요한 권한을 자동으로 추가합니다.

Lambda 콘솔을 사용하여 트리거를 생성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 트리거를 생성할 함수를 선택합니다.
3. 함수 개요 창에서 트리거 추가를 선택합니다.
4. 함수를 호출하려는 AWS 서비스를 선택합니다.
5. 트리거 구성 창의 옵션을 입력하고 추가를 선택합니다. 함수를 호출하기 위해 선택한 AWS 서비스에 따라 트리거 구성 옵션이 달라집니다.

Lambda 함수를 간접적으로 호출할 수 있는 서비스

다음 표에는 Lambda 함수를 간접적으로 호출할 수 있는 서비스가 나와 있습니다.

Service	호출 메서드
Amazon Alexa	이벤트 기반, 동기식 호출
Amazon Managed Streaming for Apache Kafka	이벤트 소스 매핑
자체 관리형 Apache Kafka	이벤트 소스 매핑
Amazon API Gateway	이벤트 기반, 동기식 호출
AWS CloudFormation	이벤트 기반, 비동기식 호출
Amazon CloudFront (Lambda@Edge)	이벤트 기반, 동기식 호출
Amazon CloudWatch Logs	이벤트 기반, 비동기식 호출
AWS CodeCommit	이벤트 기반, 비동기식 호출
AWS CodePipeline	이벤트 기반, 비동기식 호출
Amazon Cognito	이벤트 기반, 동기식 호출
AWS Config	이벤트 기반, 비동기식 호출
Amazon Connect	이벤트 기반, 동기식 호출
Amazon DynamoDB	이벤트 소스 매핑
Amazon Elastic File System	특수 통합
Elastic Load Balancing (Application Load Balancer)	이벤트 기반, 동기식 호출
AWS IoT	이벤트 기반, 비동기식 호출

Service	호출 메서드
Amazon Kinesis	이벤트 소스 매핑
Amazon Data Firehose	이벤트 기반, 동기식 호출
Amazon Lex	이벤트 기반, 동기식 호출
Amazon MQ	이벤트 소스 매핑
Amazon Simple Email Service	이벤트 기반, 비동기식 호출
Amazon Simple Notification Service	이벤트 기반, 비동기식 호출
Amazon Simple Queue Service	이벤트 소스 매핑
Amazon Simple Storage Service(S3)	이벤트 기반, 비동기식 호출
Amazon Simple Storage Service Batch	이벤트 기반, 동기식 호출
보안 관리자	이벤트 기반, 동기식 호출
Amazon VPC Lattice	이벤트 기반, 동기식 호출
AWS X-Ray	특수 통합

일반적인 Lambda 애플리케이션 유형 및 사용 사례

AWS Lambda에서 애플리케이션 빌드 시 코어 구성 요소는 Lambda 함수와 트리거입니다. 트리거는 함수를 호출하는 AWS 서비스 또는 애플리케이션이며, Lambda 함수는 이벤트를 처리하는 코드와 런타임입니다. 예시를 위해 다음 시나리오를 고려해 보세요.

- 파일 처리 – 사진 공유 애플리케이션을 가지고 있다고 가정해봅시다. 사람들이 이 애플리케이션을 사용하여 사진을 업로드하면 애플리케이션은 이러한 사용자 사진을 Amazon S3 버킷에 저장합니다. 그런 다음, 애플리케이션은 각 사용자 사진의 썸네일 버전을 만들어서 이를 사용자의 프로필 페이지에 표시합니다. 이러한 상황에서 자동으로 썸네일을 생성하는 Lambda 함수를 만드는 방법을 선택할 수 있습니다. Amazon S3는 지원되는 AWS 이벤트 소스 중 하나로, 객체에서 생성한 이벤트를 게시하고 Lambda 함수를 호출할 수 있습니다. Lambda 함수 코드는 S3 버킷에서 사진 객체를 읽어서 썸네일 버전을 만든 다음, 이를 또 다른 S3 버킷에 저장할 수 있습니다.
- 데이터 및 분석 – 분석 애플리케이션을 구축하여 DynamoDB 테이블의 원시 데이터를 저장하고 있다고 가정해봅시다. 업데이트를 기록하거나 테이블에서 항목을 삭제할 때 DynamoDB 스트림은 테이블에 연결된 스트림에 항목 업데이트 이벤트를 게시할 수 있습니다. 이 경우, 이벤트 데이터는 항목 키, 이벤트 이름(삽입, 업데이트, 삭제) 및 기타 관련 세부 정보를 제공합니다. 원시 데이터를 집계하여 사용자 지정 측정치를 생성하도록 Lambda 함수를 작성할 수 있습니다.
- 웹사이트 – 웹사이트를 생성하고 있으며 Lambda에 백엔드 로직을 호스팅하고 싶다고 가정해봅시다. HTTP 엔드포인트로 Amazon API Gateway를 사용하여 HTTP를 통해 Lambda 함수를 호출할 수 있습니다. 이제 웹 클라이언트가 API를 호출할 수 있으며, API Gateway는 Lambda에 요청을 라우팅할 수 있습니다.
- 모바일 애플리케이션 – 이벤트를 만드는 사용자 지정 모바일 애플리케이션을 가지고 있다고 가정해봅시다. Lambda 함수를 생성하여 사용자 지정 애플리케이션이 게시한 이벤트를 처리할 수 있습니다. 예를 들어 사용자 지정 모바일 애플리케이션에서 클릭을 처리하도록 Lambda 함수를 구성할 수 있습니다.

AWS Lambda는 이벤트 소스로 다수의 AWS 서비스를 지원합니다. 자세한 내용은 다른 AWS 서비스의 이벤트로 Lambda 간접 호출 섹션을 참조하세요. Lambda 함수를 트리거하도록 이들 이벤트 소스를 구성하면 이벤트 발생 시 Lambda 함수가 자동으로 호출됩니다. 이벤트 소스 매핑을 정의하여 추적할 이벤트와 호출할 Lambda 함수를 식별하는 방법을 정합니다.

다음은 이벤트 소스의 소개 예시와 end-to-end 경험의 작동 방식입니다.

예제 1: Amazon S3가 이벤트를 푸시하고 Lambda 함수를 호출

Amazon S3는 PUT, POST, COPY 및 DELETE 객체 이벤트 같이 상이한 유형의 이벤트를 버킷에 게시할 수 있습니다. 버킷 알림 기능을 사용하여 특정 유형의 이벤트가 발생할 때 Amazon S3에게 Lambda 함수를 호출하도록 지시하는 이벤트 소스 매핑을 구성할 수 있습니다.

다음은 일반적인 시퀀스입니다.

1. 사용자는 버킷에 객체를 생성합니다.
2. Amazon S3가 객체 생성 이벤트를 감지합니다.
3. Amazon S3는 실행 역할에서 제공한 권한을 사용하여 Lambda 함수를 호출합니다.
4. AWS Lambda는 Lambda 함수를 실행하여 이벤트를 파라미터로 지정합니다.

함수를 버킷 알림 작업으로 호출하도록 Amazon S3를 구성할 수 있습니다. 함수를 호출할 수 있는 권한을 Amazon S3에 부여하려면 함수의 리소스 기반 정책을 업데이트하세요.

예제 2: AWS Lambda가 Kinesis 스트림에서 이벤트를 가져와서 Lambda 함수 호출

폴 기반의 이벤트 소스에서 AWS Lambda는 소스를 폴링한 다음 레코드가 소스에서 감지될 때 Lambda 함수를 호출합니다.

- CreateEventSourceMapping
- UpdateEventSourceMapping

다음 단계는 사용자 지정 애플리케이션이 Kinesis 스트림에 레코드를 어떻게 기록하는지 보여줍니다.

1. 사용자 지정 애플리케이션은 Kinesis 스트림에 레코드를 기록합니다.
2. AWS Lambda는 스트림을 지속적으로 폴링하고 서비스가 스트림에서 새로운 레코드를 감지할 때 Lambda 함수를 호출합니다. AWS Lambda는 Lambda에서 생성하는 이벤트 소스 매핑에 따라 폴링할 대기열과 호출할 Lambda 함수를 식별합니다.
3. Lambda 함수는 수신 이벤트에서 호출됩니다.

스트림 기반 이벤트 소스를 사용할 경우AWS Lambda에 이벤트 소스 매핑을 만듭니다. Lambda는 스트림에서 항목을 읽고 함수를 동기식으로 호출합니다. 함수를 호출할 수 있는 권한을 Lambda에 부여할 필요는 없지만, 스트림에서 읽는 권한은 필요합니다.

Alexa에서 AWS Lambda 사용

Lambda 함수를 사용하여 Amazon Echo의 음성 도우미인 Alexa에게 새로운 기술을 제공하는 서비스를 구축할 수 있습니다. Alexa Skills Kit는 Lambda 함수로 실행되는 자체 서비스를 통해 이러한 새로운 기술을 개발할 수 있는 API, 도구 및 설명서를 제공합니다. Amazon Echo 사용자는 Alexa에게 질문을 하거나 요청을 하여 새로운 기술에 액세스할 수 있습니다.

Alexa 스킬 키트는 에서 사용할 수 있습니다. GitHub

- Java용 Alexa Skills Kit SDK
- Node.js용 Alexa Skills Kit SDK
- Alexa Skills Kit SDK for Python

Example Alexa smart home event

```
{
  "header": {
    "payloadVersion": "1",
    "namespace": "Control",
    "name": "SwitchOnOffRequest"
  },
  "payload": {
    "switchControlAction": "TURN_ON",
    "appliance": {
      "additionalApplianceDetails": {
        "key2": "value2",
        "key1": "value1"
      },
      "applianceId": "sampleId"
    },
    "accessToken": "sampleAccessToken"
  }
}
```

자세한 내용은 Build Skills with the Alexa Skills Kit 안내서의 AWS Lambda 함수로 사용자 지정 스킬 호스팅을 참조하세요.

Amazon API Gateway 엔드포인트를 사용하여 간접적으로 Lambda 함수 호출

Amazon API Gateway를 사용하여 웹API와 Lambda 함수의 HTTP 엔드포인트를 생성할 수 있습니다. API Gateway는 HTTP 요청을 Lambda 함수로 라우팅하는 웹 API를 생성하고 문서화하는 도구를 제공합니다. 인증 및 승인 제어를 통해 API에 대한 액세스를 보호할 수 있습니다. API는 인터넷을 통해 트래픽을 처리하거나 VPC 내에서만 액세스할 수 있습니다.

API의 리소스는 GET 또는 POST와 같은 하나 이상의 메서드를 정의합니다. 메서드에는 요청을 Lambda 함수 또는 다른 통합 유형으로 라우팅하는 통합이 있습니다. 각 리소스와 메서드를 개별적으로 정의하거나 특수 리소스 및 메서드 유형을 사용하여 패턴에 맞는 모든 요청을 일치시킬 수 있습니다. 프록시 리소스는 리소스 아래의 모든 경로를 포착합니다. ANY 메서드는 모든 HTTP 메서드를 포착합니다.

Sections

API 유형 선택

API Gateway는 Lambda 함수를 호출하는 세 가지 유형의 API를 지원합니다.

- HTTP API: 가볍고 대기 시간이 짧은 RESTful API입니다.
- REST API: 사용자 지정 가능하고 기능이 풍부한 RESTful API입니다.
- WebSocket API: 전이중 통신을 위해 클라이언트와의 지속적인 연결을 유지하는 웹 API입니다.

HTTP API와 REST API는 모두 HTTP 요청을 처리하고 응답을 반환하는 RESTful API입니다. HTTP API는 최신 버전이며 API Gateway 버전 2 API로 빌드되었습니다. HTTP API의 새로운 기능은 다음과 같습니다.

HTTP API 기능

- 자동 배포 – 경로 또는 통합을 수정하면 자동 배포가 활성화된 단계에 변경 내용이 자동으로 배포됩니다.
- 기본 단계 – API URL의 루트 경로에서 요청을 처리하는 기본 단계(`$default`)를 만들 수 있습니다. 명명된 단계의 경우 경로 시작 부분에 단계 이름을 포함해야 합니다.
- CORS 구성 – CORS 헤더를 함수 코드에 수동으로 추가하는 대신 발신 응답에 추가하도록 API를 구성할 수 있습니다.

REST API는 출시 이후 API Gateway가 지원한 클래식 RESTful API입니다. REST API에는 현재 더 많은 사용자 지정, 통합 및 관리 기능이 있습니다.

REST API 기능

- 통합 유형 – REST API는 사용자 지정 Lambda 통합을 지원합니다. 사용자 지정 통합을 사용하면 요청 본문만 함수로 전송하거나 변환 템플릿을 요청 본문에 적용한 후에 함수로 전송할 수 있습니다.
- 액세스 제어 – REST API는 인증 및 승인에 대한 더 많은 옵션을 지원합니다.
- 모니터링 및 추적 – REST API는 AWS X-Ray 추적 및 추가 로깅 옵션을 지원합니다.

자세한 비교는 API Gateway 개발자 안내서의 HTTP API와 REST API 중에서 선택을 참조하세요.

또한 WebSocket API는 API Gateway 버전 2 API를 사용하고 유사한 기능 세트를 지원합니다. 클라이언트와 API 간의 지속적인 연결을 활용하는 애플리케이션에 대해 WebSocket API를 사용합니다. WebSocket API는 전이중 통신을 제공하므로 클라이언트와 API 모두 응답을 기다리지 않고 지속적으로 메시지를 전송할 수 있습니다.

HTTP API는 간소화된 이벤트 형식(버전 2.0)을 지원합니다. 다음 예제에서는 HTTP API의 이벤트를 보여줍니다.

Example event-v2.json – API Gateway 프록시 이벤트(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
}
```

자세한 내용은 API Gateway 개발자 안내서의 AWS Lambda 통합을 참조하세요.

Lambda 함수에 엔드포인트 추가

Lambda 함수에 퍼블릭 엔드포인트를 추가하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 개요(Function overview)에서 트리거 추가(Add trigger)를 선택합니다.

4. API 게이트웨이(API Gateway)를 선택합니다.
5. API 생성(Create an API) 혹은 기존 API 사용(Use an existing API)을 선택합니다.
   a. 새 API:API 유형에서 HTTP API를 선택합니다. 자세한 내용은 API 유형을 참조하세요.
   b. 기존 API: 드롭다운 메뉴에서 API를 선택하거나 API ID(예: r3pmxmplak)를 입력합니다.
6. Security(보안)에서 Open(열기)을 선택합니다.
7. 추가를 선택합니다.

프록시 통합

API Gateway API는 스테이지, 리소스, 메서드 및 통합으로 구성됩니다. 스테이지와 리소스가 엔드포인트의 경로를 결정합니다.

API 경로 형식

- `/prod/` – `prod` 스테이지 및 루트 리소스입니다.
- `/prod/user` – `prod` 스테이지 및 `user` 리소스입니다.
- `/dev/{proxy+}` – `dev` 스테이지의 모든 경로입니다.
- `/` – (HTTP API) 기본 스테이지 및 루트 리소스입니다.

Lambda 통합은 경로와 HTTP 메서드 조합을 Lambda 함수에 매핑합니다. HTTP 요청의 본문을 있는 그대로 전달하거나(사용자 지정 통합) 헤더, 리소스, 경로 및 메서드를 비롯한 모든 요청 정보가 포함된 문서에 요청 본문을 캡슐화하도록 API Gateway를 구성할 수 있습니다.

자세한 내용은 API Gateway에서 Lambda 프록시 통합 설정을 참조하세요.

이벤트 형식

Amazon API Gateway는 HTTP 요청의 JSON 표현을 포함하는 이벤트와 동기적으로 함수를 호출합니다. 사용자 지정 통합의 경우 이벤트는 요청 본문입니다. 프록시 통합의 경우 이벤트의 구조가 정의되어 있습니다. 다음 예제에서는 API Gateway REST API의 프록시 이벤트를 보여줍니다.

Example event.json API Gateway 프록시 이벤트(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
  }
```

응답 형식

API Gateway는 함수의 응답을 기다렸다가 결과를 호출자에게 중계합니다. 사용자 지정 통합의 경우 함수의 출력을 HTTP 응답으로 변환하는 통합 응답 및 메서드 응답을 정의합니다. 프록시 통합의 경우 함수는 특정 형식의 응답 표현으로 응답해야 합니다.

다음 예제는 Node.js 함수의 응답 객체를 보여줍니다. 응답 객체는 JSON 문서를 포함하는 성공적인 HTTP 응답을 나타냅니다.

Example index.mjs – 프록시 통합 응답 객체(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 런타임은 응답 객체를 JSON으로 직렬화하여 API로 전송합니다. API는 응답을 구문 분석하고 이를 사용하여 HTTP 응답을 만든 다음 원래 요청을 한 클라이언트로 보냅니다.

Example HTTP 응답

```
< 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
  }
```

권한

Amazon API Gateway는 함수의 리소스 기반 정책에서 함수를 호출하는 권한을 가져옵니다. 전체 API에 호출 권한을 부여하거나 스테이지, 리소스 또는 메서드에 제한된 액세스 권한을 부여할 수 있습니다.

Lambda 콘솔, API Gateway 콘솔 또는 AWS SAM 템플릿을 사용하여 API를 함수에 추가할 때 함수의 리소스 기반 정책이 자동으로 업데이트됩니다. 다음은 함수 정책의 예입니다.

Example 함수 정책

```
{
  "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/"
        }
      }
    }
  ]
}
```

다음 API 작업을 사용하여 함수 정책 권한을 수동으로 관리할 수 있습니다.

- AddPermission
- RemovePermission
- GetPolicy

기존 API에 호출 권한을 부여하려면 `add-permission` 명령을 사용합니다.

```
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/"
```

다음 결과가 표시됩니다.

```
{
    "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**
>
> 함수와 API가 다른 AWS 리전에 있는 경우 소스 ARN의 리전 식별자는 API의 리전이 아닌 함수의 리전과 일치해야 합니다. API Gateway에서 함수를 간접적으로 호출하면 API의 ARN을 기반으로 하지만 함수의 리전과 일치하도록 수정된 리소스 ARN을 사용합니다.

이 예제의 소스 ARN은 ID `mnh1xmpli7`과 함께 API의 기본 단계에서 루트 리소스의 GET 메서드에 대한 통합 권한을 부여합니다. 소스 ARN에 별표를 사용하여 여러 단계, 메서드 또는 리소스에 권한을 부여할 수 있습니다.

리소스 패턴

- `mnh1xmpli7/*/GET/*` – 모든 단계의 모든 리소스에 대한 GET 메서드.
- `mnh1xmpli7/prod/ANY/user` – `prod` 스테이지의 `user` 리소스에 대한 ANY 메서드.
- `mnh1xmpli7/*/*/*` – 모든 단계의 모든 리소스에 대한 모든 메서드.

정책 보기 및 명령문 제거에 대한 자세한 내용은 리소스 기반 정책 정리 단원을 참조하세요.

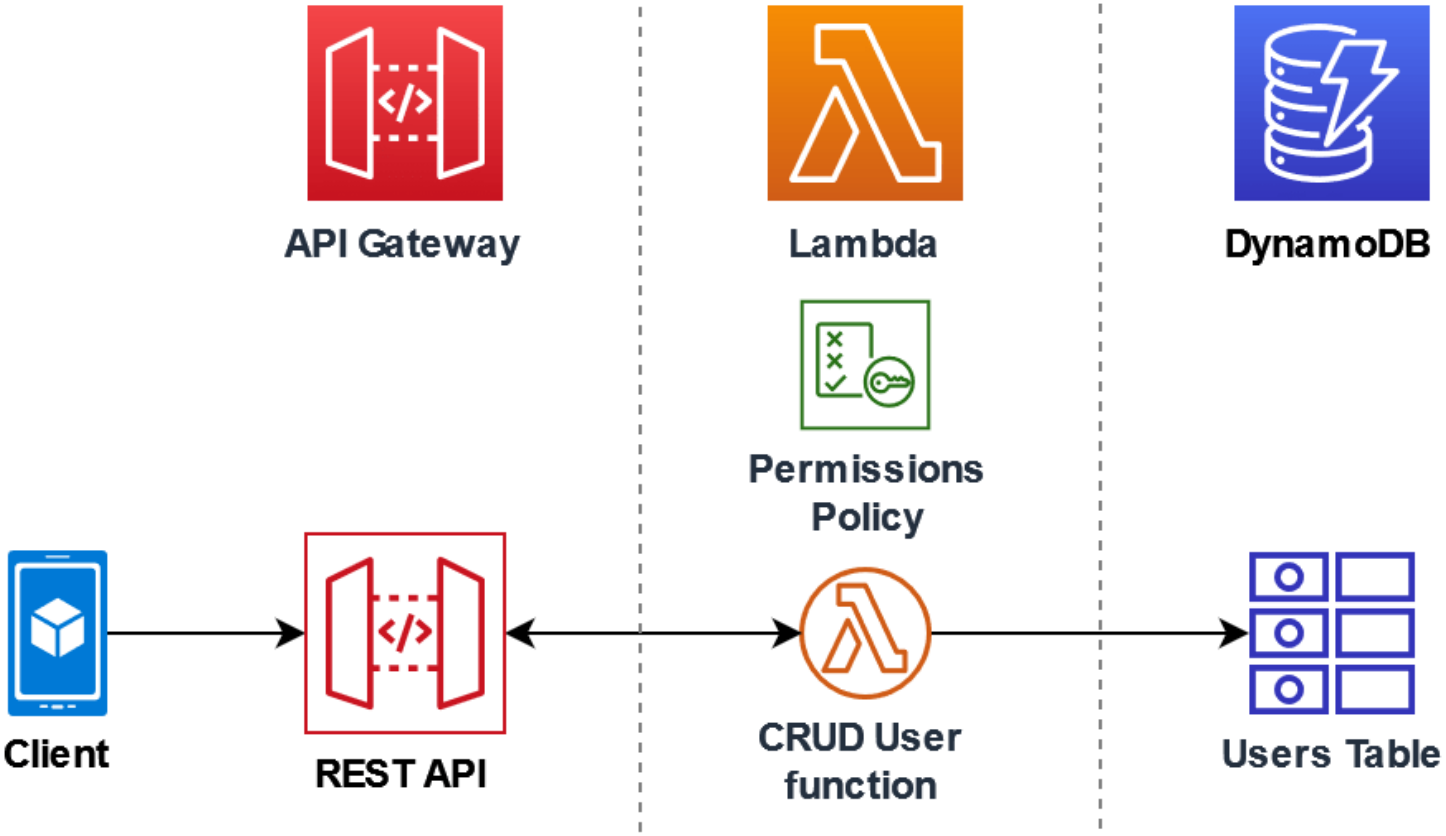
샘플 애플리케이션

Node.js를 사용하는 API Gateway 샘플 앱에는 AWS X-Ray 추적이 활성화된 REST API를 생성하는 AWS SAM 템플릿이 포함된 함수가 포함되어 있습니다. 여기에는 배포, 함수 간접 호출, API 테스트 및 정리를 위한 스크립트도 포함됩니다.

자습서: API Gateway에서 Lambda 사용

이 자습서에서는 HTTP 요청을 사용하여 Lambda 함수를 호출하는 REST API를 생성합니다. Lambda 함수는 DynamoDB 테이블에서 생성, 읽기, 업데이트 및 삭제(CRUD) 작업을 수행합니다. 이 함수는 여

기서 데모용으로 제공되지만, 모든 Lambda 함수를 호출할 수 있는 API Gateway REST API를 구성하는 방법을 배울 수 있습니다.



API Gateway를 사용하면 Lambda 함수를 호출할 수 있는 안전한 HTTP 엔드포인트가 사용자에게 제공되며, 트래픽을 제한하고 API 호출을 자동으로 검증 및 승인하여 함수에 대한 대량의 호출을 관리할 수 있습니다. 또한 API Gateway는 AWS Identity and Access Management(IAM) 및 Amazon Cognito를 사용하여 유연한 보안 제어 기능을 제공합니다. 이는 애플리케이션을 호출하기 위해 사전 승인이 필요한 사용 사례에 유용합니다.

이 자습서를 완료하는 과정에서 다음 단계를 거치게 됩니다.

1. Python 또는 Node.js 언어로 Lambda 함수를 생성하고 구성하여 DynamoDB 테이블에 대해 작업을 수행합니다.
2. API Gateway에서 REST API를 생성하여 Lambda 함수에 연결합니다.
3. DynamoDB 테이블을 정의한 후 콘솔에서 Lambda 함수를 사용하여 테스트합니다.
4. 터미널에서 curl을 사용하여 API를 배포하고 전체 설정을 테스트합니다.

이 단계를 완료하면 API Gateway를 사용하여 어떤 규모에서든 Lambda 함수를 안전하게 호출할 수 있는 HTTP 엔드포인트를 생성하는 방법을 배우게 됩니다. 또한 API를 배포하는 방법과 콘솔에서 API를 테스트하는 방법, 그리고 터미널에서 HTTP 요청을 보내 AI를 테스트하는 방법을 배우게 됩니다.

Sections

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한

을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

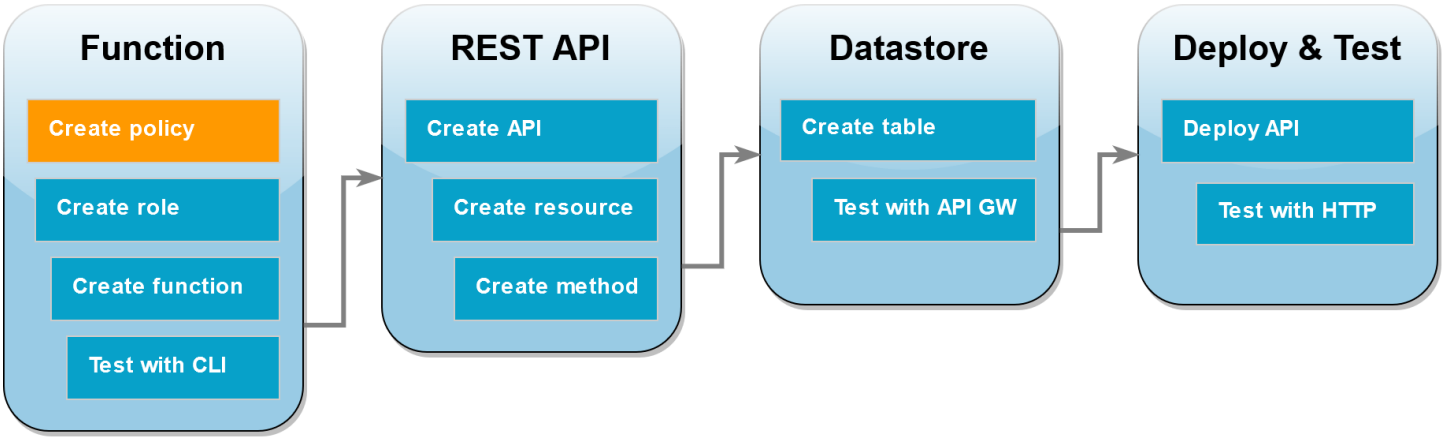
AWS Command Line Interface 설치

아직 AWS Command Line Interface를 설치하지 않은 경우 AWS CLI의 최신 버전 설치 또는 업데이트에서 설명하는 단계에 따라 설치하세요.

이 자습서에서는 명령을 실행할 셸 또는 명령줄 터미널이 필요합니다. Linux 및 macOS에서는 선호하는 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다.

권한 정책 생성



Lambda 함수의 실행 역할을 생성하려면 먼저 권한 정책을 생성하여 필요한 AWS 리소스에 액세스할 권한을 함수에 부여해야 합니다. 이 자습서에서는 이 정책을 통해 Lambda가 DynamoDB 테이블에서 CRUD 작업을 수행하고 Amazon CloudWatch Logs 로그에 쓸 수 있도록 합니다.

정책 생성

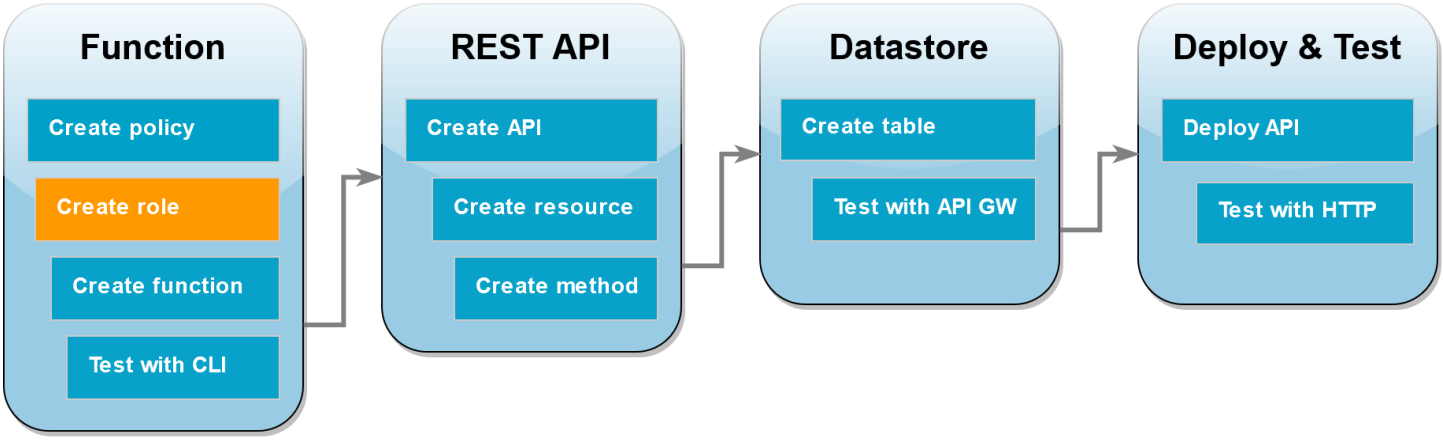
1. IAM 콘솔에서 정책 페이지를 엽니다.
2. 정책 생성(Create Policy)을 선택합니다.
3. JSON 탭에서 다음과 같은 사용자 지정 정책을 JSON 편집기에 붙여 넣습니다.

```
{
  "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. 다음: 태그를 선택합니다.
5. 다음: 검토(Next: Review)를 선택합니다.

6. 정책 검토의 이름에 **lambda-apigateway-policy**를 입력합니다.
7. 정책 생성을 선택합니다.

실행 역할 만들기



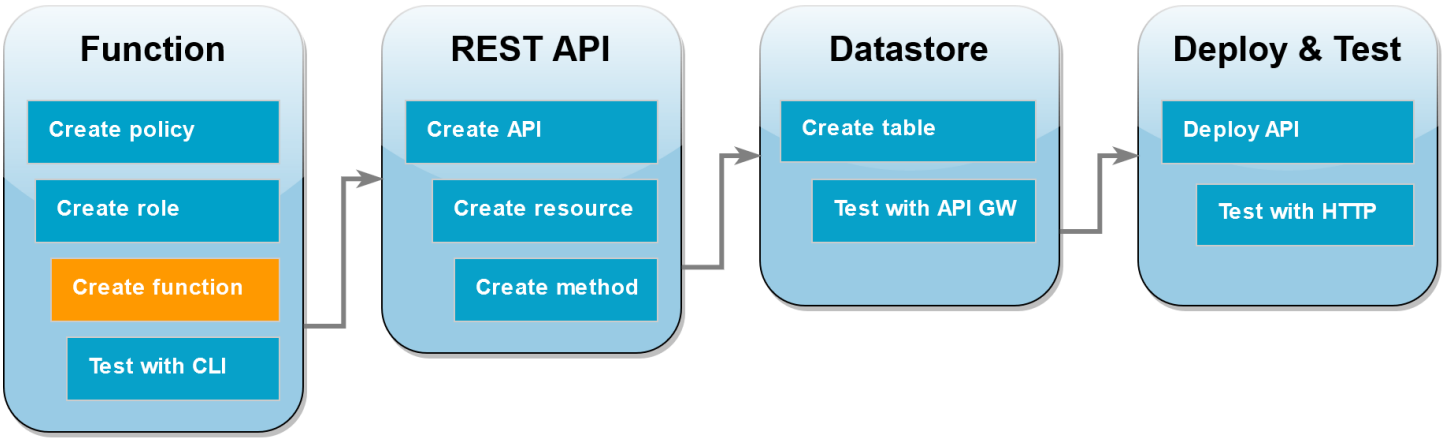
실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 AWS Identity and Access Management(IAM) 역할입니다. 함수가 DynamoDB 테이블에 대해 작업을 수행할 수 있도록 하려면 이전 단계에서 생성한 권한 정책을 연결합니다.

실행 역할을 생성하고 사용자 지정 권한 정책을 연결하려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티의 유형으로 AWS 서비스를 선택한 다음 사용 사례로 Lambda를 선택합니다.
4. 다음을 선택합니다.
5. 정책 검색 상자에 **lambda-apigateway-policy**를 입력합니다.
6. 검색 결과에서 생성한 정책(`lambda-apigateway-policy`)을 선택한 후, 다음(Next)을 선택합니다.
7. Role details(역할 세부 정보)에서 Role name(역할 이름)에 **lambda-apigateway-role**을 입력한 다음 Create role(역할 생성)을 선택합니다.

자습서 뒷부분에서는 방금 생성한 역할의 Amazon 리소스 이름(ARN)이 필요합니다. IAM 콘솔의 Roles(역할) 페이지에서 역할의 이름(`lambda-apigateway-role`)을 선택하고 Summary(요약) 페이지에 표시된 Role ARN(역할 ARN)을 복사합니다.

함수 생성



다음 코드 예제는 API Gateway로부터 생성하려는 DynamoDB 테이블과 일부 페이로드 데이터에 대해 수행할 작업을 지정하는 이벤트 입력을 수신합니다. 수신된 파라미터가 유효하면 함수가 테이블에 대해 요청된 작업을 수행합니다.

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**
>
> 이 예제에서는 DynamoDB 테이블의 이름이 함수 코드에 변수로 정의됩니다. 실제 애플리케이션에서는 이 파라미터를 환경 변수로 전달하고 테이블 이름을 하드 코딩하지 않는 것이 좋습니다. 자세한 내용은 AWS Lambda 환경 변수 사용 섹션을 참조하세요.

함수를 만들려면

1. 코드 예제를 이름이 `index.mjs`인 파일로 저장하고 필요한 경우 코드에 지정된 AWS 리전을 편집합니다. 코드에 지정된 리전은 자습서 뒷부분에서 DynamoDB 테이블을 생성하는 리전과 동일해야 합니다.

2. 다음 `zip` 명령을 사용하여 배포 패키지를 생성합니다.

```
zip function.zip index.mjs
```

3. `create-function` AWS CLI 명령을 사용하여 Lambda 함수를 생성합니다. `role` 파라미터에 이전에 복사한 실행 역할의 Amazon 리소스 이름(ARN)을 입력합니다.

```
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**
>
> 이 예제에서는 DynamoDB 테이블의 이름이 함수 코드에 변수로 정의됩니다. 실제 애플리케이션에서는 이 파라미터를 환경 변수로 전달하고 테이블 이름을 하드 코딩하지 않는 것이 좋습니다. 자세한 내용은 AWS Lambda 환경 변수 사용 섹션을 참조하세요.

함수를 만들려면

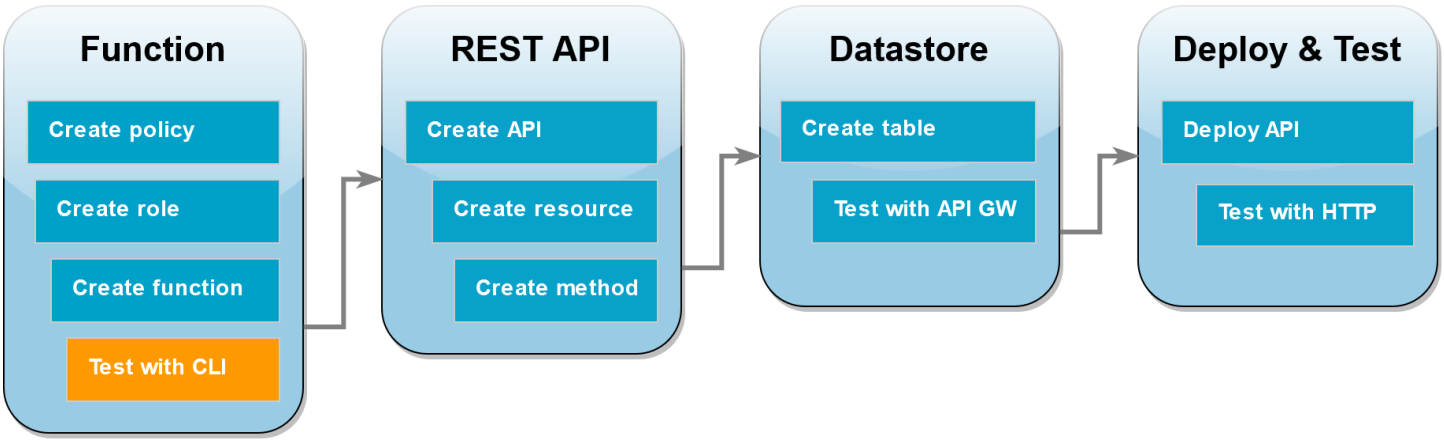
1. 예제 코드를 `LambdaFunctionOverHttps.py`라는 파일로 저장합니다.
2. 다음 `zip` 명령을 사용하여 배포 패키지를 생성합니다.

```
zip function.zip LambdaFunctionOverHttps.py
```

3. `create-function` AWS CLI 명령을 사용하여 Lambda 함수를 생성합니다. `role` 파라미터에 이전에 복사한 실행 역할의 Amazon 리소스 이름(ARN)을 입력합니다.

```
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 CLI를 사용하여 함수 호출



함수를 API Gateway와 통합하기 전에 함수를 성공적으로 배포했는지 확인합니다. API Gateway API가 Lambda로 전송할 파라미터를 포함하는 테스트 이벤트를 생성하고 AWS CLI `invoke` 명령을 사용하여 함수를 실행합니다.

AWS CLI를 사용하여 Lambda 함수를 호출하려면

1. 다음 JSON을 `input.txt`라는 파일로 저장합니다.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

2. 다음 `invoke` AWS CLI 명령을 실행합니다.

```
aws lambda invoke \
--function-name LambdaFunctionOverHttps \
--payload file://input.txt outputfile.txt \
--cli-binary-format raw-in-base64-out
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

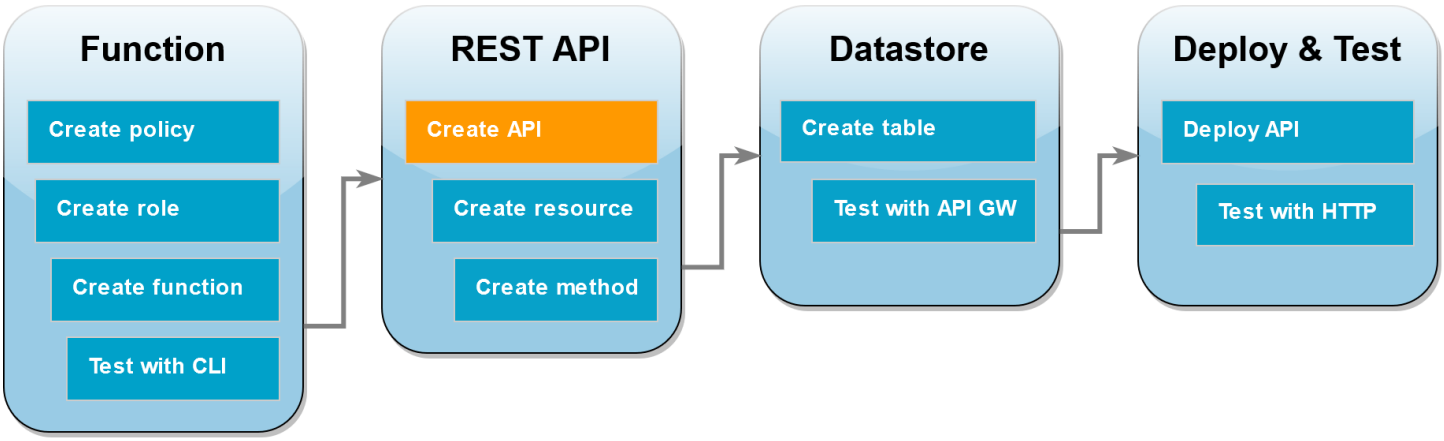
다음과 같은 응답이 표시되어야 합니다.

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

3. 함수가 JSON 테스트 이벤트에 지정된 echo 작업을 수행했는지 확인합니다. `outputfile.txt` 파일을 검사하여 다음 항목이 포함되어 있는지 확인합니다.

```
{"somekey1": "somevalue1", "somekey2": "somevalue2"}
```

API Gateway를 사용하여 REST API 생성

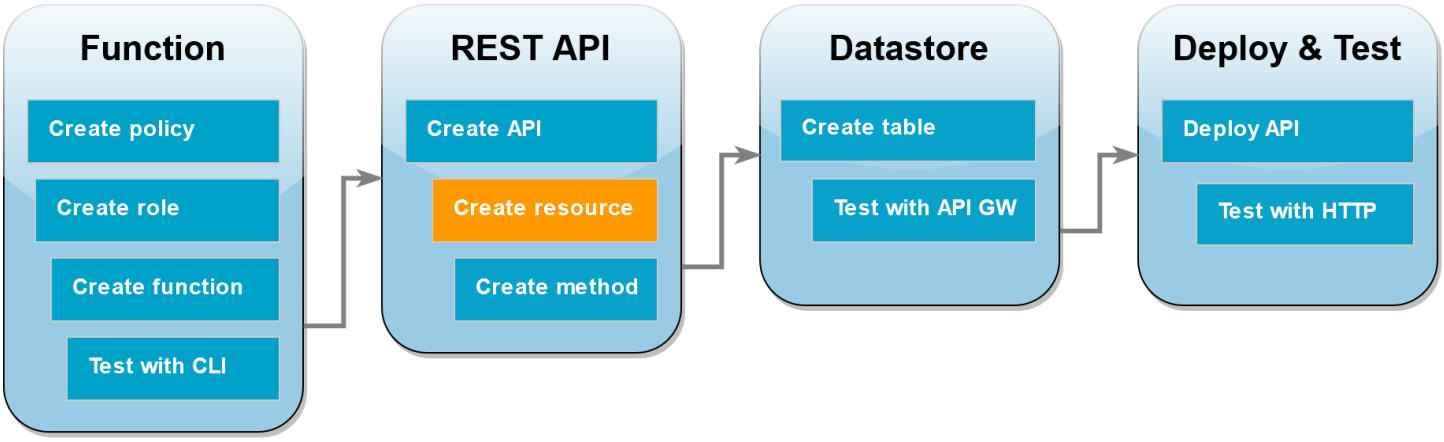


이 단계에서는 Lambda 함수를 호출하는 데 사용할 API Gateway REST API를 생성합니다.

API를 생성하려면

1. API Gateway 콘솔을 엽니다.
2. API 생성(Create API)을 선택합니다.
3. REST API 상자에서 빌드를 선택합니다.
4. API 세부 정보에서 새 API를 선택한 상태로 두고 API 이름에 **DynamoDBOperations**를 입력합니다.
5. API 생성(Create API)을 선택합니다.

REST API에서 리소스 생성

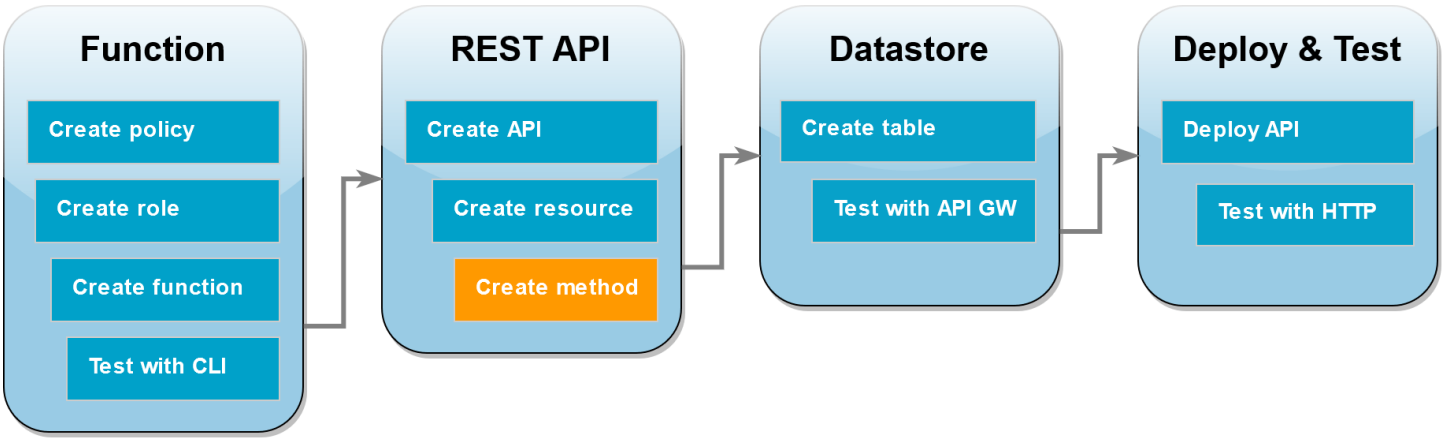


API에 HTTP 메서드를 추가하려면 먼저 해당 메서드가 작동하는 데 필요한 리소스를 생성해야 합니다. 여기서는 DynamoDB 테이블을 관리하기 위한 리소스를 생성합니다.

리소스를 생성하려면

1. API Gateway 콘솔의 사용하는 API에 대한 리소스 페이지에서 리소스 생성을 선택합니다.
2. 리소스 세부 정보에서 리소스 이름에 **DynamoDBManager**를 입력합니다.
3. 리소스 생성을 선택합니다.

HTTP POST 메서드 생성



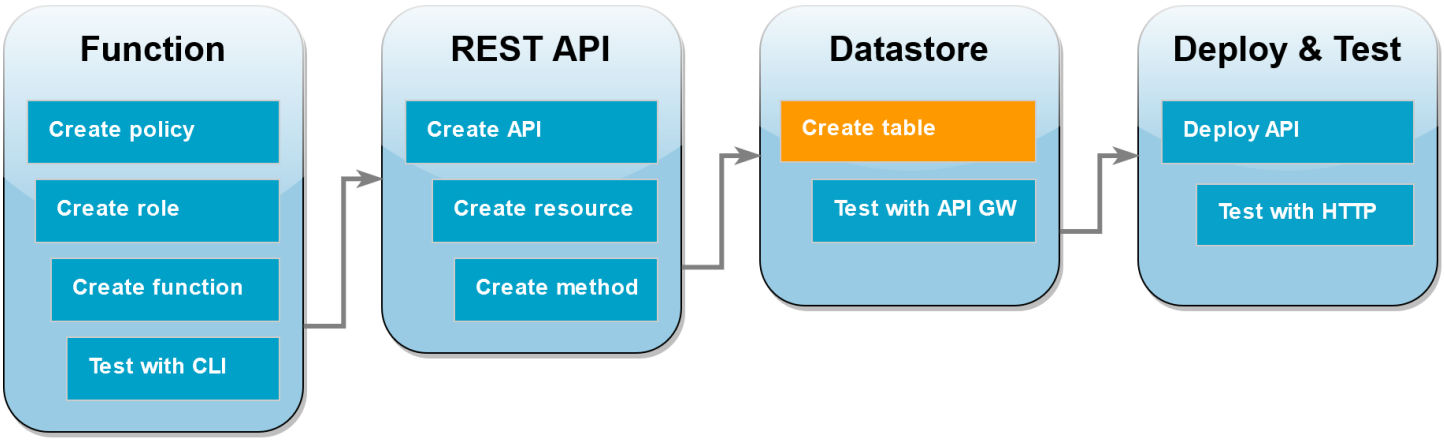
이 단계에서는 DynamoDBManager 리소스의 메서드(POST)를 생성합니다. 이 POST 메서드를 Lambda 함수에 연결하여 이 메서드가 HTTP 요청을 수신하면 API Gateway가 Lambda 함수를 호출하도록 합니다.

> **Note**
>
> 이 자습서에서는 HTTP 메서드(POST)를 하나 사용하여 DynamoDB 테이블에 대해 모든 작업을 수행하는 단일 Lambda 함수를 호출합니다. 실제 애플리케이션에서는 각 작업마다 서로 다른 Lambda 함수와 HTTP 메서드를 사용하는 것이 좋습니다. 자세한 내용은 Serverless Land의 The Lambda monolith를 참조하세요.

POST 메서드를 생성하려면

1. API의 리소스 페이지에서 /DynamoDBManager 리소스가 강조 표시되어 있는지 확인합니다. 그런 다음 메서드 창에서 메서드 생성을 선택합니다.
2. 메서드 유형에서 POST를 선택합니다.
3. 통합 유형에서 Lambda 함수를 선택한 상태로 둡니다.
4. Lambda 함수에서 함수의 Amazon 리소스 이름(ARN)(LambdaFunctionOverHttps)을 선택합니다.
5. 메서드 생성을 선택합니다.

DynamoDB 테이블 생성

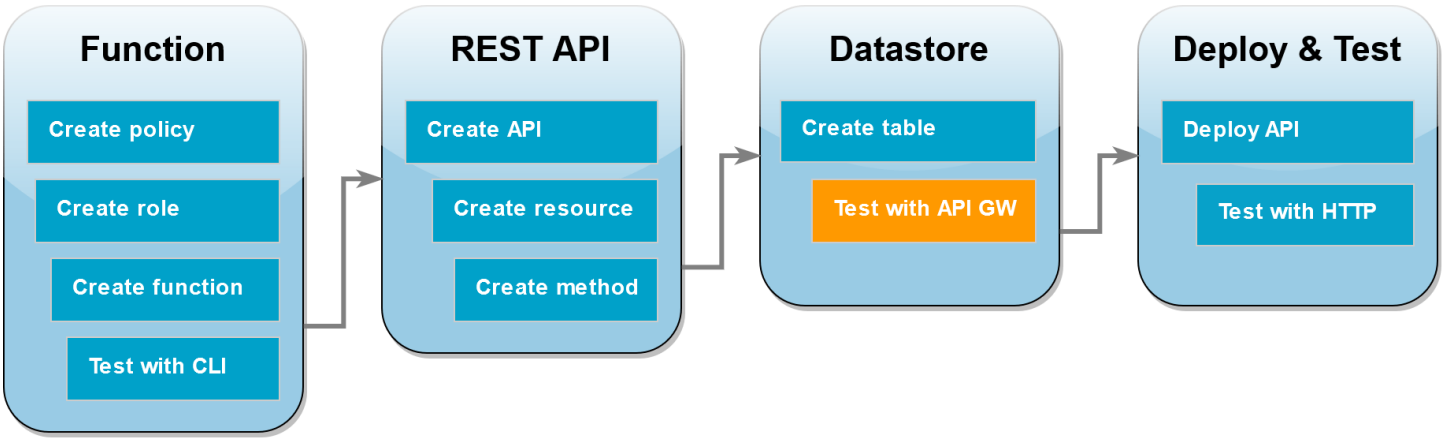


Lambda 함수가 CRUD 작업을 수행할 빈 DynamoDB 테이블을 생성합니다.

DynamoDB 테이블을 생성하려면

1. DynamoDB 콘솔의 테이블 페이지를 엽니다.
2. 테이블 생성을 선택합니다.
3. 테이블 세부 정보에서 다음을 수행합니다.
    1. 테이블 이름에 **`lambda-apigateway`**을(를) 입력합니다.
    2. 파티션 키에서 **`id`**를 입력하고 데이터 유형을 문자열로 설정합니다.
4. Table settings(테이블 설정)에서 Default settings(기본 설정)을 그대로 둡니다.
5. 테이블 생성을 선택합니다.

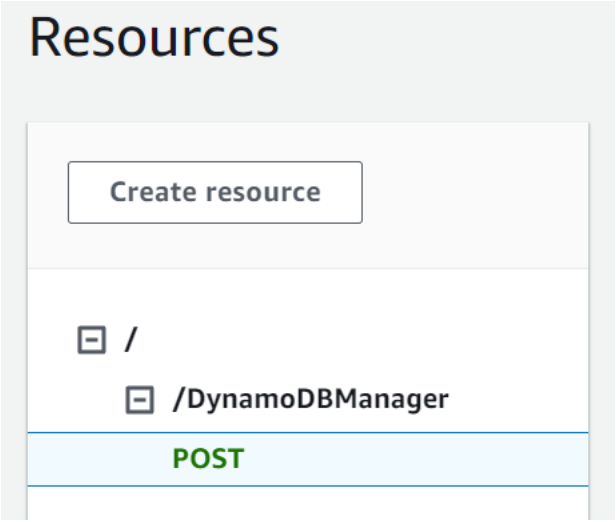
API Gateway, Lambda 및 DynamoDB의 통합 테스트



이제 API Gateway API 메서드와 Lambda 함수 및 DynamoDB 테이블의 통합을 테스트할 준비가 되었습니다. API Gateway 콘솔을 사용하여 콘솔의 테스트 함수를 통해 POST 메서드에 직접 요청을 보냅니다. 이 단계에서는 먼저 `create` 작업을 사용하여 DynamoDB 테이블에 새 항목을 추가한 다음 `update` 작업을 사용하여 해당 항목을 수정합니다.

테스트 1: DynamoDB 테이블에 새 항목을 생성하려면

1. API Gateway 콘솔에서 API(`DynamoDBOperations`)를 선택합니다.
2. `DynamoDBManager` 리소스에서 POST 메서드를 선택합니다.



3. 테스트 탭을 선택합니다. 탭을 표시하려면 오른쪽 화살표 버튼을 선택해야 할 수도 있습니다.
4. 메서드 테스트에서 쿼리 문자열과 헤더는 비워 둡니다. 요청 본문에 다음 JSON을 붙여넣습니다.

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. 테스트를 선택합니다.

   테스트가 완료될 때 표시되는 결과에 `200` 상태가 표시되어야 합니다. 이 상태 코드는 `create` 작업이 성공했음을 나타냅니다.

   이를 확인하려면 DynamoDB 테이블에 새 항목이 포함되어 있는지 확인합니다.

6. DynamoDB 콘솔의 Tables(테이블) 페이지를 열고 `lambda-apigateway` 테이블을 선택합니다.
7. Explore table items(테이블 항목 탐색)를 선택합니다. Items returned(반환된 항목) 창에 ID가 `1234ABCD`이고 number(번호)가 `5`인 항목이 하나 표시되어야 합니다.

테스트 2: DynamoDB 테이블의 항목을 업데이트하려면

1. API Gateway 콘솔에서 POST 메서드의 테스트 탭으로 돌아갑니다.
2. 메서드 테스트에서 쿼리 문자열과 헤더는 비워 둡니다. 요청 본문에 다음 JSON을 붙여넣습니다.

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "AttributeUpdates": {
            "number": {
                "Value": 10
            }
        }
    }
}
```

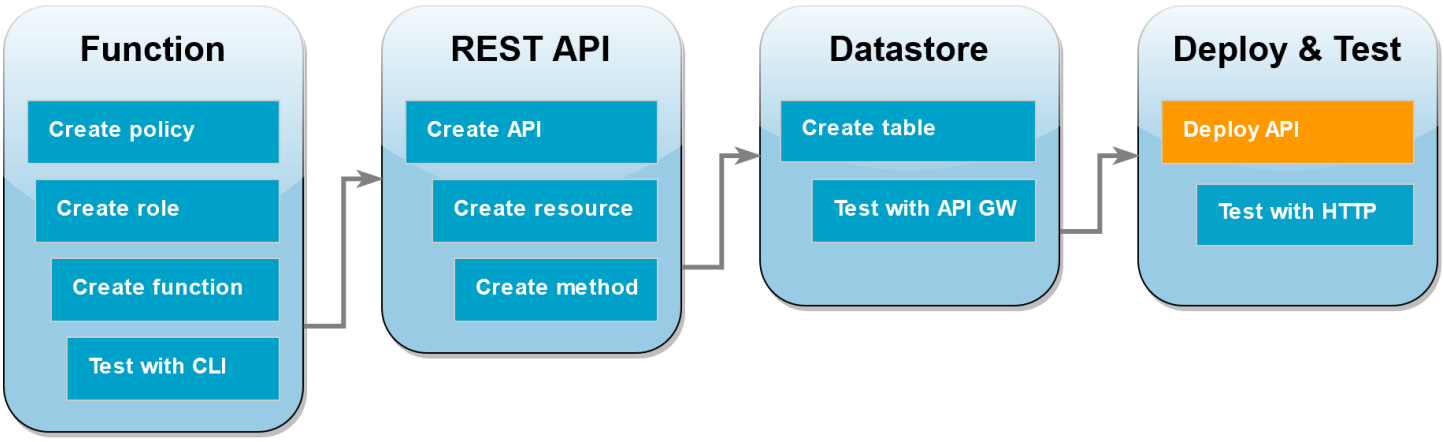
3. 테스트를 선택합니다.

   테스트가 완료될 때 표시되는 결과에 `200` 상태가 표시되어야 합니다. 이 상태 코드는 `update` 작업이 성공했음을 나타냅니다.

   이를 확인하려면 DynamoDB 테이블의 항목이 수정되었는지 확인합니다.

4. DynamoDB 콘솔의 Tables(테이블) 페이지를 열고 `lambda-apigateway` 테이블을 선택합니다.
5. Explore table items(테이블 항목 탐색)를 선택합니다. Items returned(반환된 항목) 창에 ID가 `1234ABCD`이고 number(번호)가 `10`인 항목이 하나 표시되어야 합니다.

API 배포

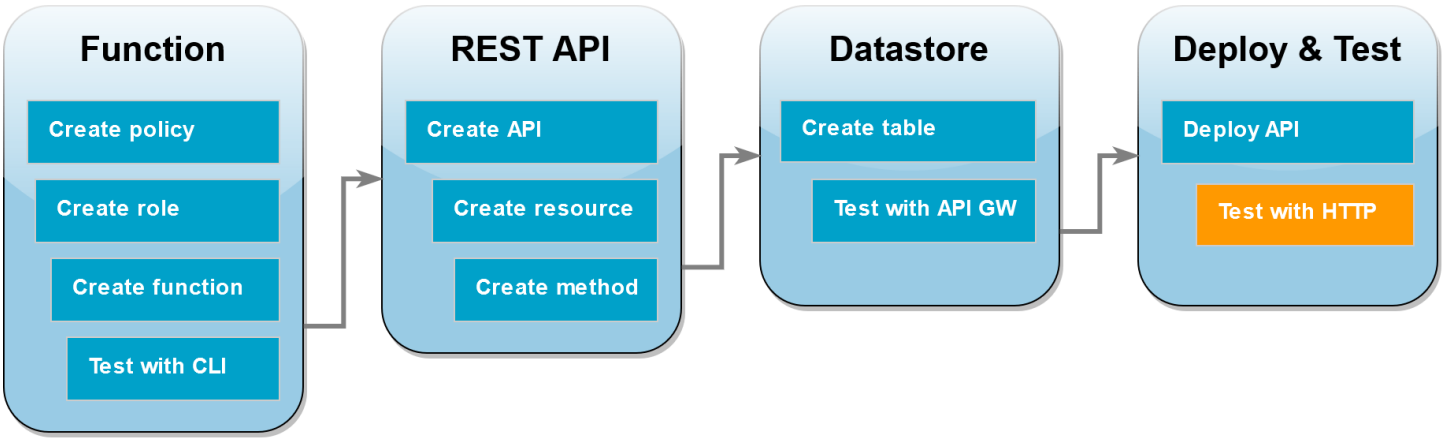


클라이언트가 API를 호출하기 위해서는 배포 및 연결된 단계를 생성해야 합니다. 단계는 메서드와 통합을 포함한 API의 스냅샷을 나타냅니다.

API를 배포하려면

1. API Gateway 콘솔의 APIs(API) 페이지를 열고 `DynamoDBOperations` API를 선택합니다.
2. API의 리소스 페이지에서 API 배포를 선택합니다.
3. 단계에서 *새 단계*를 선택하고 단계 이름에 **`test`**를 입력합니다.
4. [배포]를 선택합니다.
5. 단계 세부 정보 창에서 호출 URL을 복사합니다. 다음 단계에서는 이 URL을 사용하여 HTTP 요청을 통해 함수를 호출합니다.

curl을 사용하여 HTTP 요청을 통해 함수 호출



이제 API에 HTTP 요청을 전송하여 Lambda 함수를 호출할 수 있습니다. 이 단계에서는 DynamoDB 테이블에 새 항목을 생성한 다음 해당 항목을 삭제합니다.

curl을 사용하여 Lambda 함수를 호출하려면

1. 이전 단계에서 복사한 호출 URL을 사용하여 다음 `curl` 명령을 실행합니다. curl을 `-d`(데이터) 옵션과 함께 사용하면 자동으로 HTTP POST 메서드가 사용됩니다.

```
curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number": 15}}}'
```

2. 생성 작업이 성공했는지 확인하려면 다음을 수행합니다.
   1. DynamoDB 콘솔의 Tables(테이블) 페이지를 열고 `lambda-apigateway` 테이블을 선택합니다.
   2. Explore table items(테이블 항목 탐색)를 선택합니다. Items returned(반환된 항목) 창에 ID가 `5678EFGH`이고 번호가 `15`인 항목이 표시되어야 합니다.
3. 다음 `curl` 명령을 실행하여 방금 생성한 항목을 삭제합니다. 실제 호출 URL을 사용하세요.

```
curl https://l8togsqxd8.execute-api.us-west-2.amazonaws.com/test/DynamoDBManager \
-d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
```

4. 삭제 작업이 성공했는지 확인합니다. DynamoDB 콘솔 Explore items(항목 탐색) 페이지의 Items returned(반환된 항목) 창에서 ID가 `5678EFGH`인 항목이 더 이상 테이블에 없는지 확인합니다.

리소스 정리(선택 사항)

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.

3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

API를 삭제하려면

1. API Gateway 콘솔의 API 페이지를 엽니다.
2. 생성한 API를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 삭제를 선택합니다.

DynamoDB 테이블을 삭제하려면

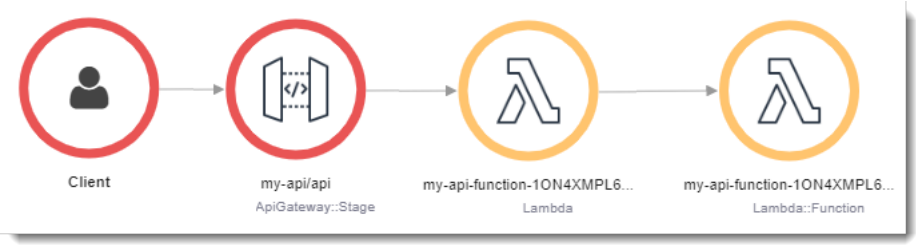
1. DynamoDB 콘솔의 테이블 페이지를 엽니다.
2. 생성한 테이블을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 상자에 **`delete`**를 입력합니다.
5. 테이블 삭제를 선택합니다.

API Gateway API를 사용한 Lambda 오류 처리

API Gateway는 모든 호출 및 함수 오류를 내부 오류로 처리합니다. Lambda API가 호출 요청을 거부하면 API Gateway는 500 오류 코드를 반환합니다. 함수가 실행되지만 오류를 반환하거나 잘못된 형식으로 응답을 반환하는 경우 API Gateway에서 502를 반환합니다. 두 경우 모두 API Gateway의 응답 본문은 `{"message": "Internal server error"}`입니다.

> **Note**
>
> API Gateway는 Lambda 호출을 다시 시도하지 않습니다. Lambda가 오류를 반환하면 API Gateway는 클라이언트에 오류 응답을 반환합니다.

다음 예제에서는 함수 오류가 발생해 API Gateway에서 502가 반환된 요청에 대한 X-Ray 추적 맵을 보여 줍니다. 클라이언트는 제네릭 오류 메시지를 수신합니다.

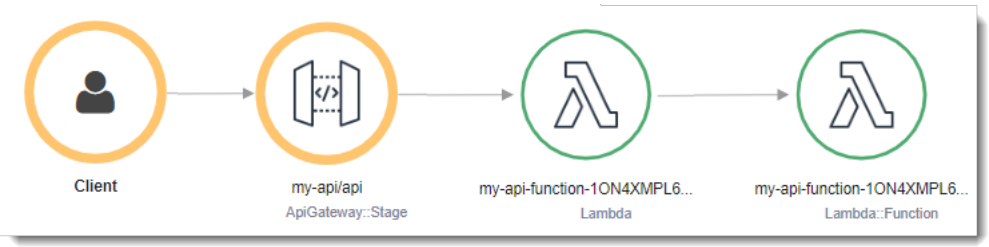


오류 응답을 사용자 지정하려면 코드에서 오류를 포착하고 필요한 형식으로 응답을 지정해야 합니다.

Example index.mjs – 오류 형식 지정

```
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 Gateway는 이 응답을 사용자 지정 상태 코드 및 본문을 사용한 HTTP 오류로 변환합니다. 오류를 처리했기 때문에 추적 맵에서 함수 노드는 녹색입니다.



AWS Lambda와 함께 AWS Application Composer 사용

AWS Application Composer은 AWS에 최신 애플리케이션을 디자인하기 위한 비주얼 빌더입니다. 시각적 캔버스에서 AWS 서비스를 드래그, 그룹화 및 연결하여 애플리케이션 아키텍처를 설계합니다. Application Composer는 설계에서 AWS SAM 또는 AWS CloudFormation을 사용하여 배포할 수 있는 코드형 인프라 (IaC) 템플릿을 생성합니다.

Lambda 함수를 Application Composer로 내보내기

Lambda 콘솔을 사용하여 기존 Lambda 함수의 구성을 기반으로 새 프로젝트를 생성하여 Application Composer 사용을 시작할 수 있습니다. 함수의 구성 및 코드를 Application Composer로 내보내기하여 새 프로젝트를 생성하려면 다음을 수행하세요.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. Application Composer 프로젝트의 기준으로 사용하려는 함수를 선택합니다.
3. 함수 개요 창에서 Application Composer로 내보내기를 선택합니다.

   함수의 구성 및 코드를 Application Composer로 내보내기 위해 Lambda는 계정에 Amazon S3 버킷을 생성하여 이 데이터를 임시로 저장합니다.
4. 대화 상자에서 확인 및 프로젝트 생성을 선택하여 이 버킷의 기본 이름을 수락하고 함수의 구성 및 코드를 Application Composer로 내보내기합니다.
5. (선택 사항) Lambda가 생성하는 Amazon S3 버킷의 다른 이름을 선택하려면 새 이름을 입력하고 확인 및 프로젝트 생성을 선택합니다. Amazon S3 버킷에 이름은 전역적으로 고유해야 하며 버킷 이름 지정 규칙을 따라야 합니다.
6. 프로젝트 및 함수 파일을 Application Composer에 저장하려면 로컬 동기화 모드를 활성화합니다.

> **Note**
>
> 이전에 Application Composer로 내보내기 기능을 사용하고 기본 이름을 사용하여 Amazon S3 버킷을 생성한 경우, Lambda는 이 버킷이 아직 존재한다면 재사용할 수 있습니다. 대화 상자에서 기본 버킷 이름을 수락하여 기존 버킷을 재사용합니다.

Amazon S3 버킷 구성 전송

Lambda가 함수 구성을 전송하기 위해 생성하는 Amazon S3 버킷은 AES 256 암호화 표준을 사용하여 객체를 자동으로 암호화합니다. Lambda는 또한 버킷 소유자 조건을 사용하도록 버킷을 구성하여 AWS 계정만 버킷에 객체를 추가할 수 있도록 합니다.

Lambda는 업로드 후 10일이 지나면 객체를 자동으로 삭제하도록 버킷을 구성합니다. 하지만 Lambda는 버킷 자체를 자동으로 삭제하지 않습니다. AWS 계정에서 버킷을 삭제하려면 버킷 삭제의 지침을 따릅니다. 기본 버킷 이름은 10자리 영숫자 문자열인 접두사 `lambdasam`을 사용하며, 함수를 생성한 AWS 리전의 위치는 다음과 같습니다.

```
lambdasam-06f22da95b-us-east-1
```

AWS 계정에 추가되어 추가 요금이 부과되지 않도록 하려면 Application Composer로 함수 내보내기를 완료하는 즉시 Amazon S3 버킷을 삭제하는 것이 좋습니다.

표준 Amazon S3 요금이 적용됩니다.

필요한 권한

Application Composer 기능을 포함하는 Lambda 통합을 사용하려면 AWS SAM 템플릿을 다운로드하고 Amazon S3에 함수 구성을 작성할 수 있는 특정 권한이 필요합니다.

AWS SAM 템플릿을 다운로드하려면 다음 API 작업을 사용할 권한이 있어야 합니다.

- GetPolicy
- iam:GetPolicyVersion
- iam:GetRole
- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

IAM 사용자 역할에 `AWSLambda_ReadOnlyAccess` AWS 관리형 정책을 추가하여 이러한 모든 작업을 사용할 권한을 부여할 수 있습니다.

Lambda가 Amazon S3에 함수 구성을 작성하려면 다음 API 작업을 사용할 권한이 있어야 합니다.

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

함수 구성을 Application Composer로 내보내기할 수 없는 경우 계정에 이러한 작업에 필요한 권한이 있는지 확인합니다. 필요한 권한이 있지만 여전히 함수 구성을 내보낼 수 없는 경우 Amazon S3에 대한 액세스 권한을 제한할 수 있는 리소스 기반 정책이 있는지 확인합니다.

기타 리소스

Application Composer에서 기존 Lambda 함수를 기반으로 서버리스 애플리케이션을 설계하는 방법에 대한 자세한 자습서는 the section called “코드형 인프라(IaC)”를 참조하세요.

Application Composer 및 AWS SAM을 사용하여 Lambda를 사용하는 완전한 서버리스 애플리케이션을 설계 및 배포하려면 AWS Application Composer 자습서를 AWS 서버리스 패턴 워크숍에서 찾아 따라할 수도 있습니다.

CloudWatch Logs에서 Lambda 사용

Lambda 함수를 사용하여 Amazon CloudWatch Logs 로그 스트림의 로그를 모니터링하고 분석할 수 있습니다. 하나 이상의 로그 스트림에 대해 구독을 생성하여 로그가 생성되거나 선택적 패턴과 일치할 때 함수를 호출할 수 있습니다. 함수를 사용하여 알림을 보내거나 데이터베이스 또는 스토리지에 로그를 영구적으로 유지합니다.

CloudWatch Logs는 로그 데이터를 포함하는 이벤트와 비동기적으로 함수를 호출합니다. 데이터 필드의 값은 Base64로 인코딩된 .gzip 파일 아카이브입니다.

Example CloudWatch Logs 메시지 이벤트

```
{
  "awslogs": {
    "data":
 "ewogICAgIm1lc3NhZ2VUeXBlIjogIkRBVEFfTUVTU0FHRSIsCiAgICAib3duZXIiOiAiMTIzNDU2Nzg5MDEyIiwKICAgI
  }
}
```

디코딩 및 압축 해제된 로그 데이터는 다음 구조를 갖춘 JSON 문서입니다.

Example CloudWatch Logs 메시지 데이터(디코딩됨)

```
{
    "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 Lambda와 함께 AWS CloudFormation 사용

AWS CloudFormation 템플릿에서는 사용자 지정 리소스의 대상으로 Lambda 함수를 지정할 수 있습니다. 사용자 지정 리소스를 사용하여 파라미터를 처리하고, 설정 값을 검색하거나, 스택 수명 주기 이벤트 도중 다른 AWS 서비스를 호출할 수 있습니다.

다음 예시는 템플릿에 정의된 함수를 호출합니다.

Example - 사용자 지정 리소스 정의

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

서비스 토큰은 스택 생성, 업데이트 또는 삭제 시 AWS CloudFormation이 호출한 함수의 ARN(Amazon Resource Name)입니다. `FunctionName`이 원형 그대로 함수로 전달하는, AWS CloudFormation 같은 추가 속성을 포함할 수도 있습니다.

AWS CloudFormation은 콜백 URL을 포함하는 이벤트와 비동기적으로 Lambda 함수를 호출합니다.

Example – AWS CloudFormation 메시지 이벤트

```
{
    "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"
    }
}
```

함수는 성공 또는 실패를 나타내는 콜백 URL에 대한 응답을 반환할 책임을 집니다. 전체 응답 구문은 사용자 지정 리소스 응답 객체에서 확인할 수 있습니다.

Example – AWS CloudFormation 사용자 지정 리소스 응답

```
{
    "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 CloudFormation은 응답 전송을 처리하는 `cfn-response`라는 라이브러리를 제공합니다. 템플릿 내에서 함수를 정의하려면, 함수를 이름으로 요청하면 됩니다. 그러면 AWS CloudFormation이 함수를 위해 생성된 배포 패키지에 라이브러리를 추가합니다.

사용자 지정 리소스에서 사용하는 함수에 탄력적 네트워크 인터페이스가 연결된 경우 다음 리소스를 VPC 정책에 추가합니다. 여기서 **`region`**은 함수가 있는 리전(대시 없음)입니다. 예를 들어 `us-east-1`은 `useast1`입니다. 이렇게 하면 Custom Resource가 AWS CloudFormation 스택으로 신호를 다시 보내는 콜백 URL에 응답할 수 있습니다.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

다음 예시 함수는 두 번째 함수를 호출합니다. 호출이 성공한다면, 함수는 성공 응답을 AWS CloudFormation에 전송하고 스택 업데이트가 진행됩니다. 템플릿은 AWS Serverless Application Model에서 제공하는 AWS::Serverless::Function 리소스 유형을 사용합니다.

Example - 사용자 지정 리소스 함수

```
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
```

사용자 지정 리소스가 호출하는 함수가 템플릿에 정의되어 있지 않다면, AWS CloudFormation 사용 설명서의 cfn-response 모듈에서 `cfn-response`의 소스 코드를 얻을 수 있습니다.

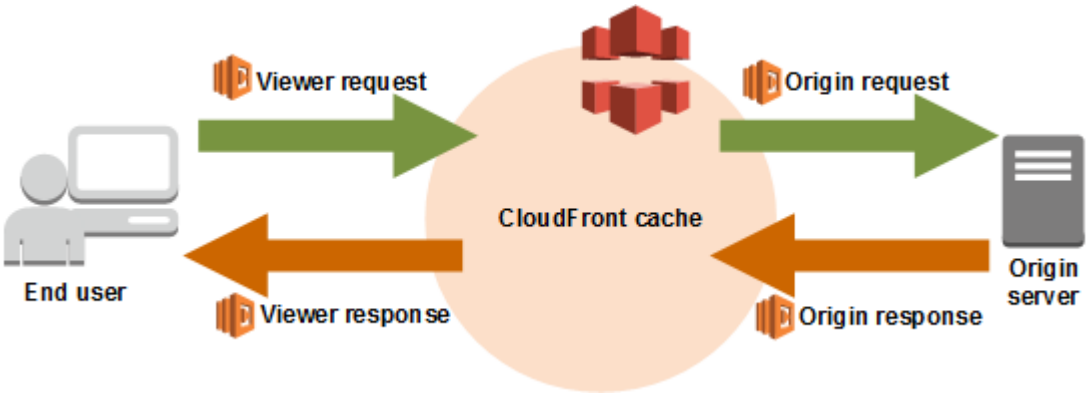
사용자 지정 리소스에 대한 자세한 내용은 AWS CloudFormation 사용 설명서의 사용자 지정 리소스를 참조하세요.

CloudFront Lambda AWS Lambda @Edge 와 함께 사용

Lambda @Edge 는 Python 및 Node.js 함수를 Amazon CloudFront 엣지 로케이션에 배포할 수 있는 확장 AWS Lambda 프로그램입니다. Lambda @Edge 의 일반적인 사용 사례는 함수를 사용하여 배포가 최종 사용자에게 제공하는 CloudFront 콘텐츠를 사용자 지정하는 것입니다. 오리진 서버가 아니라 최종 사용자에게 가까운 위치에서 이들 함수를 간접적으로 호출하므로 지연 시간이 크게 단축되고 사용자 경험이 상당히 개선됩니다.

CloudFront 배포를 Lambda @Edge 함수와 연결하면 엣지 로케이션에서 요청 CloudFront 및 응답을 가로채습니다. CloudFront CloudFront 그런 다음 이벤트를 전송하여 Lambda 함수를 호출합니다. 다음과 같은 이벤트가 발생할 때 Lambda 함수를 CloudFront 호출할 수 있습니다.

- 뷰어로부터 요청을 CloudFront 받는 경우 (뷰어 요청)
- 요청을 오리진에 CloudFront 전달하기 전 (오리진 요청)
- 오리진으로부터 응답을 CloudFront 받는 경우 (오리진 응답)
- 시청자에게 응답을 CloudFront 반환하기 전에 (시청자 응답)



> **Note**
>
> Lambda@Edge는 제한된 런타임 및 기능 집합을 지원합니다. 자세한 내용은 Amazon CloudFront 개발자 안내서의 Lambda 함수에 대한 요구 사항 및 제한을 참조하십시오.

다음은 이벤트의 예입니다. CloudFront

Example CloudFront 메시지 이벤트

```
{
```

```
  "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 사용에 대한 자세한 내용은 Lambda @Edge 사용을 CloudFront 참조하십시오.

AWS Lambda와 함께 AWS CodeCommit 사용

AWS CodeCommit 리포지토리에 대한 트리거를 생성하여 리포지토리의 이벤트가 Lambda 함수를 호출하게 할 수 있습니다. 예를 들어, 브랜치 또는 태그가 생성되거나 기존 브랜치를 푸시할 때 Lambda 함수를 호출할 수 있습니다.

Example AWS CodeCommit 메시지 이벤트

```
{
    "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"
        }
    ]
}
```

자세한 내용은 AWS CodeCommit 리포지토리 트리거 관리를 참조하세요.

Amazon Cognito에서 AWS Lambda 사용

Amazon Cognito 이벤트 기능을 사용하면 Amazon Cognito o의 이벤트에 반응하여 Lambda 함수를 실행할 수 있습니다. Amazon Cognito는 웹 및 모바일 앱에 대한 인증, 권한 부여 및 사용자 관리를 제공합니다. Amazon Cognito에서 중요한 이벤트에 대한 응답으로 Lambda 함수를 호출할 수 있습니다. 예를 들어 동기화 트리거 이벤트를 사용하여 데이터 세트가 동기화될 때마다 게시되는 Lambda 함수를 호출할 수 있습니다. 자세한 내용 및 예제를 살펴보려면 모바일 개발 블로그의 Introducing Amazon Cognito Events: Sync Triggers를 참조하세요.

Example Amazon Cognito 메시지 이벤트

```
{
  "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 이벤트 구독 구성을 통해 이벤트 소스 매핑을 구성합니다. 이벤트 소스 매핑 및 샘플 이벤트에 대한 자세한 내용은 Amazon Cognito 개발자 안내서의 Amazon Cognito 이벤트를 참조하세요.

Amazon Connect에서 Lambda 사용

Lambda 함수를 사용하여 Amazon Connect의 요청을 처리할 수 있습니다. Amazon Connect를 사용하여 클라우드 콜 센터를 구축할 수 있습니다.

Amazon Connect는 요청 본문 및 메타데이터가 포함된 이벤트와 동기적으로 Lambda 함수를 호출합니다.

Example Amazon Connect 요청 이벤트

```
{
  "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"
}
```

Lambda에 Amazon Connect를 사용하는 방법에 대한 자세한 내용은 Amazon Connect 관리자 안내서에서 Lambda 함수 호출을 참조하세요.

Amazon EC2에서 AWS Lambda 사용

AWS Lambda를 사용해 Amazon Elastic Compute Cloud의 수명 주기 이벤트를 처리하고 Amazon EC2 리소스를 관리할 수 있습니다. Amazon EC2는 인스턴스가 상태를 변경하는 경우, Amazon Elastic Block Store 볼륨 스냅샷이 완료되는 경우, 또는 스팟 인스턴스가 종료되도록 예약된 경우와 같은 수명 주기 이벤트를 Amazon EventBridge(CloudWatch Events)에 보냅니다. 처리를 위해 이러한 이벤트를 Lambda 함수로 전달하도록 EventBridge(CloudWatch Events)를 구성합니다.

EventBridge(CloudWatch Events)는 Amazon EC2의 이벤트 문서와 비동기적으로 Lambda 함수를 호출합니다.

Example 인스턴스 수명 주기 이벤트

```
{
    "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"
    }
}
```

이벤트를 구성하는 방법에 대한 자세한 내용은 Amazon EventBridge 스케줄러와 함께 Lambda 사용 섹션을 참조하세요. Amazon EBS 스냅샷 알림을 처리하는 예제 함수는 EventBridge Scheduler for Amazon EBS를 참조하세요.

AWS SDK를 사용하여 Amazon EC2 API로 인스턴스 및 기타 리소스를 관리할 수도 있습니다.

권한

Amazon EC2의 수명 주기 이벤트를 처리하려면 EventBridge(CloudWatch Events)가 함수를 호출할 권한이 필요합니다. 이 권한은 함수의 리소스 기반 정책에서 비롯됩니다. EventBridge(CloudWatch

Events) 콘솔을 사용하여 이벤트 트리거를 구성하면 콘솔이 사용자를 대신하여 리소스 기반 정책을 업데이트합니다. 또는, 다음과 같은 문을 추가합니다.

Example Amazon EC2 수명 주기 알림을 위한 리소스 기반 정책 문

```
{
  "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/*"
    }
  }
}
```

문을 추가하려면 `add-permission` AWS CLI 명령을 사용합니다.

```
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/*'
```

함수가 AWS SDK를 사용하여 Amazon EC2 리소스를 관리하는 경우 Amazon EC2 권한을 함수의 실행 역할에 추가합니다.

자습서: Amazon VPC에서 Amazon ElastiCache에 액세스하도록 Lambda 함수 구성

Amazon VPC에서 Amazon ElastiCache에 액세스하도록 Lambda를 구성하는 방법을 알아보려면 ElastiCache for Redis 사용 설명서의 Lambda 자습서를 참조하십시오.

Lambda로 Application Load Balancer 요청 처리

Lambda 함수를 사용하여 Application Load Balancer의 요청을 처리할 수 있습니다. Elastic Load Balancing은 Application Load Balancer의 대상으로 Lambda 함수를 지원합니다. 로드 밸런서 규칙을 사용하여 경로 또는 헤더 값을 기반으로 HTTP 요청을 함수에 라우팅합니다. 요청을 처리하고 Lambda 함수의 HTTP 응답을 반환합니다.

Elastic Load Balancing은 요청 본문 및 메타데이터가 포함된 이벤트와 동기적으로 Lambda 함수를 호출합니다.

Example Application Load Balancer 요청 이벤트

```
{
    "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
```

```
}
```

함수는 이벤트를 처리하고 JSON 형식의 응답 문서를 로드 밸런서에 반환합니다. Elastic Load Balancing은 이 문서를 HTTP 성공 또는 오류 응답으로 변환하고 사용자에게 반환합니다.

Example 응답 문서 형식

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

Application Load Balancer를 함수 트리거로 구성하려면 함수 실행 권한을 Elastic Load Balancing에 부여하고 해당 요청을 함수로 라우팅하는 대상 그룹을 만든 다음, 대상 그룹에 요청을 보내는 로드 밸런서에 하나의 규칙을 추가하세요.

`add-permission` 명령을 사용하여 권한 설명문을 함수의 리소스 기반 정책에 추가하세요.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

다음 결과가 표시됩니다:

```
{
    "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 리스너와 대상 그룹을 구성하는 방법에 대한 지침은 Application Load Balancers 사용 설명서에서 대상으로서 Lambda 함수를 참조하세요.

Lambda에서 Amazon EFS 사용

Lambda는 Amazon Elastic File System(Amazon EFS)과 통합하여 Lambda 애플리케이션에 대한 안전한 공유 파일 시스템 액세스를 지원합니다. VPC 내의 로컬 네트워크를 통해 NFS 프로토콜에서 초기화가 수행되는 동안 파일 시스템을 마운트하도록 함수를 구성할 수 있습니다. Lambda는 연결을 관리하고 파일 시스템과 주고 받는 모든 트래픽을 암호화합니다.

파일 시스템과 Lambda 함수는 동일한 리전에 있어야 합니다. 한 계정의 Lambda 함수는 다른 계정에 파일 시스템을 마운트할 수 있습니다. 이 시나리오에서 함수 VPC와 파일 시스템 VPC 간에 VPC 피어링을 구성합니다.

> **Note**
>
> 파일 시스템에 연결되도록 함수를 구성하는 방법은 Lambda 함수에 대한 파일 시스템 액세스 구성 단원을 참조하세요.

Amazon EFS는 여러 함수가 동시에 동일한 파일 시스템에 쓰기 작업을 시도하는 경우 손상을 방지하기 위해 파일 잠금을 지원합니다. Amazon EFS의 잠금은 권고 잠금에 대한 v4.1 프로토콜을 따르며, 애플리케이션에서 전체 파일과 바이트 범위 잠금을 모두 사용할 수 있게 해줍니다.

Amazon EFS는 대규모로 고성능을 유지해야 하는 애플리케이션의 요구에 따라 파일 시스템을 사용자 지정할 수 있는 옵션을 제공합니다. 고려해야 할 주요 요소는 연결 수, 처리량(초당 MiB) 및 IOPS입니다.

> **할당량**
>
> 파일 시스템 할당량 및 제한에 대한 자세한 내용은 Amazon Elastic File System 사용 설명서의 Amazon EFS 파일 시스템에 대한 할당량을 참조하세요.

조정, 처리량 및 IOPS와 관련된 문제를 방지하려면 Amazon EFS가 Amazon에 전송하는 지표를 모니터링하십시오. CloudWatch Amazon EFS의 모니터링에 대한 개요는 Amazon Elastic File System 사용 설명서의 Amazon EFS 모니터링을 참조하세요.

Sections

- IOPS

연결

Amazon EFS는 파일 시스템당 최대 25,000개의 연결을 지원합니다. 초기화하는 동안 함수의 각 인스턴스는 호출 간에 지속되는 파일 시스템에 대한 단일 연결을 생성합니다. 즉, 파일 시스템에 연결된 하나 이상의 함수에서 25,000의 동시성에 도달할 수 있습니다. 함수가 생성하는 연결 수를 제한하려면 예약된 동시성을 사용하세요.

그러나 대규모로 함수의 코드 또는 구성을 변경하면 함수 인스턴스 수가 현재 동시성을 초과하여 일시적으로 증가합니다. Lambda는 새 요청을 처리하기 위해 새 인스턴스가 프로비저닝되며, 이전 인스턴스가 파일 시스템에 대한 연결을 닫기 전에 약간의 지연이 발생합니다. 배포 중에 최대 연결 제한에 도달하지 않게 하려면 롤링 배포를 사용합니다. 롤링 배포에서는 변경을 수행할 때마다 트래픽을 새 버전으로 점진적으로 마이그레이션합니다.

Amazon EC2 등의 다른 서비스에서 동일한 파일 시스템에 연결하는 경우, Amazon EC2에서 연결에 대한 규모 조정 동작을 파악하고 있어야 합니다. 파일 시스템은 버스트에서 최대 3,000개의 연결을 생성할 수 있으며, 그 이후에 분당 500개의 새 연결을 지원합니다.

연결에 대한 경보를 모니터링 및 트리거하려면 `ClientConnections` 지표를 사용합니다.

처리량

규모에 따라 파일 시스템의 최대 처리량을 초과할 수도 있습니다. 버스트 모드(기본값)에서 파일 시스템의 기준 처리량은 낮으며, 크기에 따라 선형으로 확장됩니다. 활동 버스트를 허용하기 위해 파일 시스템에 100MiB/s 이상의 처리량을 사용할 수 있는 버스트 크레딧이 부여됩니다. 크레딧은 지속적으로 누적되며, 모든 읽기 및 쓰기 작업에 사용됩니다. 파일 시스템에 크레딧이 부족하면 읽기 및 쓰기 작업이 기본 처리량을 초과해 제한되며, 이로 인해 호출 시간이 초과될 수 있습니다.

> **Note**
>
> 프로비저닝된 동시성을 사용하는 경우 함수가 유휴 상태에서도 버스트 크레딧을 사용할 수 있습니다. Lambda는 프로비저닝된 동시성을 사용하여 호출되기 전에 함수의 인스턴스를 초기화하고 몇 시간마다 인스턴스를 재활용합니다. 초기화 중에 첨부 파일 시스템의 파일을 사용하는 경우, 이러한 활동에서는 모든 버스트 크레딧을 사용할 수 있습니다.

처리량에 대한 경보를 모니터링 및 트리거하려면 `BurstCreditBalance` 지표를 사용합니다. 지표는 함수의 동시성이 낮을 때는 감소하고 높을 때는 증가합니다. 피크 트래픽을 처리하기 위한 활동이 적은

동안에 지표가 항상 감소하거나 누적되지 않으면 함수의 동시성을 제한하거나 프로비저닝된 처리량을 활성화해야 할 수 있습니다.

IOPS

초당 입/출력 작업 수(IOPS)는 파일 시스템에서 처리되는 읽기 및 쓰기 작업 수를 측정한 값입니다. 범용 모드에서는 지연 시간을 줄이기 위해 IOPS가 제한되므로 대부분의 애플리케이션에 유용합니다.

범용 모드에서 IOPS를 모니터링하고 경보를 발행하려면 `PercentIOLimit` 지표를 사용합니다. 이 지표가 100%에 도달하면 함수가 읽기 및 쓰기 작업이 완료되기를 기다리며 시간 초과될 수 있습니다.

Amazon EventBridge 스케줄러와 함께 Lambda 사용

Amazon EventBridge 스케줄러는 하나의 중앙 관리형 서비스에서 작업을 생성, 실행 및 관리할 수 있는 서버리스 스케줄러입니다. EventBridge 스케줄러를 사용하면 반복 패턴에 대해 cron 및 rate 표현식을 사용하여 일정을 만들거나 일회성 간접 호출을 구성할 수 있습니다. 전송을 위한 유연한 기간을 설정하고, 재시도 제한을 정의하고, 처리되지 않은 이벤트의 최대 보존 시간을 설정할 수 있습니다.

Lambda와 함께 EventBridge Scheduler를 설정하는 경우 EventBridge Scheduler는 Lambda 함수를 비동기적으로 호출합니다. 이 페이지에서는 EventBridge 스케줄러를 사용하여 일정에 따라 Lambda 함수를 간접적으로 호출하는 방법을 설명합니다.

실행 역할 설정

새 일정을 생성할 때 EventBridge 스케줄러에 사용자를 대신하여 대상 API 작업을 간접적으로 호출할 수 있는 권한이 있어야 합니다. 실행 역할을 사용하여 EventBridge 스케줄러에 이러한 권한을 부여합니다. 일정의 실행 역할에 연결하는 권한 정책은 필요한 권한을 정의합니다. 이러한 권한은 EventBridge 스케줄러가 간접적으로 호출하려는 대상 API에 따라 달라집니다.

다음 절차와 같이 EventBridge 스케줄러 콘솔을 사용하여 일정을 생성하면 EventBridge 스케줄러가 선택한 대상을 기준으로 실행 역할을 자동으로 설정합니다. EventBridge 스케줄러 SDK, AWS CLI 또는 AWS CloudFormation 중 하나를 사용하여 일정을 생성하려면 EventBridge 스케줄러가 대상을 간접적으로 호출하는 데 필요한 권한을 부여하는 기존 실행 역할이 있어야 합니다. 일정에 대한 실행 역할을 수동으로 설정하는 방법에 대한 자세한 내용은 EventBridge 스케줄러 사용 설명서의 Setting up an execution role을 참조하세요.

일정 생성

콘솔을 사용하여 일정 생성

1. https://console.aws.amazon.com/scheduler/home에서 Amazon EventBridge 스케줄러 콘솔을 엽니다.
2. 일정 페이지에서 일정 생성을 선택합니다.
3. 일정 세부 정보 지정 페이지의 일정 이름 및 설명 섹션에서 다음을 수행합니다.

    a. 일정 이름에 일정의 이름을 입력합니다. 예: **`MyTestSchedule`**

    b. (선택 사항) 설명에 일정에 대한 설명을 입력합니다. 예: **`My first schedule`**

    c. 일정 그룹 드롭다운 목록에서 일정 그룹을 선택합니다. 그룹이 없는 경우 기본값을 선택합니다. 일정 그룹을 생성하려면 자체 일정 생성을 선택합니다.

일정 그룹을 사용하여 일정 그룹에 태그를 추가합니다.

4. • 일정 옵션을 선택합니다.

발생	수행할 작업	
일회성 일정 일회성 일정은 사용자가 지정하는 날짜와 시간에 한 번만 대상을 간접적으로 호출합니다.	날짜 및 시간에 대해 다음을 수행합니다. • YYYY/MM/DD 형식으로 유효한 날짜를 입력합니다. • 24시간 hh:mm 형식으로 타임스탬프를 입력합니다. • 시간대에서 시간대를 선택하세요.	
반복되는 일정 반복 일정은 cron 표현식 또는 rate 표현식을 사용하여 지정한 속도로 대상을 간접적으로 호출합니다.	a. 일정 유형에서 다음 중 하나를 수행합니다. • cron 표현식을 사용하여 일정을 정의하려면 Cron 기반 일정을 선택하고 cron 표현식을 입력합니다. • rate 표현식을 사용하여 일정을 정의하려면 Rate 기반 일정을 선택하고 rate 표현식을 입력합니다. cron 및 rate 표현식에 대한 자세한 내용은 Amazon EventBridge 스케줄러 사용 설명서의 EventBridge 스케줄	

발생	수행할 작업	
	러의 스케줄 유형을 참조하세요. b. 유연한 기간에서 끄기를 선택하여 옵션을 끄거나 미리 정의된 기간 중 하나를 선택합니다. 예를 들어, 15분을 선택하고 1시간에 한 번씩 대상을 간접적으로 호출하도록 반복 일정을 설정하면 일정은 매시간 시작 후 15분 이내에 실행됩니다.	

5. (선택 사항) 이전 단계에서 반복 일정을 선택한 경우 기간 섹션에서 다음을 수행합니다.
   a. 시간대에서 시간대를 선택합니다.
   b. 시작 날짜 및 시간에 YYYY/MM/DD 형식으로 유효한 날짜를 입력한 다음 24시간 hh:mm 형식으로 타임스탬프를 지정합니다.
   c. 종료 날짜 및 시간에 YYYY/MM/DD 형식으로 유효한 날짜를 입력한 다음 24시간 hh:mm 형식으로 타임스탬프를 지정합니다.
6. 다음을 선택합니다.
7. 대상 선택 페이지에서 EventBridge 스케줄러가 간접적으로 호출하는 AWS API 작업을 선택합니다.
   a. AWS Lambda Invoke를 선택합니다.
   b. 간접 호출 섹션에서 함수를 선택하거나 새 Lambda 함수 생성을 선택합니다.
   c. (선택 사항) JSON 페이로드를 입력합니다. 페이로드를 입력하지 않으면 EventBridge 스케줄러는 빈 이벤트를 사용하여 함수를 간접적으로 호출합니다.
8. 다음을 선택합니다.
9. 설정 페이지에서 다음 작업을 수행합니다.
   a. 일정을 켜려면 일정 상태에서 일정 활성화를 토글합니다.
   b. 일정에 대한 재시도 정책을 구성하려면 재시도 정책 및 데드-레터 큐(DLQ)에서 다음을 수행합니다.

- 재시도를 토글합니다.
- 최대 이벤트 수명에 EventBridge 스케줄러가 처리되지 않은 이벤트를 보관해야 하는 최대 시간과 분을 입력합니다.
- 최대 시간은 24시간입니다.
- 최대 재시도 횟수에는 대상이 오류를 반환할 경우 EventBridge 스케줄러가 일정을 재시도하는 최대 횟수를 입력합니다.

  최댓값은 185회입니다.

재시도 정책을 사용하면 일정이 대상을 간접적으로 호출하지 못할 경우 EventBridge 스케줄러가 일정을 다시 실행합니다. 구성된 경우 일정에 대한 최대 보존 기간과 재시도 횟수를 설정해야 합니다.

c. EventBridge 스케줄러가 전송되지 않은 이벤트를 저장하는 위치를 선택합니다.

DLQ(Dead Letter Queue) 옵션	수행할 작업
저장 안 함	None을 선택합니다.
일정을 생성하는 위치와 같은 AWS 계정에 이벤트 저장	a. 내 AWS 계정의 Amazon SQS 대기열을 DLQ로 선택을 선택합니다. b. Amazon SQS 대기열의 Amazon 리소스 이름(ARN)을 선택합니다.
일정을 생성하는 위치와 다른 AWS 계정에 이벤트 저장	a. 다른 AWS 계정의 Amazon SQS 대기열을 DLQ로 지정을 선택합니다. b. Amazon SQS 대기열의 Amazon 리소스 이름(ARN)을 입력합니다.

d. 고객 관리형 키를 사용하여 대상 입력을 암호화하려면 암호화에서 암호화 설정 사용자 지정(고급)을 선택합니다.

이 옵션을 선택하는 경우 기존 KMS 키 ARN을 입력하거나 AWS KMS key 생성을 선택하여 AWS KMS 콘솔로 이동합니다. EventBridge 스케줄러가 저장 데이터를 암호화하는 방법에 대한 자세한 내용은 Amazon EventBridge 스케줄러 사용 설명서의 Encryption at rest를 참조하세요.

e. EventBridge 스케줄러가 새 실행 역할을 생성하도록 하려면 이 일정에 대한 새 역할 생성을 선택합니다. 그런 다음 역할 이름을 입력합니다. 이 옵션을 선택하면 EventBridge 스케줄러가 템플릿 대상에 필요한 필수 권한을 역할에 연결합니다.

10. 다음을 선택합니다.
11. 일정 검토 및 생성 페이지에서 일정의 세부 정보를 검토합니다. 각 섹션에서 편집을 선택하여 해당 단계로 돌아가서 세부 정보를 편집합니다.
12. 일정 생성을 선택합니다.

    일정 페이지에서 새 일정과 기존 일정 목록을 볼 수 있습니다. 상태 열에서 새 일정이 활성화됨 상태인지 확인합니다.

EventBridge 스케줄러가 함수를 간접적으로 호출했는지 확인하려면 함수의 Amazon CloudWatch 로그를 확인합니다.

관련 리소스

EventBridge 스케줄러에 대한 자세한 내용은 다음을 참조하세요.

- EventBridge 스케줄러 사용 설명서
- EventBridge 스케줄러 API 참조
- EventBridge 스케줄러 요금

AWS Lambda와 함께 AWS IoT 사용

AWS IoT는 인터넷에 연결된 디바이스(예: 센서)와 AWS 클라우드 간의 안전한 통신을 제공합니다. 이를 통해 여러 디바이스에서 원격 측정 데이터를 수집하고, 저장하고, 분석할 수 있습니다.

디바이스가 AWS IoT 서비스와 상호 작용하는 AWS 규칙을 생성할 수 있습니다. AWS IoT 규칙 엔진은 SQL 기반 언어를 사용하여 메시지 페이로드에서 데이터를 선택하고, Amazon S3, Amazon DynamoDB, AWS Lambda 등의 다른 서비스로 데이터를 전송할 수 있습니다. 다른 AWS 서비스 또는 서드 파티 서비스를 호출할 경우에는 Lambda 함수를 호출하는 규칙을 정의합니다.

들어오는 IoT 메시지가 규칙을 트리거하면 AWS IoT는 Lambda 함수를 비동기적으로 호출하고 IoT 메시지의 데이터를 함수로 전달합니다.

다음 예는 온실 센서의 수분 수치를 보여줍니다. 행 및 pos 값으로 센서의 위치를 식별합니다. 이 예제 이벤트는 AWS IoT 규칙 자습서의 온실 유형을 기반으로 합니다.

Example AWS IoT 메시지 이벤트

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

비동기 호출의 경우 함수에서 오류를 반환할 때 Lambda에서 메시지를 대기열에 배치하고 재시도합니다. 함수에서 처리하지 못한 이벤트를 유지할 대상과 함께 함수를 구성합니다.

Lambda 함수를 호출하려면 AWS IoT 서비스에 대한 권한을 부여해야 합니다. `add-permission` 명령을 사용하여 권한 설명문을 함수의 리소스 기반 정책에 추가하세요.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

다음 결과가 표시됩니다:

```
{
    "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 IoT에서 Lambda를 사용하는 방법에 관한 자세한 내용은 AWS Lambda 규칙 생성을 참조하세요.

Amazon Data Firehose에서 AWS Lambda 사용

Amazon Data Firehose는 Managed Service for Apache Flink 또는 Amazon S3 등의 다운스트림 서비스에 대한 스트리밍 데이터를 캡쳐, 변환, 로드합니다. 다운스트림으로 보내기 전에 Lambda 함수를 작성하여 추가로 사용자 지정된 데이터 처리를 요청할 수 있습니다.

Example Amazon Data Firehose 메시지 이벤트


```
{
  "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": ""
      }
    }
  ]
}
```


자세한 내용은 Firehose 개발자 안내서의 Amazon Data Firehose 데이터 변환을 참조하세요.

Amazon Lex에서 AWS Lambda 사용

Amazon Lex를 사용하여 대화 봇을 애플리케이션에 통합할 수 있습니다. Amazon Lex 봇은 사용자와 대화할 수 있는 인터페이스를 제공합니다. Amazon Lex는 사전 빌드된 Lambda 통합을 제공하므로 Amazon Lex 봇에서 Lambda 함수를 사용할 수 있습니다.

Amazon Lex 봇을 구성할 때 검증, 이행 또는 둘 모두를 수행하는 Lambda 함수를 지정할 수 있습니다. 검증을 위해, Amazon Lex는 사용자로부터 각 응답 후에 Lambda 함수를 호출합니다. Lambda 함수는 응답을 검증하고 필요한 경우 사용자에게 수정 피드백을 제공할 수 있습니다. 이행을 위해, Amazon Lex는 봇이 필요한 모든 정보를 성공적으로 수집하고 사용자로부터 확인을 수신한 후에 Lambda 함수를 호출하여 사용자 요청을 이행합니다.

Lambda 함수의 동시성을 관리하여 제공되는 최대 동시 봇 대화 수를 제어할 수 있습니다. 함수가 최대 동시성에 있을 경우 Amazon Lex API는 HTTP 429 상태 코드(요청이 너무 많음)를 반환합니다.

Lambda 함수가 예외를 발생시키는 경우 API는 HTTP 424 상태 코드(종속성 실패 예외)를 반환합니다.

Amazon Lex 봇은 Lambda 함수를 동기적으로 호출합니다. 이벤트 파라미터에는 대화에서의 각 슬롯 값과 봇에 대한 정보가 들어 있습니다. 이벤트 및 응답 필드의 정의는 Amazon Lex 개발자 안내서의 Lambda 이벤트 및 응답 형식을 참조하세요. Amazon Lex 메시지 이벤트의 `invocationSource` 파라미터는 Lambda 함수가 입력 DialogCodeHook () 을 검증해야 하는지 아니면 인텐트 () 를 충족해야 하는지를 나타냅니다. FulfillmentCodeHook

Amazon Lex와 함께 Lambda를 사용하는 방법을 보여주는 자습서 예시는 Amazon Lex 개발자 안내서의 연습 1: 블루프린트를 사용하여 Amazon Lex 봇 만들기를 참조하세요.

역할 및 권한

서비스 연결 역할을 함수의 실행 역할로 구성해야 합니다. Amazon Lex는 사전 정의된 권한으로 서비스 연결 역할을 정의합니다. 콘솔을 사용하여 Amazon Lex 봇을 생성하는 경우 서비스 연결 역할이 자동으로 생성됩니다. AWS CLI를 사용하여 서비스 연결 역할을 생성하려면 `create-service-linked-role` 명령을 사용합니다.

```
aws iam create-service-linked-role --aws-service-name lex.amazonaws.com
```

이 명령은 다음과 같은 역할을 생성합니다.

```
{
  "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 함수가 다른 AWS 서비스를 사용하는 경우 해당 권한을 서비스 연결 역할에 추가해야 합니다.

리소스 기반 권한 정책을 사용하여 Amazon Lex 봇이 Lambda 함수를 호출할 수 있도록 합니다. Amazon Lex 콘솔을 사용하는 경우 권한 정책이 자동으로 생성됩니다. AWS CLI에서 Lambda `add-permission` 명령을 사용하여 권한을 설정합니다.

Amazon Lex V2의 경우 다음 명령을 실행합니다. 소스 ARN에서 AWS 리전 Amazon Lex 봇이 들어 있는 `us-east-1`(으)로 바꾸고 고유한 AWS 계정 번호와 봇 별칭을 사용하세요.

```
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 V1을 사용하여 Lambda 함수를 간접적으로 호출할 수도 있습니다. Amazon Lex V1의 경우 다음 명령을 실행합니다. 소스 ARN에서 AWS 리전 Amazon Lex 의도가 들어 있는 `us-east-1`(으)로 바꾸고 고유한 AWS 계정 번호와 의도 이름을 사용하세요.

```
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 V1은 더 이상 유지 관리되지 않는다는 점에 유의하세요. Amazon Lex V2를 사용하는 것이 좋습니다.

Amazon RDS와 함께 AWS Lambda 사용

Lambda 함수를 Amazon RDS 프록시를 통해 Amazon Relational Database Service(Amazon RDS) 데이터베이스에 직접 연결할 수 있습니다. 간단한 시나리오에서는 직접 연결이 유용하며 프로덕션에서는 프록시를 사용하는 것이 좋습니다. 데이터베이스 프록시는 함수가 데이터베이스 연결을 소진하지 않고 높은 동시성 레벨에 도달할 수 있도록 하는 공유 데이터베이스 연결 풀을 관리합니다.

짧게 데이터베이스를 연결하거나 다수의 데이터베이스 연결을 열고 닫는 Lambda 함수에는 Amazon RDS 프록시를 사용하는 것이 좋습니다.

함수 구성

Lambda 콘솔에서는 Amazon RDS 데이터베이스 인스턴스와 프록시 리소스를 프로비저닝하고 구성할 수 있습니다. 자세한 내용은 구성 탭에서 RDS 데이터베이스를 참조하세요. 또는 Amazon RDS 콘솔에서 Lambda 함수에 대한 연결을 생성하고 구성할 수도 있습니다.

- 데이터베이스에 연결하려면 데이터베이스를 실행하는 동일한 Amazon VPC에 함수가 있어야 합니다.
- MySQL, MariaDB, PostgreSQL 또는 Microsoft SQL Server 엔진과 함께 Amazon RDS 데이터베이스를 사용할 수 있습니다.
- MySQL 또는 PostgreSQL 엔진과 함께 Aurora DB 클러스터를 사용할 수도 있습니다.
- 데이터베이스 인증을 위해 Secrets Manager 보안 암호를 제공해야 합니다.
- IAM 역할은 보안 암호를 사용할 권한을 제공해야 하며, 신뢰 정책에서는 Amazon RDS가 역할을 수임하도록 허용해야 합니다.
- 콘솔을 사용하여 Amazon RDS 리소스를 구성하고 이를 함수에 연결하는 IAM 보안 주체에는 다음 권한이 있어야 합니다.

> **Note**
>
> 데이터베이스 연결 풀을 관리하기 위해 Amazon RDS 프록시를 구성하는 경우에만 Amazon RDS 프록시 권한이 필요합니다.

Example 권한 정책

```
{
  "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 RDS는 데이터베이스 인스턴스 크기를 기준으로 프록시에 시간당 요금을 부과합니다. 자세한 내용은 RDS 프록시 요금을 참조하세요. 일반적인 프록시 연결에 대한 자세한 내용은 Amazon RDS 사용 설명서에서 Amazon RDS 프록시 사용을 참조하세요.

> **Lambda 및 Amazon RDS 설정**
>
> Lambda와 Amazon RDS 콘솔 모두 Lambda와 Amazon RDS를 연결하는 데 필요한 일부 리소스를 자동으로 구성하는 데 도움이 됩니다.

Lambda 함수를 사용하여 Amazon RDS 데이터베이스에 연결합니다.

다음 코드 예제는 Amazon RDS 데이터베이스에 연결하는 Lambda 함수를 구현하는 방법을 보여줍니다. 이 함수는 간단한 데이터베이스 요청을 하고 결과를 반환합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda 함수에서 Amazon RDS 데이터베이스에 연결

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Javascript를 사용하여 Lambda 함수에서 Amazon RDS 데이터베이스에 연결

```
// 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 RDS의 이벤트 알림 처리

Lambda를 사용하여 Amazon RDS 데이터베이스의 이벤트 알림을 처리할 수 있습니다. Amazon RDS는 Amazon Simple Notification Service(Amazon SNS) 주제에 알림을 전송합니다. 이 알림이 Lambda 함수를 호출하도록 구성할 수 있습니다. Amazon SNS는 Amazon RDS의 메시지를 자체 이벤트 문서로 래핑하여 함수에 이를 전송합니다.

알림 전송을 위한 Amazon RDS 데이터베이스 구성에 대한 자세한 내용은 Amazon RDS 이벤트 알림 사용을 참조하세요.

Example Amazon SNS 이벤트의 Amazon RDS 메시지


```
{
        "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 및 Amazon RDS 자습서

- Amazon RDS 데이터베이스에 액세스하기 위해 Lambda 함수 사용 – Amazon RDS 사용 설명서에서 Lambda 함수를 사용하여 Amazon RDS 프록시를 통해 Amazon RDS 데이터베이스에 데이터를 쓰

는 방법을 알아보세요. Lambda 함수는 메시지가 추가될 때마다 Amazon SQS 대기열에서 레코드를 읽고 데이터베이스의 테이블에 새 항목을 기록합니다.

Lambda를 사용하여 Amazon S3 이벤트 알림 처리

Lambda를 사용하여 Amazon Simple Storage Service의 이벤트 알림을 처리할 수 있습니다. Amazon S3는 객체가 생성되거나 삭제될 때 이벤트를 Lambda 함수에 전송할 수 있습니다. 버킷에 알림 설정을 구성하고 Amazon S3에 함수의 리소스 기반 권한 정책에 따라 함수를 호출할 수 있는 권한을 부여합니다.

> **Warning**
>
> Lambda 함수가 해당 함수를 트리거하는 동일한 버킷을 사용하는 경우 함수가 루프에서 실행될 수 있습니다. 예를 들어 객체가 업로드될 때마다 버킷이 함수를 트리거하고 그 함수가 객체를 버킷에 업로드하는 경우, 함수는 간접적으로 자신을 트리거합니다 이렇게 되지 않도록 하려면 두 개의 버킷을 사용하거나, 수신 객체에 사용되는 접두사에만 적용되도록 트리거를 구성합니다.

Amazon S3는 객체에 대한 세부 정보를 포함하는 이벤트와 비동기적으로 함수를 호출합니다. 다음 예제에서는 배포 패키지가 Amazon S3에 업로드될 때 Amazon S3에서 전송한 이벤트를 보여줍니다.

Example Amazon S3 알림 이벤트

```
{
  "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"
        }
      }
    }
  ]
}
```

함수를 호출하려면 Amazon S3는 함수의 리소스 기반 정책의 권한이 필요합니다. Lambda 콘솔에서 Amazon S3 트리거를 구성할 때, 콘솔은 버킷 이름과 계정 ID가 일치할 경우 Amazon S3에서 함수를 호출할 수 있도록 리소스 기반 정책을 수정합니다. Amazon S3에서 알림을 구성할 경우, Lambda API를 사용하여 정책을 업데이트합니다. 또한 Lambda API를 사용하여 다른 계정에 권한을 부여하거나 지정된 별칭으로 권한을 제한할 수도 있습니다.

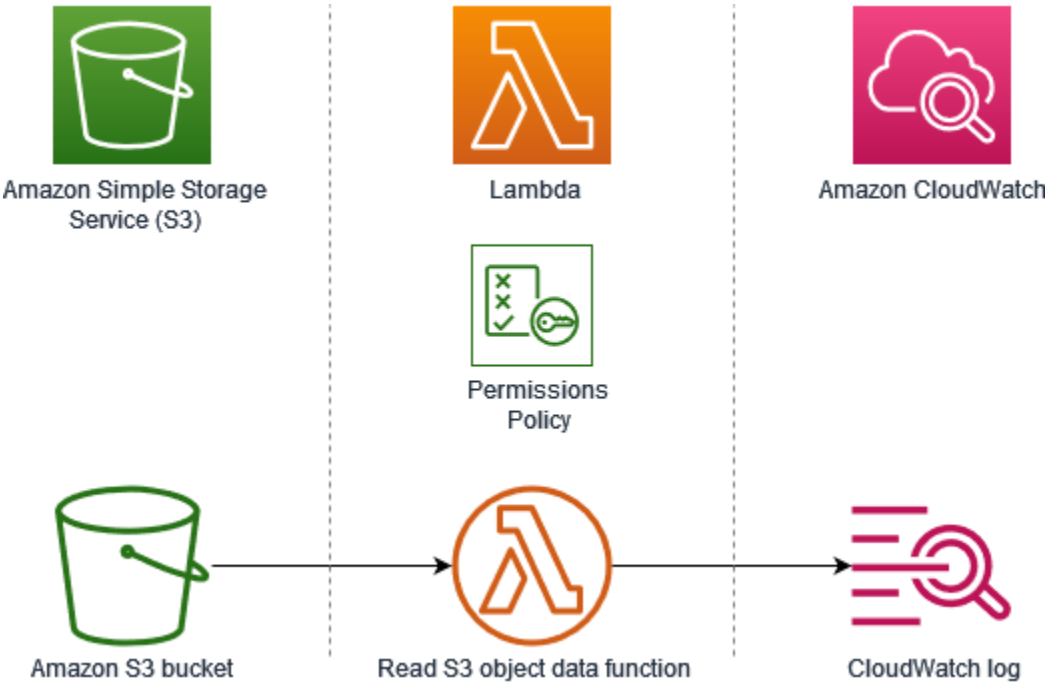
함수에서 AWS SDK를 사용하여 Amazon S3 리소스를 관리하는 경우, 실행 역할에 Amazon S3 권한이 있어야 합니다.

주제

- 자습서: Amazon S3 트리거를 사용하여 Lambda 함수 호출
- 자습서: Amazon S3 트리거를 사용하여 썸네일 이미지 생성

자습서: Amazon S3 트리거를 사용하여 Lambda 함수 호출

이 자습서에서는 콘솔을 사용하여 Lambda 함수를 생성하고 Amazon Simple Storage Service(Amazon S3) 버킷에 대한 트리거를 구성합니다. Amazon S3 버킷에 객체를 추가할 때마다 함수가 실행되고 Amazon CloudWatch Logs에 객체 유형을 출력합니다.



이 자습서에서는 다음을 수행하는 방법을 설명합니다.

1. Amazon S3 버킷을 생성합니다.
2. Amazon S3 버킷에 있는 객체의 유형을 반환하는 Lambda 함수를 생성합니다.
3. 객체가 버킷에 업로드될 때 함수를 호출하는 Lambda 트리거를 구성합니다.
4. 먼저 더미 이벤트로 함수를 테스트한 다음 트리거를 사용하여 함수를 테스트합니다.

이 단계를 완료하면 Amazon S3 버킷에서 객체가 추가되거나 삭제될 때마다 실행되도록 Lambda 함수를 구성하는 방법을 알게 됩니다. AWS Management Console만 사용하여 이 자습서를 완료할 수 있습니다.

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

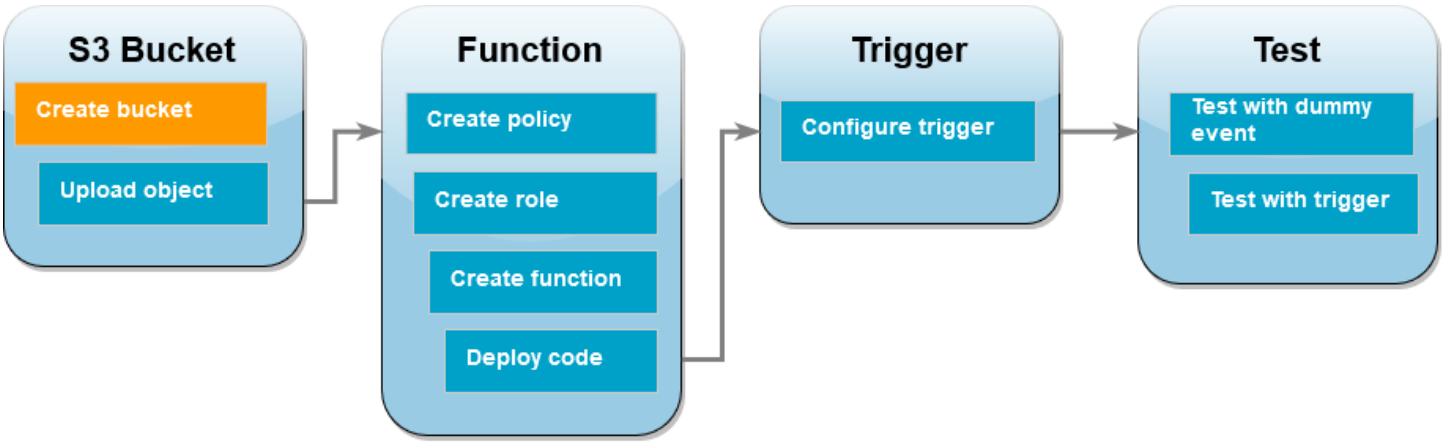
추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

Amazon S3 버킷 생성



Amazon S3 버킷을 생성하려면

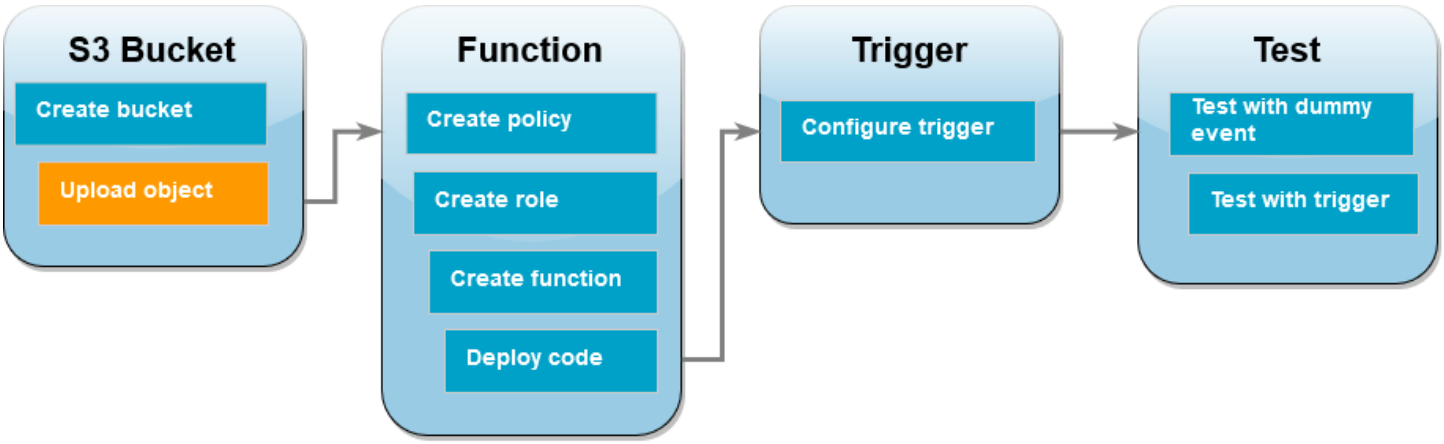
1. Amazon S3 콘솔을 열고 버킷 페이지를 선택합니다.
2. 버킷 생성을 선택합니다.
3. [일반 구성(General configuration)]에서 다음을 수행합니다.

   a. 버킷 이름에 Amazon S3 버킷 이름 지정 규칙을 충족하는 전역적으로 고유한 이름을 입력합니다. 버킷 이름은 소문자, 숫자, 점(.) 및 하이픈(-)만 포함할 수 있습니다.

b. AWS 리전에서 리전을 선택합니다. 자습서 뒷부분에서 동일한 리전에 Lambda 함수를 생성해야 합니다.

4. 다른 모든 옵션을 기본값으로 두고 버킷 생성을 선택합니다.

버킷에 테스트 객체 업로드

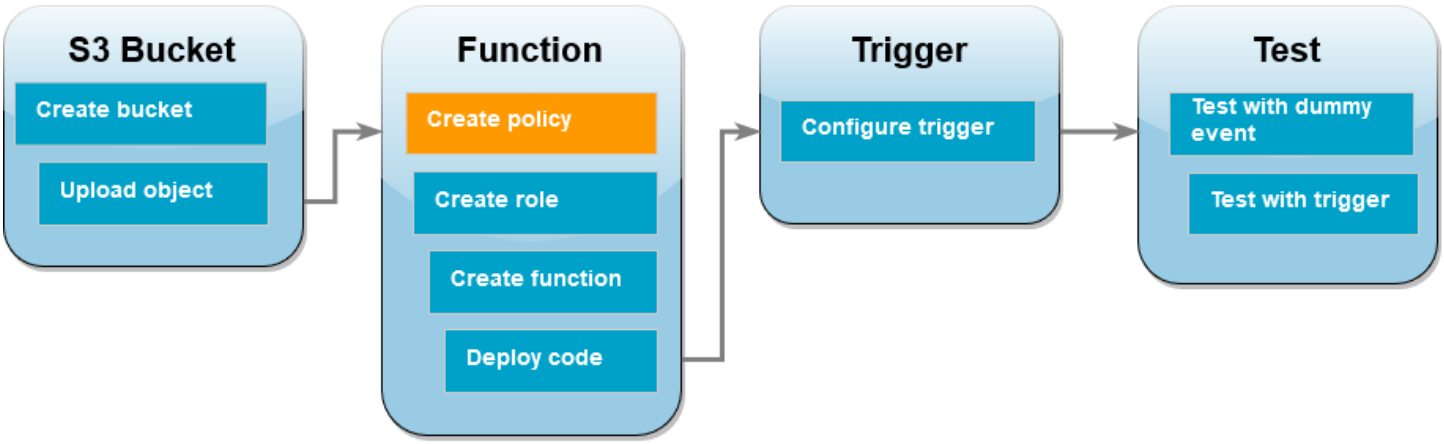


테스트 객체 업로드

1. Amazon S3 콘솔의 버킷 페이지를 열고 이전 단계 중 생성한 버킷을 선택합니다.
2. 업로드를 선택합니다.
3. 파일 추가를 선택하고 업로드하려는 객체를 선택합니다. 모든 파일을 선택할 수 있습니다(예: `HappyFace.jpg`).
4. 열기를 선택한 후 업로드를 선택합니다.

자습서의 뒷부분에서 이 객체를 사용하여 Lambda 함수를 테스트합니다.

권한 정책 생성



Lambda가 Amazon S3 버킷에서 객체를 가져오고 Amazon CloudWatch Logs에 쓸 수 있도록 허용하는 권한 정책을 생성합니다.

정책 생성

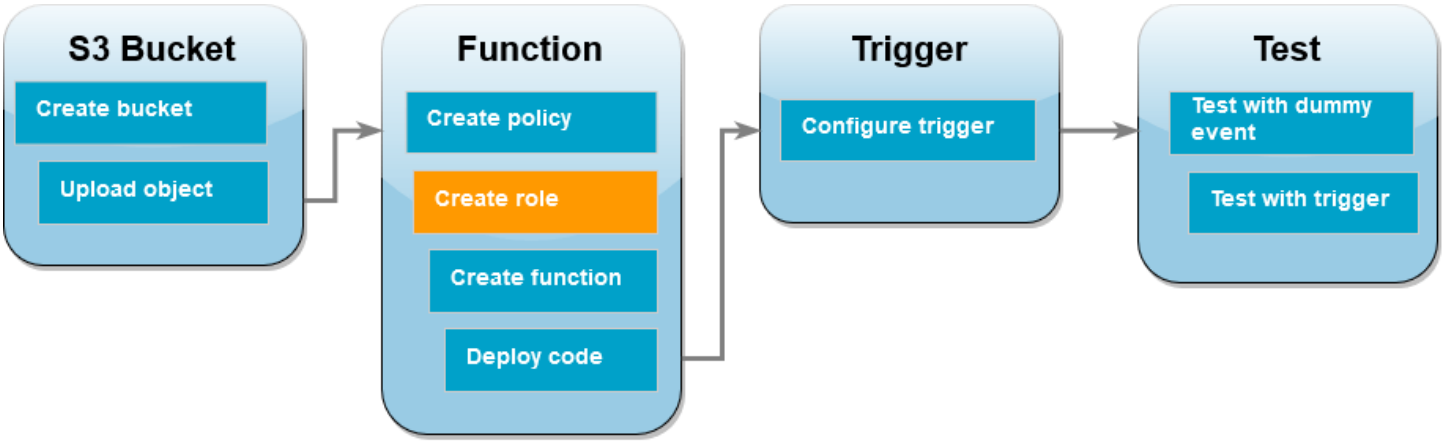
1. IAM 콘솔에서 정책 페이지를 엽니다.
2. 정책 생성(Create Policy)을 선택합니다.
3. JSON 탭에서 다음과 같은 사용자 지정 정책을 JSON 편집기에 붙여 넣습니다.

```
{
    "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. 다음: 태그를 선택합니다.
5. 다음: 검토(Next: Review)를 선택합니다.
6. 정책 검토의 이름에 **s3-trigger-tutorial**를 입력합니다.
7. 정책 생성을 선택합니다.

실행 역할 만들기

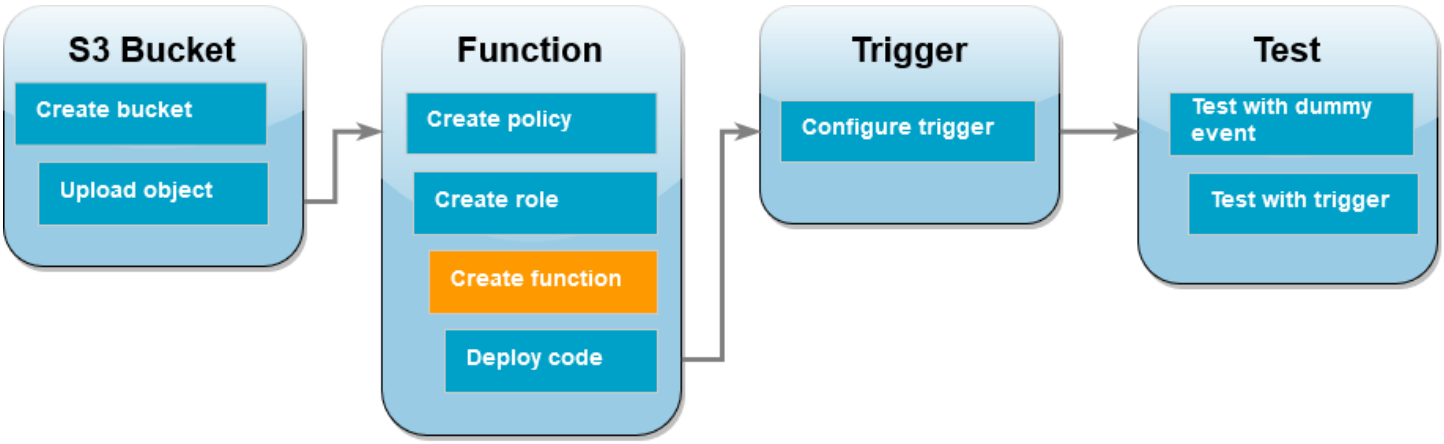


실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 AWS Identity and Access Management(IAM) 역할입니다. 이 단계에서는 이전 단계에서 생성한 권한 정책을 사용하여 실행 역할을 생성하세요.

실행 역할을 생성하고 사용자 지정 권한 정책을 연결하려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티의 유형으로 AWS 서비스를 선택한 다음 사용 사례로 Lambda를 선택합니다.
4. Next(다음)를 선택합니다.
5. 정책 검색 상자에 **s3-trigger-tutorial**를 입력합니다.
6. 검색 결과에서 생성한 정책(`s3-trigger-tutorial`)을 선택한 후, 다음(Next)을 선택합니다.
7. Role details(역할 세부 정보)에서 Role name(역할 이름)에 **lambda-s3-trigger-role**을 입력한 다음 Create role(역할 생성)을 선택합니다.

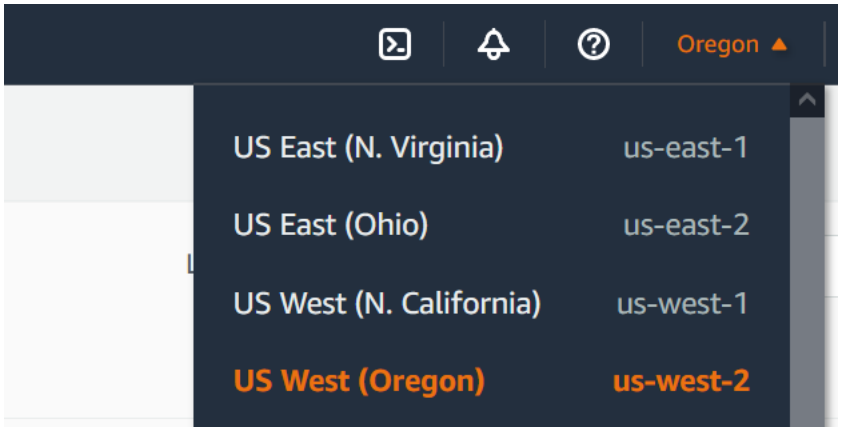
Lambda 함수 생성



Python 3.12 런타임을 사용하여 콘솔에서 Lambda 함수를 생성하세요.

Lambda 함수를 생성하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. Amazon S3 버킷을 생성한 동일한 AWS 리전에서 작업 중인지 확인합니다. 화면 상단의 드롭다운 목록을 사용하여 리전을 변경할 수 있습니다.



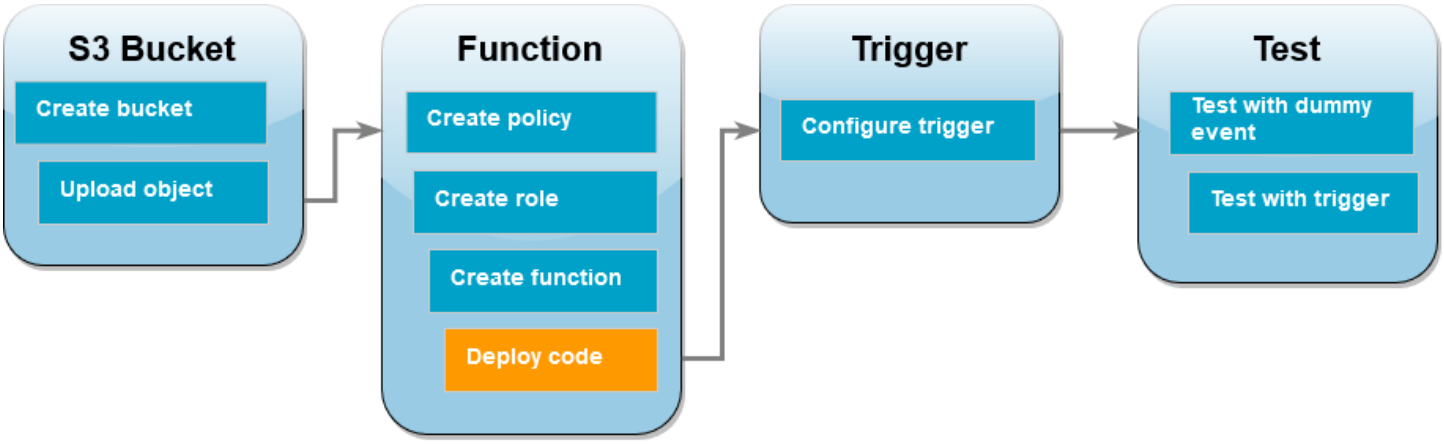
3. 함수 생성을 선택합니다.
4. 새로 작성을 선택합니다.
5. 기본 정보에서 다음과 같이 합니다.
   a. 함수 이름에 `s3-trigger-tutorial`을 입력합니다.
   b. 런타임에서 Python 3.12를 선택합니다.
   c. 아키텍처에서는 x86_64를 선택합니다.
6. 기본 실행 역할 변경 탭에서 다음을 수행합니다.

a. 탭을 확장한 다음 기존 역할 사용을 선택합니다.

b. 이전에 생성한 `lambda-s3-trigger-role`을 선택합니다.

7. 함수 생성을 선택합니다.

함수 코드 배포



이 자습서에서는 Python 3.12 런타임을 사용하지만 다른 런타임의 예제 코드 파일도 제공했습니다. 다음 상자에서 탭을 선택하여 관심 있는 런타임에 대한 코드를 볼 수 있습니다.

Lambda 함수는 Amazon S3에서 수신한 `event` 파라미터에서 업로드된 객체의 키 이름과 버킷 이름을 검색합니다. 그런 다음 함수는 AWS SDK for Python (Boto3)의 get_object 메서드를 사용하여 업로드된 객체의 콘텐츠 유형(MIME 유형)을 비롯한 객체의 메타데이터를 검색합니다.

함수 코드 배포

1. 다음 상자에서 Python 탭을 선택하고 코드를 복사합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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);
    }
};
```

TypeScript를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 S3 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 S3 이벤트 사용.

```
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

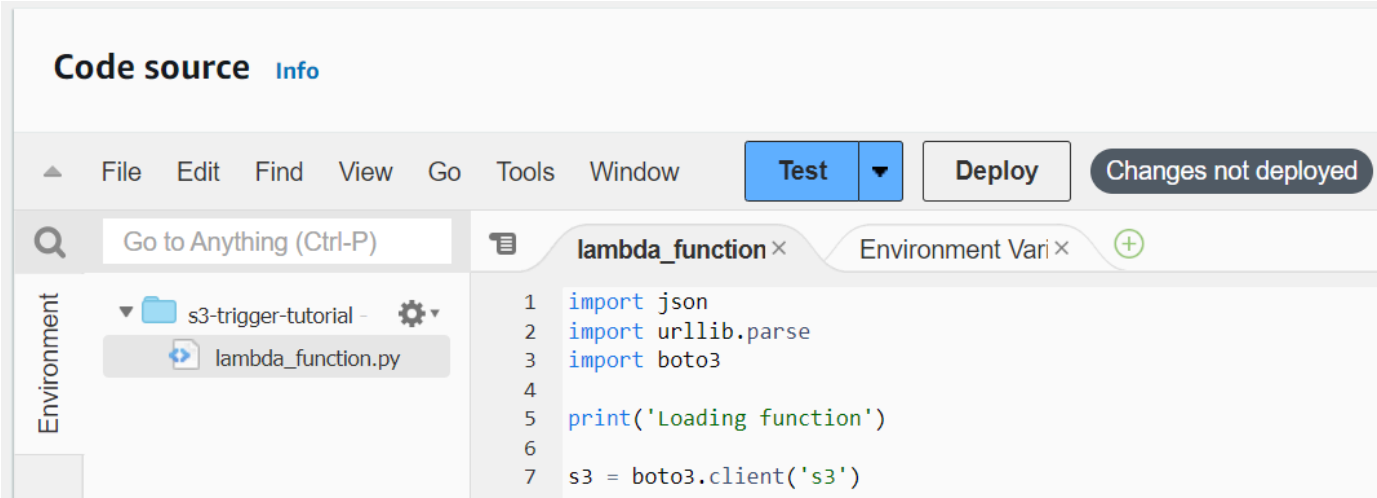
> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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,
```

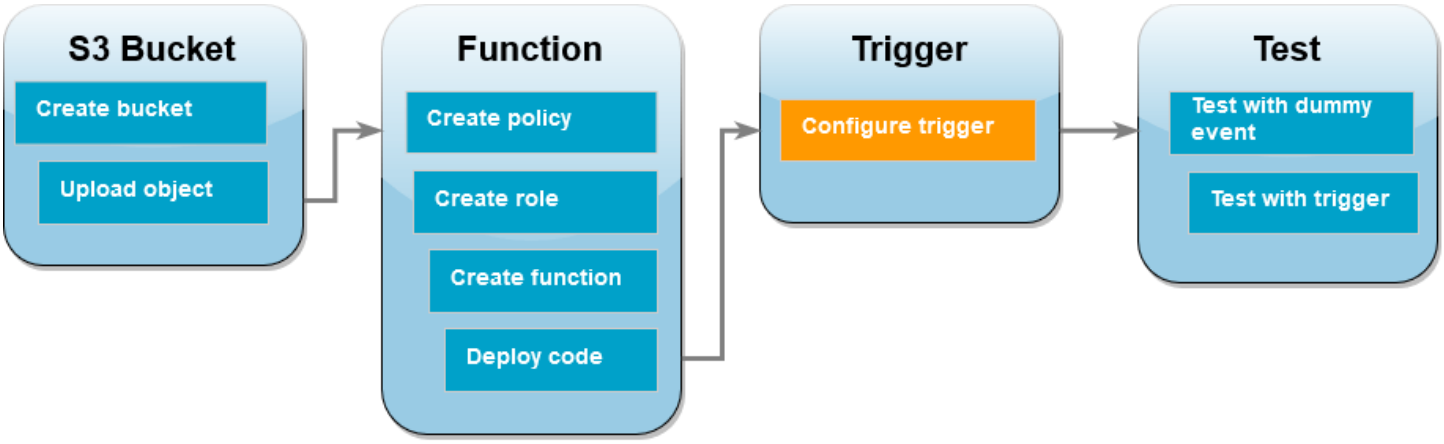
```
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    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 콘솔의 코드 소스 창에서 lambda_function.py 파일에 코드를 붙여넣습니다.



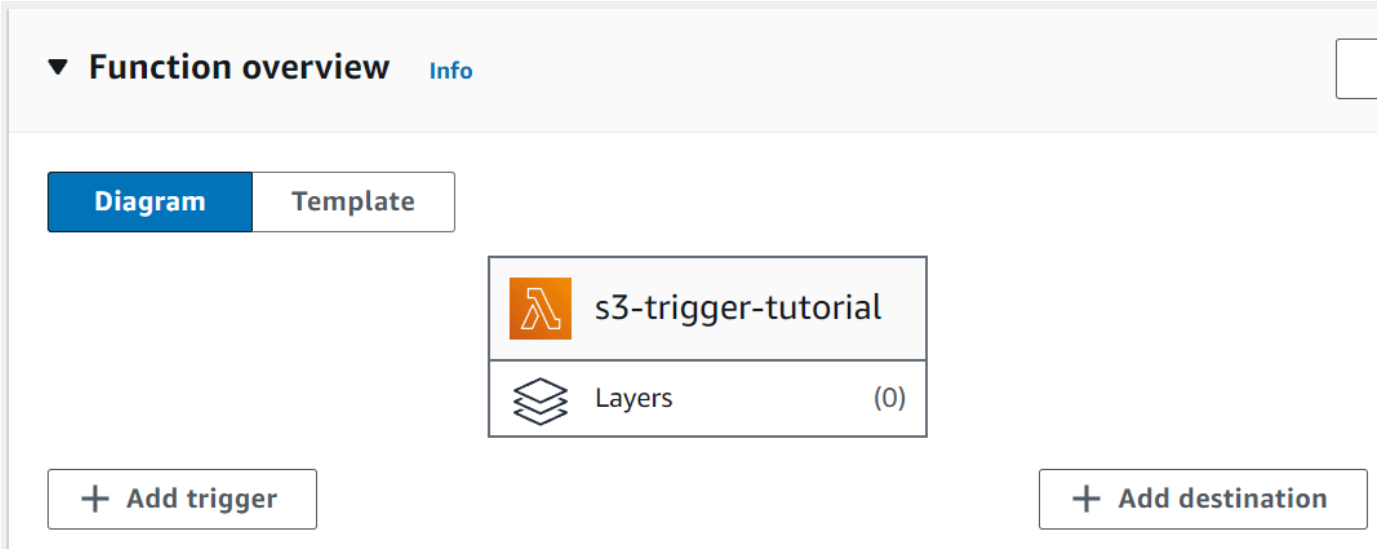
3. [배포]를 선택합니다.

Amazon S3 트리거를 생성합니다



Amazon S3 트리거 생성 방법

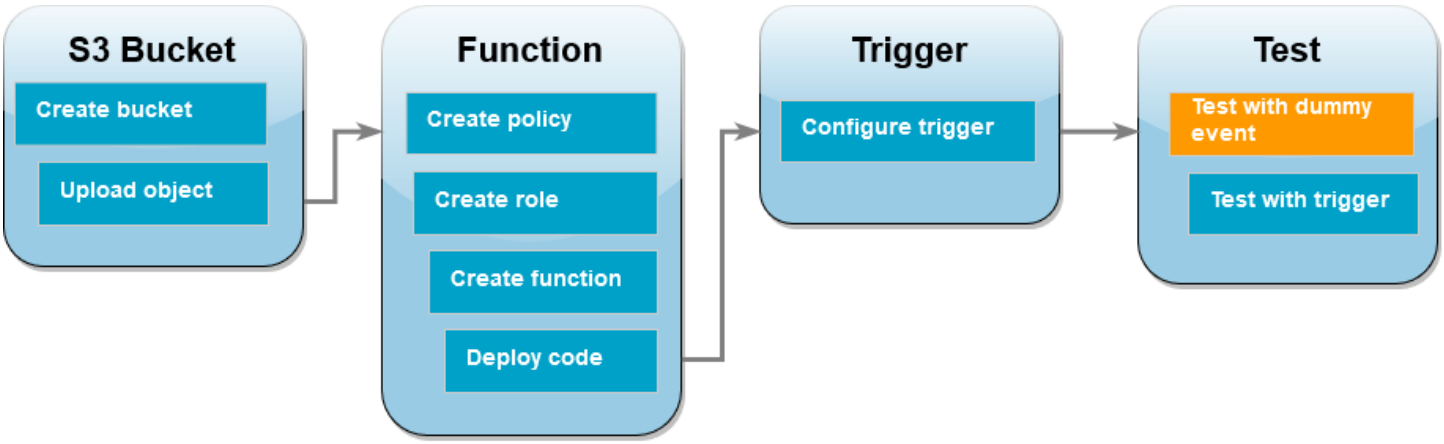
1. 함수 개요 창에서 트리거 추가를 선택합니다.



2. S3를 선택합니다.
3. 버킷에서 자습서 앞부분에서 생성한 버킷을 선택합니다.
4. 이벤트 유형에서 모든 객체 생성 이벤트를 선택합니다.
5. 재귀 호출에서 확인란을 선택하여 입력 및 출력에 동일한 Amazon S3 버킷 사용이 권장되지 않음을 확인합니다.
6. 추가를 선택합니다.

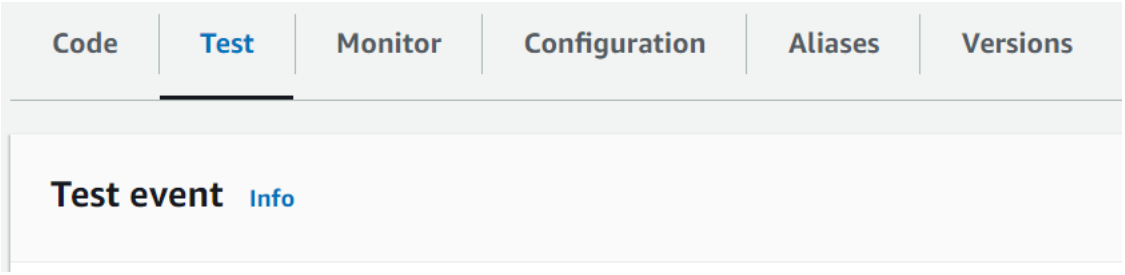
> **Note**
>
> Lambda 콘솔을 사용하여 Lambda 함수에 대한 Amazon S3 트리거를 생성하면 Amazon S3는 사용자가 지정한 버킷에 대한 이벤트 알림을 구성합니다. Amazon S3는 이 이벤트 알림을 구성하기 전에 일련의 검사를 수행하여 이벤트 대상이 존재하고 필수 IAM 정책을 갖추고 있는지 확인합니다. Amazon S3는 해당 버킷에 대해 구성된 다른 모든 이벤트 알림에 대해서도 이러한 테스트를 수행합니다.
>
> 이 검사 때문에 버킷이 이전에 더 이상 존재하지 않는 리소스나 필요한 권한 정책이 없는 리소스에 대한 이벤트 대상을 구성한 경우 Amazon S3는 새 이벤트 알림을 생성할 수 없습니다. 트리거를 생성할 수 없음을 나타내는 다음 오류 메시지가 나타납니다.
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> 이전에 동일한 버킷을 사용하여 다른 Lambda 함수에 대한 트리거를 구성한 후 함수를 삭제하거나 해당 권한 정책을 수정한 경우 이 오류가 나타날 수 있습니다.

더미 이벤트로 Lambda 함수 테스트



더미 이벤트로 Lambda 함수 테스트

1. 함수의 Lambda 콘솔 페이지에서 테스트 탭을 선택합니다.



2. 이벤트 이름에 `MyTestEvent`를 입력합니다.
3. 이벤트 JSON에서 다음 테스트 이벤트를 붙여넣습니다. 해당 값을 바꿉니다.

   - `us-east-1`을 Amazon S3 버킷을 생성한 리전으로 바꿉니다.
   - `DOC-EXAMPLE-BUCKET`의 두 인스턴스를 모두 자체 Amazon S3 버킷의 이름으로 바꿉니다.
   - `test%2FKey`를 이전에 버킷에 업로드한 테스트 객체의 이름(예: `HappyFace.jpg`)으로 바꿉니다.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
```

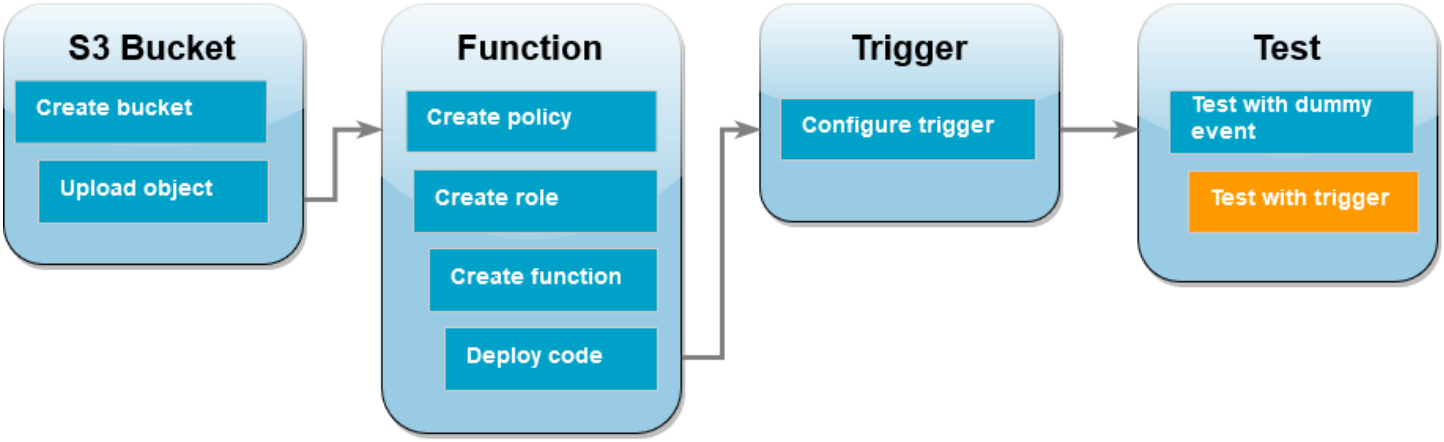
```
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "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(저장)를 선택합니다.
5. 테스트를 선택합니다.
6. 함수가 성공적으로 실행되면 실행 결과 탭에 다음과 비슷한 출력이 표시됩니다.

```
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 트리거로 Lambda 함수 테스트



구성된 트리거로 함수를 테스트하려면 콘솔을 사용하여 Amazon S3 버킷에 객체를 업로드합니다. Lambda 함수가 예상대로 실행되었는지 확인하려면 CloudWatch 로그를 사용하여 함수의 출력을 확인합니다.

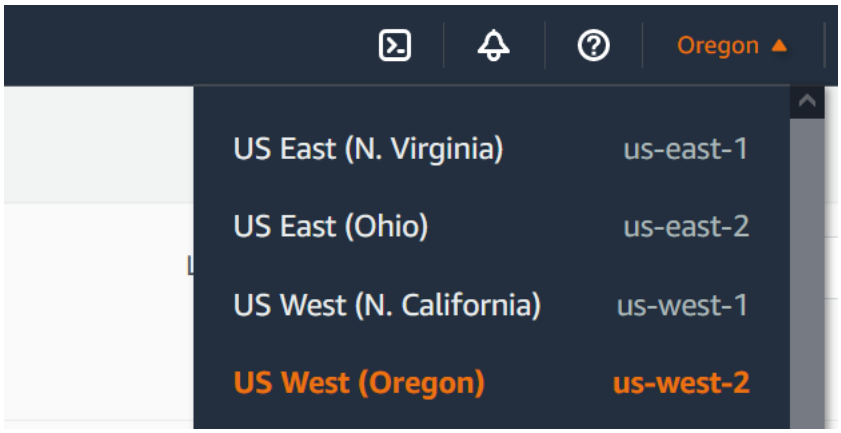
Amazon S3 버킷에 객체 업로드

1. Amazon S3 콘솔의 버킷 페이지를 열고 앞서 생성한 버킷을 선택합니다.
2. 업로드를 선택합니다.
3. 파일 추가를 선택하고 파일 선택기를 사용하여 업로드할 객체를 선택합니다. 이 객체는 사용자가 선택한 임의의 파일이 될 수 있습니다.
4. 열기를 선택한 후 업로드를 선택합니다.

CloudWatch Logs를 사용하여 함수 간접 호출을 확인하려면 다음을 수행합니다.

1. CloudWatch 콘솔을 엽니다.

2. Lambda 함수를 생성한 동일한 AWS 리전에서 작업 중인지 확인합니다. 화면 상단의 드롭다운 목록을 사용하여 리전을 변경할 수 있습니다.



3. 로그를 선택한 후 로그 그룹을 선택합니다.
4. 함수에 대한 로그 그룹(`/aws/lambda/s3-trigger-tutorial`)을 선택합니다.
5. 로그 스트림에서 가장 최근의 로그 스트림을 선택합니다.
6. Amazon S3 트리거에 대한 응답으로 함수가 제대로 간접적으로 호출된 경우 다음과 유사한 출력이 표시됩니다. 표시되는 `CONTENT TYPE`은 버킷에 업로드한 파일 유형에 따라 달라집니다.

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**를 입력하고 Delete(삭제)를 선택합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.

2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

S3 버킷을 삭제하려면

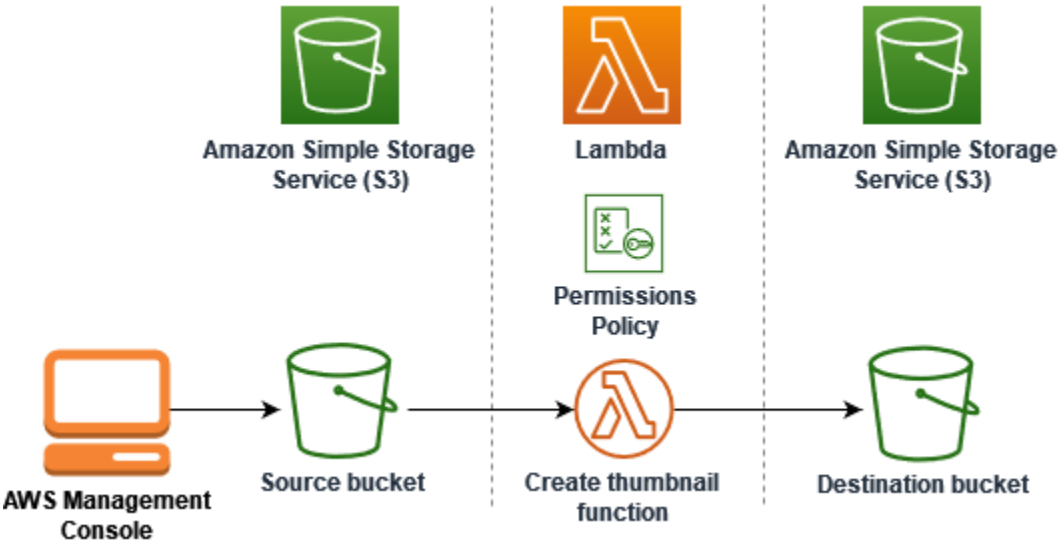
1. Amazon S3 콘솔을 엽니다.
2. 생성한 버킷을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 버킷 이름을 입력합니다.
5. 버킷 삭제(Delete bucket)를 선택합니다.

다음 단계

자습서: Amazon S3 트리거를 사용하여 썸네일 이미지 생성에서는 Amazon S3 트리거가 함수를 간접적으로 호출하여 버킷에 업로드된 각 이미지 파일에 대해 썸네일 이미지를 생성합니다. 이 자습서를 진행하려면 중급 수준의 AWS 및 Lambda 분야 지식이 필요합니다. AWS Command Line Interface(AWS CLI)를 사용하여 리소스를 생성하는 방법과 함수 및 해당 종속성에 대한 .zip 파일 아카이브 배포 패키지를 생성하는 방법을 설명합니다.

자습서: Amazon S3 트리거를 사용하여 썸네일 이미지 생성

이 자습서에서는 Amazon Simple Storage Service(Amazon S3) 버킷에 추가된 이미지의 크기를 조정하는 Lambda 함수를 생성하고 구성합니다. 버킷에 이미지 파일을 추가하면 Amazon S3가 Lambda 함수를 간접적으로 호출합니다. 그런 다음 함수는 이미지의 썸네일 버전을 생성하고 이를 다른 Amazon S3 버킷으로 출력합니다.



이 자습서를 완료하려면 다음 단계를 수행하세요.

1. 원본 및 대상 Amazon S3 버킷을 생성하고 샘플 이미지를 업로드합니다.
2. 이미지 크기를 조정하고 Amazon S3 버킷에 썸네일을 출력하는 Lambda 함수를 생성합니다.
3. 객체가 원본 버킷에 업로드될 때 함수를 간접적으로 호출하는 Lambda 트리거를 구성합니다.
4. 먼저 더미 이벤트를 사용한 다음 원본 버킷에 이미지를 업로드하여 함수를 테스트하세요.

이 단계를 완료하면 Lambda를 사용하여 Amazon S3 버킷에 추가된 객체에 대한 파일 처리 작업을 수행하는 방법을 배우게 됩니다. AWS Command Line Interface 또는 AWS CLI(AWS Management Console)를 사용하여 이 자습서를 완료할 수 있습니다.

Lambda를 위한 Amazon S3 트리거를 구성하는 방법을 배울 수 있는 더 간단한 예제를 찾고 있다면 자습서: Amazon S3 트리거를 사용하여 Lambda 함수 호출을 살펴보세요.

주제

- 필수 조건
- 두 개의 Amazon S3 버킷 생성
- 원본 버킷에 테스트 이미지 업로드
- 권한 정책 생성
- 실행 역할 만들기
- 함수 배포 패키지 생성
- Lambda 함수 생성

- 함수를 간접적으로 호출하도록 Amazon S3 구성
- 더미 이벤트로 Lambda 함수 테스트
- Amazon S3 트리거를 사용하여 함수 테스트
- 리소스 정리

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.

2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.

2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.

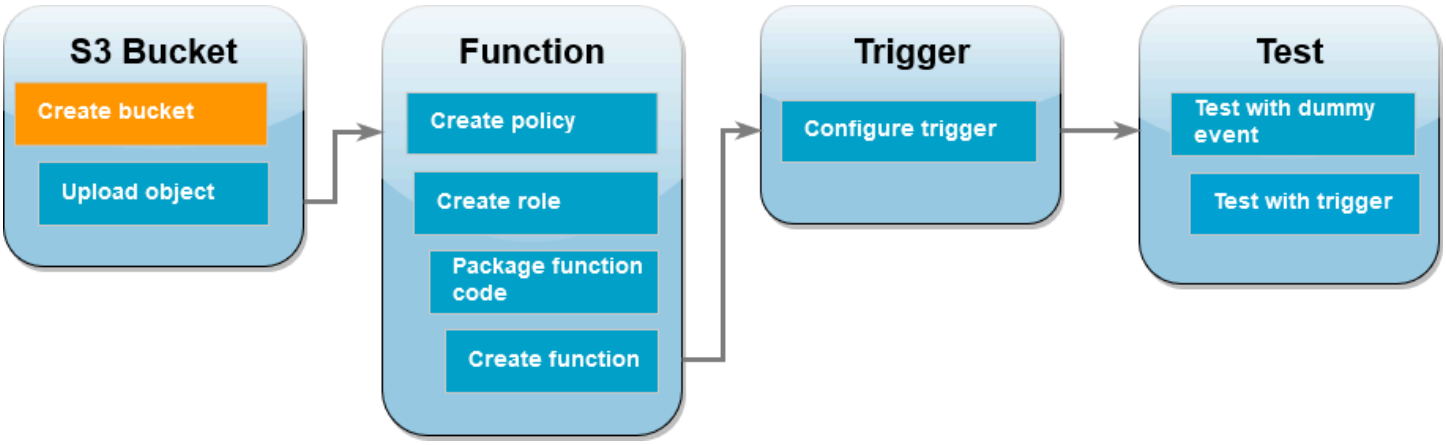
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

AWS CLI를 사용하여 튜토리얼을 완료하려면 AWS Command Line Interface의 최신 버전을 설치하세요.

Lambda 함수 코드의 경우 Python 또는 Node.js를 사용할 수 있습니다. 사용하려는 언어의 언어 지원 도구 및 패키지 관리자를 설치합니다.

두 개의 Amazon S3 버킷 생성



먼저 두 개의 Amazon S3 버킷을 생성합니다. 첫 번째 버킷은 이미지를 업로드할 원본 버킷입니다. 두 번째 버킷은 Lambda에서 함수를 간접적으로 호출할 때 크기 조정된 의 썸네일을 저장하는 데 사용됩니다.

AWS Management Console

Amazon S3 버킷을 생성하려면(콘솔)

1. Amazon S3 콘솔의 버킷 페이지를 엽니다.
2. 버킷 생성을 선택합니다.
3. [일반 구성(General configuration)]에서 다음을 수행합니다.
   a. 버킷 이름에 Amazon S3 버킷 이름 지정 규칙을 충족하는 전역적으로 고유한 이름을 입력합니다. 버킷 이름은 소문자, 숫자, 점(.) 및 하이픈(-)만 포함할 수 있습니다.
   b. AWS 리전의 경우 지리적 위치와 가장 가까운 AWS 리전을 선택하세요. 자습서의 뒷부분에서 동일한 AWS 리전에 Lambda 함수를 생성해야 하므로 선택한 리전을 기록해 둡니다.
4. 다른 모든 옵션을 기본값으로 두고 버킷 생성을 선택합니다.
5. 1~4단계를 반복하여 대상 버킷을 생성합니다. 버킷 이름에 **`DOC-EXAMPLE-SOURCE-BUCKET-resized`**를 입력합니다. 여기서 **`DOC-EXAMPLE-SOURCE-BUCKET`**은 방금 생성한 원본 버킷의 이름입니다.

AWS CLI

Amazon S3 버킷을 생성하려면(AWS CLI)

1. 다음 CLI 명령을 실행하여 원본 버킷을 생성합니다. 버킷에 대해 선택하는 이름은 전역적으로 고유해야 하며 Amazon S3 버킷 이름 지정 규칙을 따라야 합니다. 이름에는 소문자, 숫자, 점(.), 하이픈(-)만 사용할 수 있습니다. `region` 및 `LocationConstraint`의 경우 지리적 위치와 가장 가까운 AWS 리전을 선택하세요.

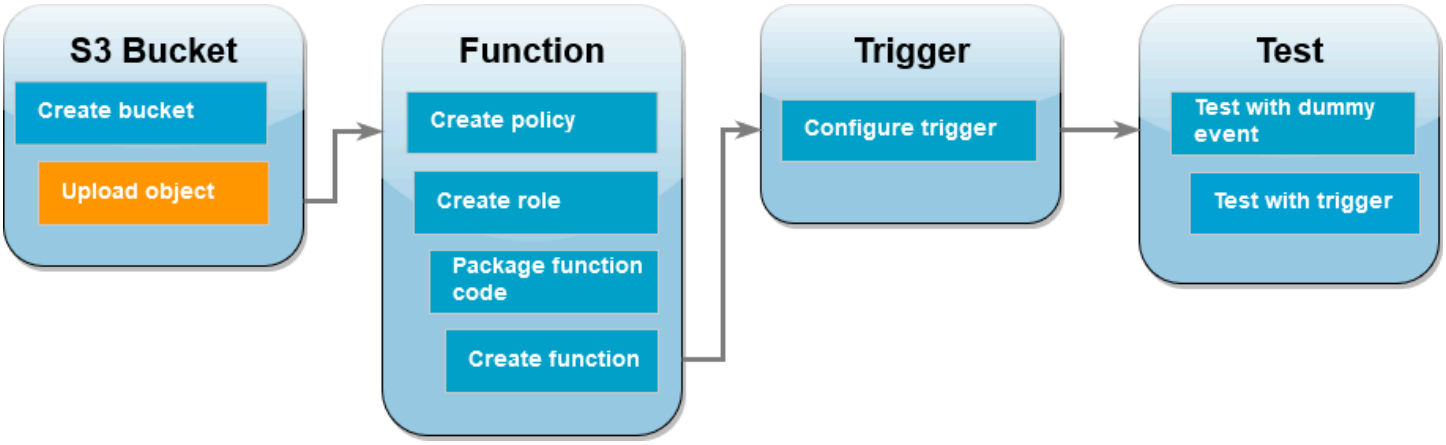
   ```
   aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET --region us-west-2 \
   --create-bucket-configuration LocationConstraint=us-west-2
   ```

   자습서의 뒷부분에서 원본 버킷과 동일한 AWS 리전에 Lambda 함수를 생성해야 하므로 선택한 리전을 기록해 둡니다.

2. 다음 명령을 실행하여 대상 버킷을 생성합니다. 버킷 이름에는 **DOC-EXAMPLE-SOURCE-BUCKET-resized**를 사용해야 합니다. 여기서 **DOC-EXAMPLE-SOURCE-BUCKET**은 1단계에서 생성한 원본 버킷의 이름입니다. `region` 및 `LocationConstraint`의 경우 원본 버킷을 만들 때 사용한 것과 동일한 AWS 리전을 선택합니다.

   ```
   aws s3api create-bucket --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized --region us-
   west-2 \
   --create-bucket-configuration LocationConstraint=us-west-2
   ```

원본 버킷에 테스트 이미지 업로드



자습서의 뒷부분에서 AWS CLI 또는 Lambda 콘솔을 사용해 Lambda 함수를 호출하여 테스트합니다. 함수가 올바르게 작동하는지 확인하려면 원본 버킷에 테스트 이미지가 포함되어 있어야 합니다. 이 이미지는 선택한 JPG 또는 PNG 파일일 수 있습니다.

AWS Management Console

원본 버킷에 테스트 이미지를 업로드하려면(콘솔)

1. Amazon S3 콘솔의 버킷 페이지를 엽니다.
2. 이전 단계에서 생성한 소스 버킷을 선택합니다.
3. 업로드를 선택합니다.
4. 파일 추가를 선택하고 파일 선택기를 사용하여 업로드할 객체를 선택합니다.
5. 열기를 선택한 후 업로드를 선택합니다.

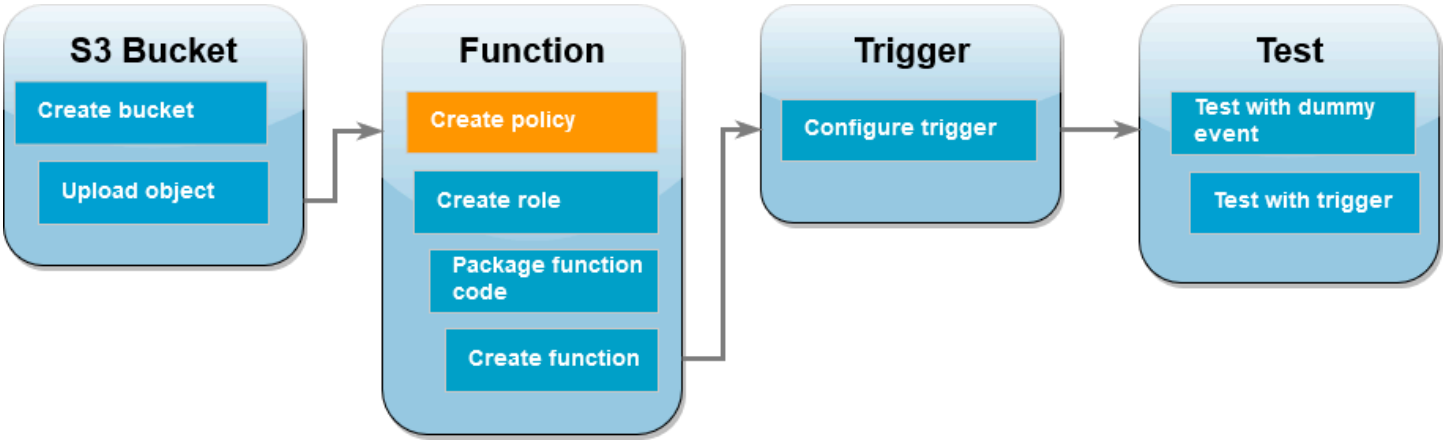
AWS CLI

원본 버킷에 테스트 이미지를 업로드하려면(AWS CLI)

- 업로드하려는 이미지가 포함된 디렉터리에서 다음 CLI 명령을 실행합니다. `--bucket` 파라미터를 원본 버킷의 이름으로 바꿉니다. `--key` 및 `--body` 파라미터에는 테스트 이미지의 파일 이름을 사용합니다.

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key HappyFace.jpg --
body ./HappyFace.jpg
```

권한 정책 생성



Lambda 함수를 생성하는 첫 번째 단계는 권한 정책의 생성입니다. 이 정책은 함수에 다른 AWS 리소스에 액세스하는 데 필요한 권한을 부여합니다. 이 자습서에서 정책을 통해 Amazon S3 버킷에 대한 읽기 및 쓰기 권한을 Lambda에 부여하고 Amazon CloudWatch Logs에 쓸 수 있습니다.

AWS Management Console

정책을 생성하려면(콘솔)

1. AWS Identity and Access Management(IAM) 콘솔의 정책 페이지를 엽니다.
2. 정책 생성을 선택합니다.
3. JSON 탭에서 다음과 같은 사용자 지정 정책을 JSON 편집기에 붙여 넣습니다.

```
{
    "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. 다음을 선택합니다.
5. 정책 세부 정보의 정책 이름에 **LambdaS3Policy**를 입력합니다.
6. 정책 생성을 선택합니다.

AWS CLI

정책을 생성하려면(AWS CLI)

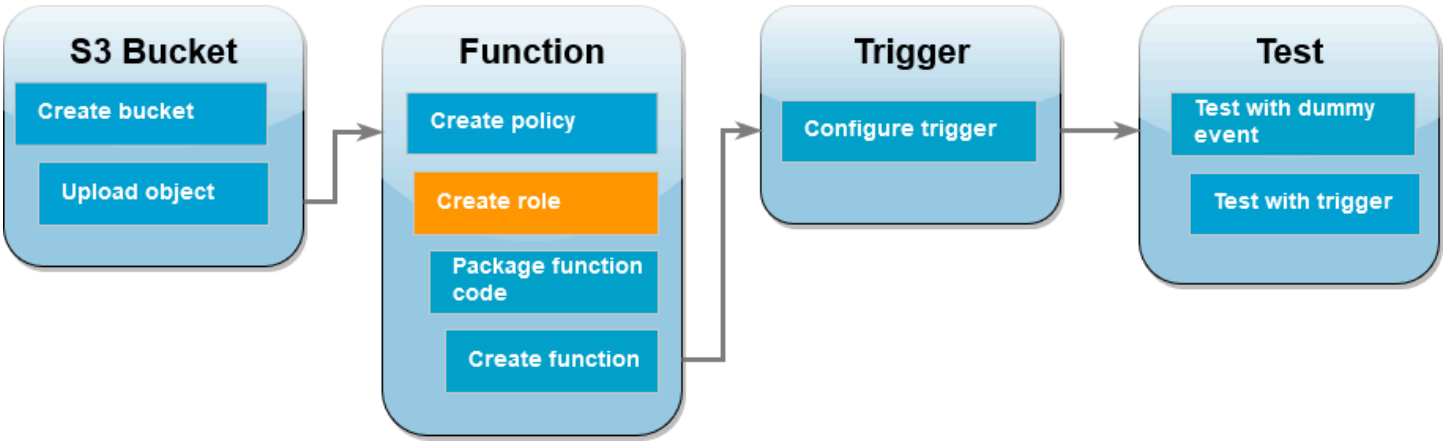
1. 다음 JSON을 policy.json이라는 파일로 저장합니다.

```
{
    "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 정책 문서를 저장한 디렉터리에서 다음 CLI 명령을 실행합니다.

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

실행 역할 만들기



실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 IAM 역할입니다. 함수에 Amazon S3 버킷에 대한 읽기 및 쓰기 액세스 권한을 부여하려면 이전 단계에서 생성한 권한 정책을 연결합니다.

AWS Management Console

실행 역할을 생성하고 권한 정책을 연결하려면(콘솔)

1. (IAM) 콘솔의 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형으로 AWS 서비스를 선택하고 사용 사례로 Lambda를 선택합니다.
4. 다음을 선택합니다.
5. 다음을 수행하여 이전 단계에서 생성한 권한 정책을 추가합니다.
   a. 정책 검색 상자에 **`LambdaS3Policy`**를 입력합니다.
   b. 검색 결과에서 `LambdaS3Policy`에 대한 확인란을 선택합니다.
   c. 다음을 선택합니다.
6. 역할 세부 정보의 역할 이름에 **`LambdaS3Role`**을 입력합니다.
7. 역할 생성을 선택합니다.

AWS CLI

실행 역할을 생성하고 권한 정책을 연결하려면(AWS CLI)

1. 다음 JSON을 `trust-policy.json`이라는 파일로 저장합니다. 이 신뢰 정책을 통해 Lambda는 서비스 보안 주체에 `lambda.amazonaws.com` 권한을 제공하여 AWS Security Token Service(AWS STS) `AssumeRole` 작업을 호출하기 위해 역할의 권한을 사용할 수 있습니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

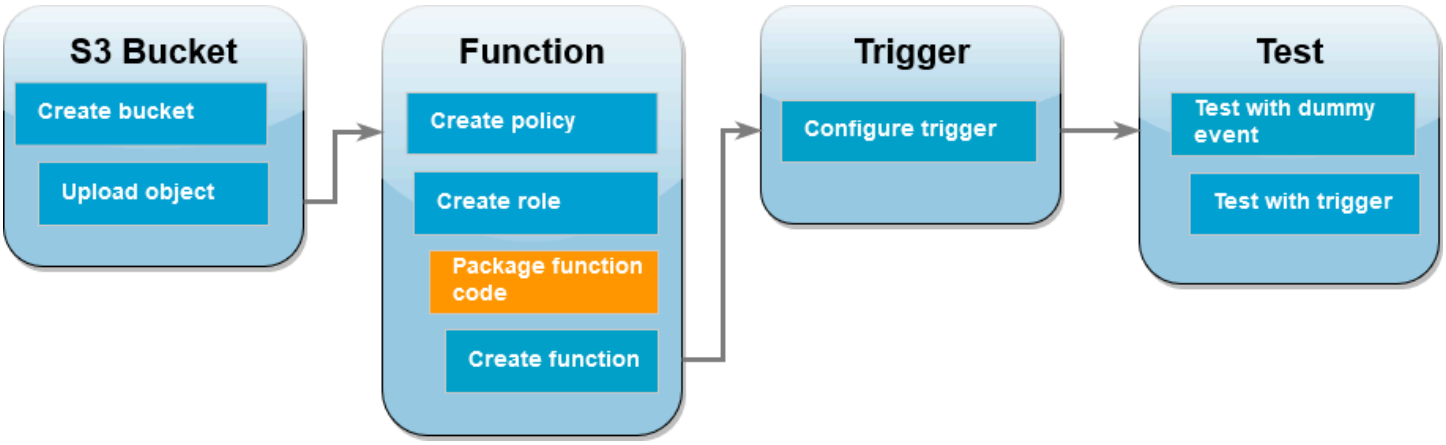
2. JSON 신뢰 정책 문서를 저장한 디렉터리에서 다음 CLI 명령을 실행하여 실행 역할을 생성합니다.

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 이전 단계에서 생성한 권한 정책을 연결하려면 다음 CLI 명령을 실행합니다. 정책의 ARN에 있는 AWS 계정 번호를 자신의 계정 번호로 바꿉니다.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

함수 배포 패키지 생성



함수를 생성하려면 함수 코드와 해당 종속 항목을 포함하는 배포 패키지를 생성합니다. 이 `CreateThumbnail` 함수의 경우 함수 코드는 이미지 크기 조정을 위해 별도의 라이브러리를 사용합니다. 선택한 언어의 지침에 따라 필요한 라이브러리가 포함된 배포 패키지를 생성합니다.

Node.js

배포 패키지를 생성하려면(Node.js)

1. 함수 코드 및 종속 항목에 대해 `lambda-s3`라는 디렉터리를 생성하고 해당 디렉터리로 이동합니다.

   ```
   mkdir lambda-s3
   cd lambda-s3
   ```

2. 다음 함수 코드를 `index.mjs`라는 파일에 저장합니다. `'us-west-2'`를 소스 및 대상 버킷을 직접 만든 AWS 리전으로 바꿔야 합니다.

   ```
   // 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. lambda-s3 디렉터리에서 npm을 사용하여 sharp 라이브러리를 설치합니다. 최신 버전의 sharp(0.33)는 Lambda와 호환되지 않습니다. 이 자습서를 완료하려면 버전 0.32.6을 설치합니다.

```
npm install sharp@0.32.6
```

npm install 명령은 모듈을 위한 node_modules 디렉터리를 생성합니다. 이 단계 후에 디렉터리 구조는 다음과 같아야 합니다.

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

4. 함수 코드와 해당 종속 항목을 포함하는 .zip 배포 패키지를 생성합니다. MacOS 및 Linux에서 다음 명령을 실행합니다.

```
zip -r function.zip .
```

Windows에서 선호하는 zip 유틸리티를 사용하여 .zip 파일을 생성합니다. index.mjs, package.json, package-lock.json 파일과 node_modules 디렉터리가 모두 .zip 파일의 루트에 있는지 확인합니다.

Python

배포 패키지를 생성하려면(Python)

1. 예제 코드를 lambda_function.py라는 파일로 저장합니다.

```
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 파일을 생성한 디렉터리에서 package라는 새 디렉터리를 생성하고 Pillow(PIL) 라이브러리와 AWS SDK for Python (Boto3)을 설치합니다. Lambda Python 런타임에는 Boto3 SDK 버전이 포함되어 있지만 런타임에 포함된 경우에도 함수의 모든 종속 항목을 배포 패키지에 추가하는 것이 좋습니다. 자세한 내용은 Python의 런타임 종속 항목을 참조하세요.

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.9 \
--only-binary=:all: --upgrade \
pillow boto3
```

Pillow 라이브러리에는 C/C++ 코드가 포함되어 있습니다. pip는 --platform manylinux_2014_x86_64 및 --only-binary=:all: 옵션을 사용하여 Amazon Linux 2 운영 체제와 호환되는 미리 컴파일된 바이너리가 포함된 Pillow 버전을 다운로드하고 설치합니다. 이렇게 하면 로컬 빌드 시스템의 운영 체제 및 아키텍처에 관계없이 배포 패키지가 Lambda 실행 환경에서 작동합니다.

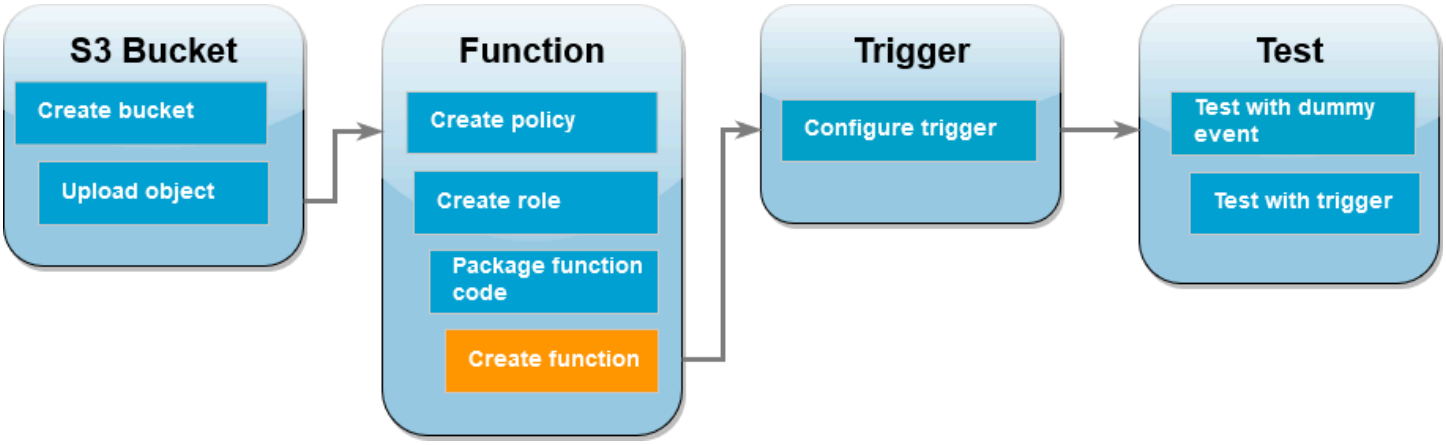
3. 애플리케이션 코드와 Pillow 및 Boto3 라이브러리가 포함된 .zip 파일을 생성합니다. Linux 또는 MacOS에서 명령줄 인터페이스에서 다음 명령을 실행합니다.

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

Windows에서는 선호하는 zip 도구를 사용하여 `lambda_function.zip` 파일을 생성합니다. `lambda_function.py` 파일과 종속 항목이 포함된 폴더가 모두 .zip 파일의 루트에 있는지 확인합니다.

Python 가상 환경을 사용하여 배포 패키지를 생성할 수도 있습니다. Python Lambda 함수에 대한 .zip 파일 아카이브 작업 섹션 참조

Lambda 함수 생성



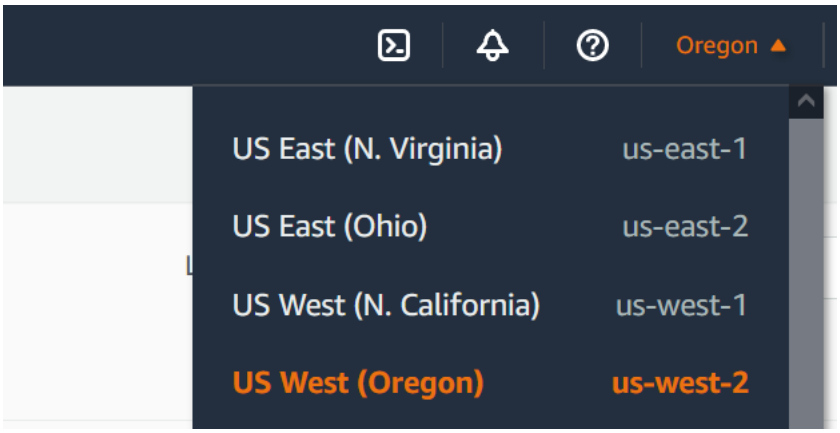
AWS CLI 또는 Lambda 콘솔을 사용하여 Lambda 함수를 생성할 수 있습니다. 선택한 언어의 지침에 따라 함수를 생성합니다.

AWS Management Console

Lambda 함수 생성(콘솔)

콘솔을 사용하여 Lambda 함수를 생성하려면 먼저 일부 'Hello world' 코드가 포함된 기본 함수를 생성합니다. 그런 다음 이전 단계에서 만든 .zip 또는 JAR 파일을 업로드하여 이 코드를 고유한 함수 코드로 바꿉니다.

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. Amazon S3 버킷을 생성한 동일한 AWS 리전에서 작업 중인지 확인합니다. 화면 상단의 드롭다운 목록을 사용하여 리전을 변경할 수 있습니다.



3. 함수 생성을 선택합니다.
4. 새로 작성을 선택합니다.
5. 기본 정보에서 다음과 같이 합니다.
   a. [함수 이름(Function name)]에 **`CreateThumbnail`**을 입력합니다.
   b. 런타임에서 선택한 함수 언어에 따라 Node.js 18.x 또는 Python 3.9를 선택합니다.
   c. 아키텍처에서는 x86_64를 선택합니다.
6. 기본 실행 역할 변경 탭에서 다음을 수행합니다.
   a. 탭을 확장한 다음 기존 역할 사용을 선택합니다.
   b. 이전에 생성한 `LambdaS3Role`을 선택합니다.
7. 함수 생성을 선택합니다.

함수 코드를 업로드하려면(콘솔)

1. 코드 소스 창에서 에서 업로드를 선택합니다.
2. .zip 파일을 선택합니다.
3. 업로드를 선택합니다.
4. 파일 선택기에서 .zip 파일을 선택하고 열기를 선택합니다.
5. Save(저장)를 선택합니다.

AWS CLI

함수를 생성하려면(AWS CLI)

- 선택한 언어의 CLI 명령을 실행합니다. `role` 파라미터의 경우 `123456789012`를 자신의 AWS 계정 ID로 바꿉니다. `region` 파라미터의 경우 `us-west-2`를 Amazon S3 버킷을 생성한 리전으로 바꿉니다.

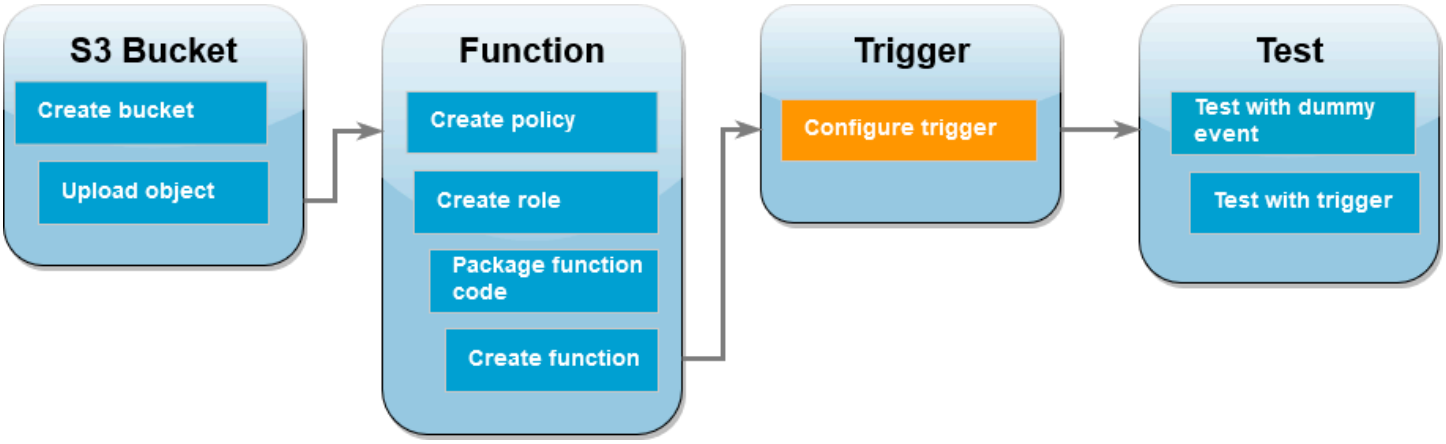
  - Node.js의 경우 `function.zip` 파일이 포함된 디렉터리에서 다음 명령을 실행합니다.

```
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
```

  - Python의 경우 `lambda_function.zip` 파일이 포함된 디렉터리에서 다음 명령을 실행합니다.

```
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
```

함수를 간접적으로 호출하도록 Amazon S3 구성



원본 버킷에 이미지를 업로드할 때 Lambda 함수를 실행하려면 함수에 대한 트리거를 구성해야 합니다. 콘솔 또는 AWS CLI를 사용하여 Amazon S3 트리거를 구성할 수 있습니다.

> **Important**
>
> 이 절차는 버킷에서 객체가 생성될 때마다 함수를 간접적으로 호출하도록 Amazon S3 버킷을 구성합니다. 원본 버킷에서만 이를 구성해야 합니다. Lambda 함수가 함수를 간접적으로 호출하는 동일한 버킷에 객체를 생성하는 경우 함수를 루프에서 연속적으로 간접 호출할 수 있습니다. 이로 인해 예상치 못한 요금이 AWS 계정에 청구될 수 있습니다.

AWS Management Console

Amazon S3 트리거를 구성하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수(`CreateThumbnail`)를 선택합니다.
2. 트리거 추가를 선택합니다.
3. S3를 선택합니다.
4. 버킷에서 원본 버킷을 선택합니다.
5. 이벤트 유형에서 모든 객체 생성 이벤트를 선택합니다.
6. 재귀 호출에서 확인란을 선택하여 입력 및 출력에 동일한 Amazon S3 버킷 사용이 권장되지 않음을 확인합니다. Serverless Land의 Recursive patterns that cause run-away Lambda functions를 읽고 Lambda의 재귀 호출 패턴에 대해 자세히 알아볼 수 있습니다.
7. 추가를 선택합니다.

   Lambda 콘솔을 사용하여 트리거를 생성하면 Lambda는 선택한 서비스에 함수 간접 호출 권한을 부여하는 리소스 기반 정책을 자동으로 생성합니다.

AWS CLI

Amazon S3 트리거를 구성하려면(AWS CLI)

1. 이미지 파일을 추가할 때 Amazon S3 소스 버킷이 함수를 간접적으로 호출하려면 먼저 리소스 기반 정책을 사용하여 함수에 대한 권한을 구성해야 합니다. 리소스 기반 정책 설명은 함수를 간접적으로 호출할 수 있는 다른 AWS 서비스 권한을 부여합니다. Amazon S3에 함수 간접 호출 권한을 부여하려면 다음 CLI 명령을 실행합니다. `source-account` 파라미터를 자신의 AWS 계정 ID로 바꾸고 고유한 원본 버킷 이름을 사용해야 합니다.

```
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
```

이 명령으로 정의하는 정책은 원본 버킷에서 작업이 발생할 때만 Amazon S3가 함수를 간접적으로 호출하도록 허용합니다.

> **Note**
>
> Amazon S3 버킷 이름은 전역적으로 고유하지만 리소스 기반 정책을 사용할 때는 버킷이 반드시 계정에 속하도록 지정하는 것이 가장 좋습니다. 버킷을 삭제하면 다른 AWS 계정가 동일한 Amazon 리소스 이름(ARN)으로 버킷을 생성할 수 있기 때문입니다.

2. 다음 JSON을 `notification.json`이라는 파일로 저장합니다. 이 JSON을 원본 버킷에 적용하면 JSON이 새 객체가 추가될 때마다 Lambda 함수에 알림을 보내도록 버킷을 구성합니다. Lambda 함수 ARN의 AWS 계정 번호와 AWS 리전을 자신의 계정 번호와 리전으로 바꿉니다.

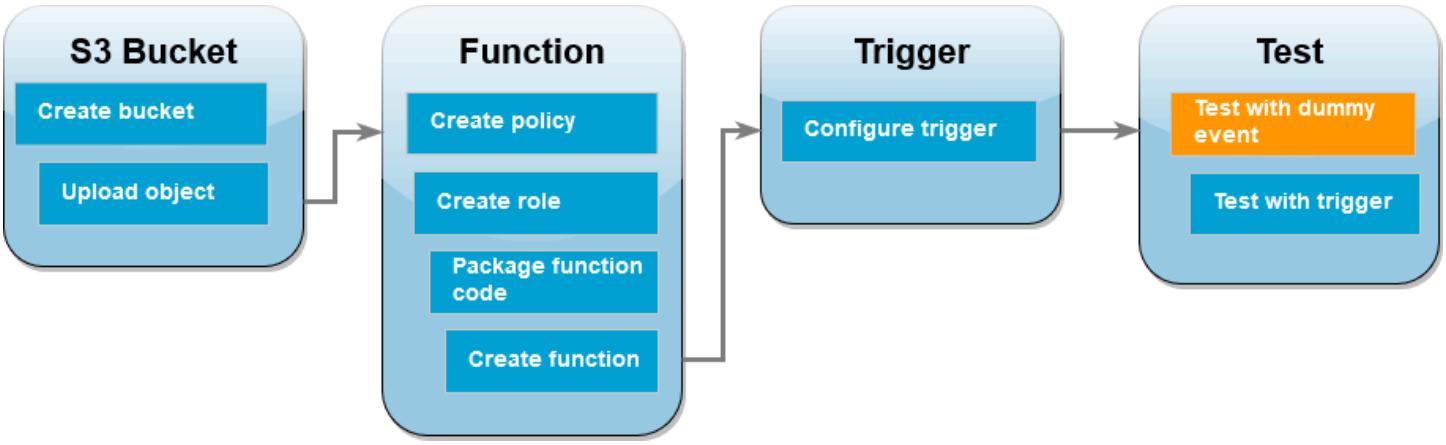
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 다음 CLI 명령을 실행하여 원본 버킷에 생성한 JSON 파일의 알림 설정을 적용합니다. `DOC-EXAMPLE-SOURCE-BUCKET`을 원본 버킷의 이름으로 바꿉니다.

```
aws s3api put-bucket-notification-configuration --bucket DOC-EXAMPLE-SOURCE-
BUCKET \
--notification-configuration file://notification.json
```

`put-bucket-notification-configuration` 명령 및 `notification-configuration` 옵션에 대한 자세한 내용은 AWS CLI 명령 참조의 put-bucket-notification-configuration을 참조하세요.

더미 이벤트로 Lambda 함수 테스트



Amazon S3 원본 버킷에 이미지 파일을 추가하여 전체 설정을 테스트하기 전에 더미 이벤트로 호출하여 Lambda 함수가 올바르게 작동하는지 테스트합니다. Lambda의 이벤트는 함수가 처리할 데이터가 포함된 JSON 형식의 문서입니다. 함수가 Amazon S3에 의해 간접적으로 호출되면 함수로 전송된 이벤트에는 버킷 이름, 버킷 ARN, 객체 키와 같은 정보가 포함됩니다.

AWS Management Console

더미 이벤트로 Lambda 함수를 테스트하려면(콘솔)

1. Lambda 콘솔의 함수 페이지를 열고 함수(`CreateThumbnail`)를 선택합니다.
2. 테스트 탭을 선택합니다.
3. 테스트 이벤트를 생성하려면 테스트 이벤트 창에서 다음을 수행합니다.
   a. 테스트 이벤트 작업에서 새 이벤트 생성을 선택합니다.
   b. 이벤트 이름에 **`myTestEvent`**를 입력합니다.
   c. 템플릿에서 S3 Put을 선택합니다.
   d. 다음 파라미터의 값을 사용자 고유의 값으로 바꿉니다.
      - `awsRegion`의 경우 `us-east-1`을 Amazon S3 버킷을 생성한 AWS 리전으로 바꿉니다.
      - `name`의 경우 `DOC-EXAMPLE-BUCKET`을 사용자 고유의 Amazon S3 원본 버킷 이름으로 바꿉니다.
      - `key`의 경우 `test%2Fkey`를 원본 버킷에 테스트 이미지 업로드 단계에서 원본 버킷에 업로드한 테스트 객체의 파일 이름으로 바꿉니다.

```
{
  "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(저장)를 선택합니다.

4. 테스트 이벤트 창에서 테스트를 선택합니다.
5. 함수가 이미지의 크기 조정된 버전을 생성하여 대상 Amazon S3 버킷에 저장했는지 확인하려면 다음을 수행합니다.

   a. Amazon S3 콘솔의 버킷 페이지를 엽니다.

   b. 대상 버킷을 선택하고 크기 조정된 파일이 객체 창에 나열되는지 확인합니다.

AWS CLI

더미 이벤트로 Lambda 함수를 테스트하려면(AWS CLI)

1. 다음 JSON을 dummyS3Event.json이라는 파일로 저장합니다. 다음 파라미터의 값을 고유한 값으로 바꿉니다.

   1. awsRegion의 경우 us-west-2를 Amazon S3 버킷을 생성한 AWS 리전으로 바꿉니다.
   2. name의 경우 DOC-EXAMPLE-SOURCE-BUCKET을 사용자 고유의 Amazon S3 원본 버킷 이름으로 바꿉니다.
   3. key의 경우 HappyFace.jpg를 원본 버킷에 테스트 이미지 업로드 단계에서 원본 버킷에 업로드한 테스트 객체의 파일 이름으로 바꿉니다.

```
{
  "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 파일을 저장한 디렉터리에서 다음 CLI 명령을 실행하여 함수를 간접적으로 호출합니다. 이 명령은 간접 호출 유형 파라미터의 값으로 RequestResponse를 지정하여 Lambda 함수를 동기식으로 간접 호출합니다. 동기 및 비동기 호출에 대한 자세한 내용은 Lambda 함수 호출을 참조하세요.

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

AWS CLI의 버전 2를 사용하는 경우 cli-binary-format 옵션이 필요합니다. 이 설정을 기본 설정으로 지정하려면 `aws configure set cli-binary-format raw-in-base64-out`을(를) 실행하세요. 자세한 내용은 AWS CLI 지원 대상 전역적 명령줄 옵션을 참조하세요.

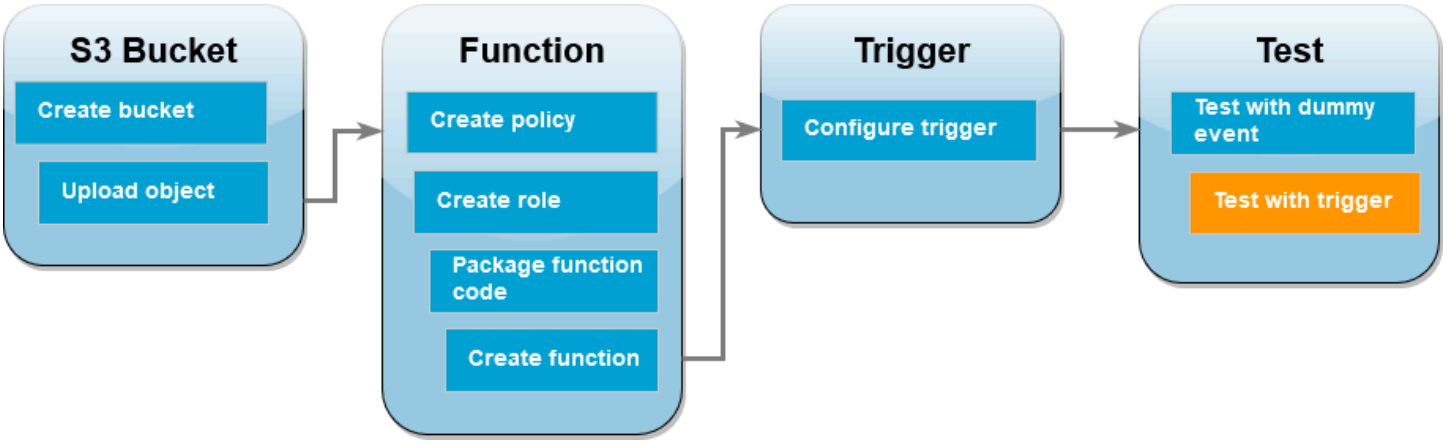
3. 함수가 이미지의 썸네일 버전을 생성하여 대상 Amazon S3 버킷에 저장했는지 확인합니다. 다음 CLI 명령을 실행하여 DOC-EXAMPLE-SOURCE-BUCKET-resized를 대상 버킷 이름으로 바꿉니다.

```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

다음과 유사한 출력 화면이 표시되어야 합니다. Key 파라미터는 크기 조정된 이미지의 파일 이름을 보여줍니다.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Amazon S3 트리거를 사용하여 함수 테스트



Lambda 함수가 올바르게 작동하는지 확인했으니 Amazon S3 원본 버킷에 이미지 파일을 추가하여 전체 설정을 테스트할 준비가 되었습니다. 원본 버킷에 이미지를 추가하면 Lambda 함수가 자동으로 간접적으로 호출됩니다. 함수는 파일의 크기 조정된 버전을 생성하여 대상 버킷에 저장합니다.

AWS Management Console

Amazon S3 트리거를 사용하여 Lambda 함수를 테스트하려면(콘솔)

1. Amazon S3 버킷에 이미지를 업로드하려면 다음을 수행합니다.
   a. Amazon S3 콘솔의 버킷 페이지를 열고 원본 버킷을 선택합니다.
   b. 업로드를 선택합니다.

c. 파일 추가를 선택하고 파일 선택기를 사용하여 업로드하려는 이미지 파일을 선택합니다. 이미지 객체는 .jpg 또는 .png 파일일 수 있습니다.

d. 열기를 선택한 후 업로드를 선택합니다.

2. 다음을 수행하여 Lambda가 이미지 파일의 크기 조정된 버전을 대상 버킷에 저장했는지 확인합니다.

a. Amazon S3 콘솔의 버킷 페이지로 돌아가서 대상 버킷을 선택합니다.

b. 이제 객체 창에 크기 조정된 이미지 파일 두 개가 표시됩니다. 이는 Lambda 함수의 각 테스트에서 가져온 것입니다. 크기 조정된 이미지를 다운로드하려면 파일을 선택한 다음 다운로드를 선택합니다.

AWS CLI

Amazon S3 트리거를 사용하여 Lambda 함수를 테스트하려면(AWS CLI)

1. 업로드하려는 이미지가 포함된 디렉터리에서 다음 CLI 명령을 실행합니다. `--bucket` 파라미터를 원본 버킷의 이름으로 바꿉니다. `--key` 및 `--body` 파라미터에는 테스트 이미지의 파일 이름을 사용합니다. 테스트 이미지는 .jpg 또는 .png 파일일 수 있습니다.

```
aws s3api put-object --bucket DOC-EXAMPLE-SOURCE-BUCKET --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. 함수가 이미지의 썸네일 버전을 생성하여 대상 Amazon S3 버킷에 저장했는지 확인합니다. 다음 CLI 명령을 실행하여 `DOC-EXAMPLE-SOURCE-BUCKET-resized`를 대상 버킷 이름으로 바꿉니다.

```
aws s3api list-objects-v2 --bucket DOC-EXAMPLE-SOURCE-BUCKET-resized
```

함수가 성공적으로 실행되면 다음과 유사한 출력이 표시됩니다. 이제 대상 버킷에는 크기 조정된 파일 두 개가 포함됩니다.

```
{
    "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"
        }
    ]
}
```

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

생성한 정책을 삭제하려면

1. IAM 콘솔에서 정책 페이지를 엽니다.
2. 생성한 정책(AWSLambdaS3Policy)을 선택합니다.
3. [정책 작업(Policy actions)], [삭제(Delete)]를 선택합니다.
4. 삭제를 선택합니다.

실행 역할을 삭제하려면

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

S3 버킷을 삭제하려면

1. Amazon S3 콘솔을 엽니다.
2. 생성한 버킷을 선택합니다.
3. 삭제를 선택합니다.
4. 텍스트 입력 필드에 버킷 이름을 입력합니다.
5. 버킷 삭제(Delete bucket)를 선택합니다.

Amazon S3 Batch Operations에 AWS Lambda 사용

Amazon S3 Batch Operations를 사용하여 대규모 Amazon S3 객체 집합에 대해 Lambda 함수를 호출할 수 있습니다. Amazon S3는 배치 작업의 진행 상황을 추적하고, 알림을 전송하며, 각 작업의 상태를 보여주는 완료 보고서를 저장합니다.

배치 작업을 실행하려면 Amazon S3 Batch Operations 작업을 생성합니다. 작업을 생성할 때 매니페스트(객체 목록)를 제공하고 해당 객체에 대해 수행할 작업을 구성합니다.

배치 작업이 시작되면 Amazon S3에서 매니페스트의 각 객체에 대해 Lambda 함수를 동기적으로 호출합니다. 이벤트 파라미터에는 버킷과 객체의 이름이 포함됩니다.

다음 예제에서는 Amazon S3가 DOC-EXAMPLE-BUCKET 버킷에 있는 customerImage1.jpg라는 객체에 대해 Lambda 함수에 전송하는 이벤트를 보여줍니다.

Example Amazon S3 배치 요청 이벤트

```
{
"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 함수는 다음 예제에 나온 것과 같은 필드를 포함한 JSON 객체를 반환해야 합니다. 이벤트 파라미터에서 `invocationId` 및 `taskId`를 복사할 수 있습니다. `resultString`의 문자열을 반환할 수 있습니다. Amazon S3는 완료 보고서의 `resultString` 값을 저장합니다.

Example Amazon S3 배치 요청 응답

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Amazon S3 Batch Operations에서 Lambda 함수 호출

정규화되거나 정규화되지 않은 함수 ARN을 사용하여 Lambda 함수를 호출할 수 있습니다. 전체 배치 작업에 동일한 함수 버전을 사용하려면 작업을 생성할 때 `FunctionARN` 파라미터에 특정 함수 버전을 구성합니다. 별칭 또는 $LATEST 한정자를 구성하는 경우 작업 실행 중에 별칭 또는 $LATEST가 업데이트되면 배치 작업이 함수의 새 버전을 즉시 호출하기 시작합니다.

기존 Amazon S3 이벤트 기반 함수는 배치 작업에 재사용할 수 없습니다. 이는 Amazon S3 Batch Operations가 Lambda 함수에 다른 이벤트 파라미터를 전달하고 특정 JSON 구조의 반환 메시지를 예상하기 때문입니다.

Amazon S3 Batch Job에 대해 생성하는 리소스 기반 정책에서 Lambda 함수를 호출하는 작업에 대한 권한을 설정해야 합니다.

함수의 실행 역할에서 Amazon S3가 함수를 실행할 때 역할을 수임하기 위한 신뢰 정책을 설정합니다.

함수가 AWS SDK를 사용하여 Amazon S3 리소스를 관리하는 경우 실행 역할에 Amazon S3 권한을 추가해야 합니다.

작업이 실행될 때 Amazon S3는 여러 함수 인스턴스를 시작하여 Amazon S3 객체를 함수의 동시성 한도까지 병렬로 처리합니다. Amazon S3는 소규모 작업에 대한 과도한 비용을 막기 위해 인스턴스의 초기 증가량을 제한합니다.

Lambda 함수가 `TemporaryFailure` 응답 코드를 반환하면 Amazon S3는 작업을 다시 시도합니다.

Amazon S3 Batch Operations에 대한 자세한 내용은 Amazon S3 개발자 안내서의 배치 작업 수행을 참조하세요.

Amazon S3 Batch Operations에서 Lambda 함수를 사용하는 방법에 대한 예는 Amazon S3 개발자 안내서의 Amazon S3 Batch Operations에서 Lambda 함수 호출을 참조하세요.

S3 객체 Lambda를 사용하여 S3 객체 변환

S3 객체 Lambda를 통해 자체 코드를 Amazon S3 GET, HEAD 및 LIST 요청에 추가하여 애플리케이션으로 데이터가 반환되기 전에 데이터를 수정 및 처리할 수 있습니다. 사용자 지정 코드를 사용하여 행을 필터링하고 동적으로 이미지 크기를 조정하며 기밀 데이터를 교정하는 등 표준 S3 GET, HEAD 또는 LIST 요청에서 반환한 데이터를 수정할 수 있습니다. AWS Lambda 함수에서 지원하는 이 기능을 사용하면 코드는 AWS의 완전관리형 인프라에서 실행되므로 데이터의 파생 사본을 생성 및 저장하거나 프록시를 실행하지 않아도 되며, 애플리케이션에 추가 요금이 발생하지 않습니다.

자세한 내용은 S3 객체 Lambda를 사용하여 객체 변환을 참고하세요.

자습서

- Amazon S3 객체 Lambda를 사용하여 애플리케이션의 데이터 변환
- Amazon S3 객체 Lambda 및 Amazon Comprehend를 사용하여 PII 데이터 감지 및 수정
- Amazon S3 객체 Lambda를 사용하여 이미지를 검색할 때 동적으로 워터마크 지정

Secrets Manager에서 AWS Lambda 사용

AWS Lambda 함수는 Secrets Manager API 또는 AWS 소프트웨어 개발 키트(SDK)를 사용하여 AWS Secrets Manager와 상호 작용할 수 있습니다. 또한 AWS 파라미터 및 보안 암호 Lambda 익스텐션을 사용하면 SDK를 사용하지 않고도 Lambda 함수에서 AWS Secrets Manager 보안 암호를 검색하고 캐싱할 수 있습니다. 자세한 내용은 AWS Lambda 함수에서 AWS Secrets Manager 보안 암호 사용을 참조하세요.

Amazon SES에서 AWS Lambda 사용

Amazon SES를 사용하여 메시지를 받으면 메시지가 도착할 때 Lambda 함수를 호출하도록 Amazon SES를 구성할 수 있습니다. 그런 다음 서비스는 수신되는 이메일 이벤트를 전달하여 Lambda 함수를 호출할 수 있습니다. 여기에서 실제 수신되는 이메일 이벤트는 Amazon SES 이벤트의 Amazon SES 메시지인 파라미터로 전달됩니다.

Example Amazon SES 메시지 이벤트

```
{
  "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"
    }
  ]
}
```

자세한 내용은 Amazon SES 개발자 안내서의 Lambda 작업을 참조하세요.

Amazon SNS 알림을 사용하여 Lambda 함수 간접 호출

Lambda 함수를 사용하여 Amazon Simple Notification Service(Amazon SNS) 알림을 처리할 수 있습니다. Amazon SNS는 Lambda 함수를 주제에 전송된 메시지에 대한 대상으로 지원합니다. 동일한 계정 또는 다른 AWS 계정의 주제에 함수를 등록할 수 있습니다. 자세한 내용은 the section called “튜토리얼” 섹션을 참조하세요.

Lambda는 표준 SNS 주제에 대해서만 SNS 트리거를 지원합니다. FIFO 주제는 지원되지 않습니다.

비동기 호출을 위해 Lambda는 메시지를 대기열에 추가하고 재시도를 처리합니다. Amazon SNS가 Lambda에 도달할 수 없거나 메시지가 거부되는 경우, Amazon SNS는 몇 시간 동안 간격을 늘리면서 재시도합니다. 자세한 내용은 Amazon SNS FAQ에서 안정성을 참조하세요.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

주제

- 콘솔을 사용하여 Lambda 함수에 대한 Amazon SNS 주제 트리거 추가
- Lambda 함수에 대한 Amazon SNS 주제 트리거 수동 추가
- 샘플 SNS 이벤트 셰이프
- 자습서: Amazon Simple Notification Service에서 AWS Lambda 사용

콘솔을 사용하여 Lambda 함수에 대한 Amazon SNS 주제 트리거 추가

Lambda 함수에 대한 트리거로 SNS 주제를 추가하는 가장 쉬운 방법은 Lambda 콘솔을 사용하는 것입니다. 콘솔을 통해 트리거를 추가하면 Lambda가 SNS 주제에서 이벤트 수신을 시작하는 데 필요한 권한과 구독을 자동으로 설정합니다.

Lambda 함수에 대한 트리거로 SNS 주제를 추가하려면 다음을 수행하세요(콘솔).

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 트리거를 추가할 함수의 이름을 선택합니다.

3. 구성을 선택하고 트리거를 선택합니다.
4. 트리거 추가를 선택합니다.
5. 트리거 구성 아래의 드롭다운 메뉴에서 SNS를 선택합니다.
6. SNS 주제로 구독할 SNS 주제를 선택합니다.

Lambda 함수에 대한 Amazon SNS 주제 트리거 수동 추가

Lambda 함수에 대한 SNS 트리거를 수동으로 설정하려면 다음 단계를 완료해야 합니다.

- SNS가 함수를 간접적으로 호출할 수 있도록 함수에 대한 리소스 기반 정책을 정의합니다.
- Amazon SNS 주제에 대한 Lambda 함수를 구독합니다.

> **Note**
>
> SNS 주제와 Lambda 함수가 서로 AWS 다른 계정에 있는 경우 SNS 주제에 대한 크로스 계정 구독을 허용하려면 추가 권한도 부여해야 합니다. 자세한 내용은 Amazon SNS 구독을 위한 크로스 계정 권한 부여를 참조하세요.

AWS Command Line Interface(AWS CLI)를 사용하여 이 두 단계를 모두 완료할 수 있습니다. 먼저, SNS 간접 호출을 허용하는 Lambda 함수에 대한 리소스 기반 정책을 정의하려면 다음 AWS CLI 명령을 사용하세요. `--function-name` 값을 Lambda 함수 이름으로 바꾸고 `--source-arn` 값을 SNS 주제 ARN으로 바꿔야 합니다.

```
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
```

SNS 주제의 함수를 구독하려면 다음 AWS CLI 명령을 사용하세요. `--topic-arn` 값을 SNS 주제 ARN으로 바꾸고, `--notification-endpoint` 값을 Lambda 함수 ARN으로 바꿉니다.

```
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
```

샘플 SNS 이벤트 셰이프

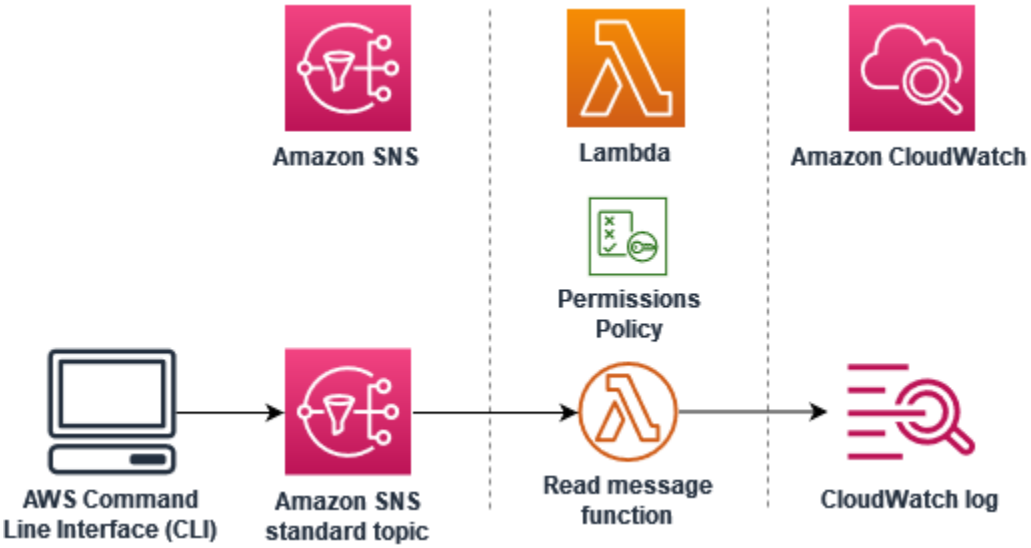
Amazon SNS는 메시지 및 메타데이터를 포함하는 이벤트와 비동기적으로 함수를 호출합니다.

Example Amazon SNS 메시지 이벤트

```
{
  "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"
      }
    }
  ]
}
```

자습서: Amazon Simple Notification Service에서 AWS Lambda 사용

이 자습서에서는 하나의 AWS 계정에서 Lambda 함수를 사용하여 별도의 AWS 계정에서 Amazon Simple Notification Service(SNS) 주제를 구독합니다. Amazon SNS 주제에 메시지를 게시하면 Lambda 함수가 메시지 내용을 읽어 Amazon CloudWatch Logs에 출력합니다. 이 자습서를 완료하려면 AWS Command Line Interface(AWS CLI)를 사용합니다.



이 자습서를 완료하려면 다음 단계를 수행합니다.

- 계정 A에서 Amazon SNS 주제를 생성합니다.
- 계정 B에서 주제에서 메시지를 읽을 Lambda 함수를 생성합니다.
- 계정 B에서 주제에 대한 구독을 생성합니다.
- 계정 A의 Amazon SNS 주제에 메시지를 게시하고 계정 B의 Lambda 함수가 메시지를 CloudWatch Logs로 출력하는지 확인합니다.

이 단계를 완료하면 Lambda 함수를 간접적으로 호출하도록 Amazon SNS 주제를 구성하는 방법을 배우게 됩니다. 또한 다른 AWS 계정의 리소스가 Lambda를 간접적으로 호출할 수 있는 권한을 부여하는 AWS Identity and Access Management(IAM) 정책을 생성하는 방법도 배웁니다.

이 자습서에서는 2개의 개별 AWS 계정을 사용합니다. AWS CLI 명령은 각각 다른 AWS 계정에서 사용하도록 구성된 `accountA`와 `accountB`라는 2개의 명명된 프로파일을 사용하여 이를 설명합니다. 다른 프로파일을 사용하도록 AWS CLI를 구성하는 방법에 대한 자세한 내용은 AWS Command Line Interface - 버전 2 사용 설명서의 구성 및 자격 증명 파일 설정을 참조하세요. 두 프로파일에 대해 동일한 기본 AWS 리전을 구성해야 합니다.

두 AWS 계정에 대해 생성하는 AWS CLI 프로파일이 서로 다른 이름을 사용하거나 기본 프로파일과 하나의 명명된 프로파일을 사용하는 경우 필요에 따라 다음 단계에서 AWS CLI 명령을 수정합니다.

필수 조건

AWS 계정에 등록

AWS 계정이 없는 경우 다음 절차에 따라 계정을 생성합니다.

AWS 계정에 등록하려면

1. https://portal.aws.amazon.com/billing/signup을 여세요.
2. 온라인 지시 사항을 따르세요.

   등록 절차 중에는 전화를 받고 키패드로 인증 코드를 입력하는 과정이 있습니다.

   AWS 계정에 가입하면 AWS 계정 루트 사용자이 생성됩니다. 루트 사용자에게는 계정의 모든 AWS 서비스 및 리소스 액세스 권한이 있습니다. 보안 모범 사례는 사용자에게 관리 액세스 권한을 할당하고, 루트 사용자만 사용하여 루트 사용자 액세스 권한이 필요한 작업을 수행하는 것입니다.

AWS는 가입 절차 완료된 후 사용자에게 확인 이메일을 전송합니다. 언제든지 https://aws.amazon.com/으로 가서 내 계정(My Account)을 선택하여 현재 계정 활동을 보고 계정을 관리할 수 있습니다.

관리자 액세스 권한이 있는 사용자 생성

AWS 계정에 가입하고 AWS 계정 루트 사용자에 보안 조치를 한 다음, AWS IAM Identity Center를 활성화하고 일상적인 작업에 루트 사용자를 사용하지 않도록 관리 사용자를 생성합니다.

귀하의 AWS 계정 루트 사용자 보호

1. 루트 사용자를 선택하고 AWS 계정이메일 주소를 입력하여 AWS Management Console에 계정 소유자로 로그인합니다. 다음 페이지에서 비밀번호를 입력합니다.

   루트 사용자를 사용하여 로그인하는 데 도움이 필요하면 AWS 로그인 사용 설명서의 루트 사용자로 로그인을 참조하세요.
2. 루트 사용자의 다중 인증(MFA)을 활성화합니다.

   지침은 IAM 사용 설명서의 AWS 계정루트 사용자용 가상 MFA 디바이스 활성화(콘솔)를 참조하세요.

관리자 액세스 권한이 있는 사용자 생성

1. IAM Identity Center를 활성화합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 AWS IAM Identity Center 설정을 참조하세요.
2. IAM Identity Center에서 사용자에게 관리 액세스 권한을 부여합니다.

   IAM Identity Center 디렉터리를 ID 소스로 사용하는 방법에 대한 자습서는 AWS IAM Identity Center 사용 설명서의 기본 IAM Identity Center 디렉터리로 사용자 액세스 구성을 참조하세요.

관리 액세스 권한이 있는 사용자로 로그인

- IAM IDentity Center 사용자로 로그인하려면 IAM IDentity Center 사용자를 생성할 때 이메일 주소로 전송된 로그인 URL을 사용합니다.

  IAM Identity Center 사용자로 로그인하는 데 도움이 필요한 경우 AWS 로그인 사용 설명서의 AWS 액세스 포털에 로그인을 참조하세요.

추가 사용자에게 액세스 권한 할당

1. IAM Identity Center에서 최소 권한 적용 모범 사례를 따르는 권한 세트를 생성합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Create a permission set를 참조하세요.
2. 사용자를 그룹에 할당하고, 그룹에 Single Sign-On 액세스 권한을 할당합니다.

   지침은 AWS IAM Identity Center 사용 설명서의 Add groups를 참조하세요.

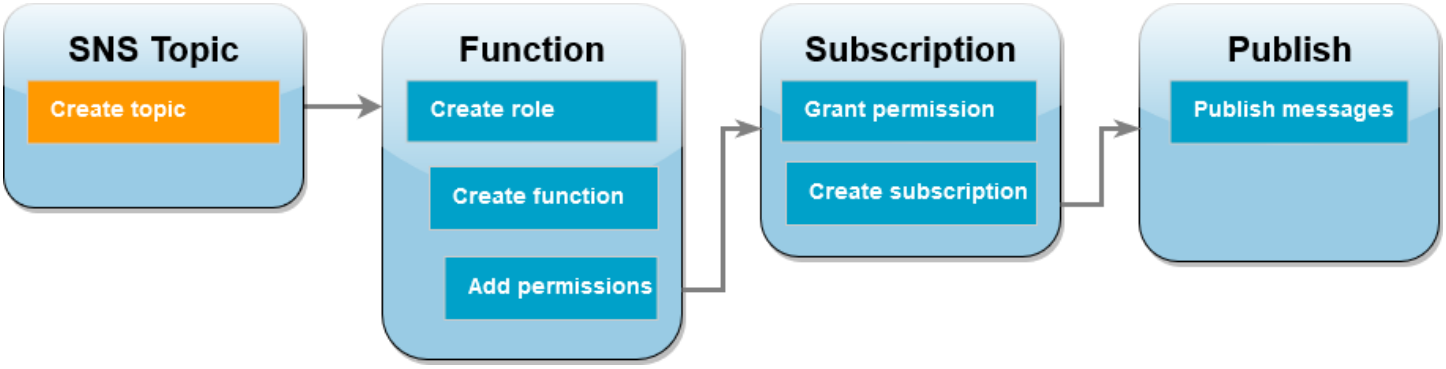
AWS Command Line Interface 설치

아직 AWS Command Line Interface를 설치하지 않은 경우 AWS CLI의 최신 버전 설치 또는 업데이트에서 설명하는 단계에 따라 설치하세요.

이 자습서에서는 명령을 실행할 셸 또는 명령줄 터미널이 필요합니다. Linux 및 macOS에서는 선호하는 셸과 패키지 관리자를 사용합니다.

> **Note**
>
> Windows에서는 Lambda와 함께 일반적으로 사용하는 일부 Bash CLI 명령(예:`zip`)은 운영 체제의 기본 제공 터미널에서 지원되지 않습니다. Ubuntu와 Bash의 Windows 통합 버전을 가져오려면 Linux용 Windows Subsystem을 설치합니다.

Amazon SNS 주제 생성(계정 A)



주제를 생성하려면

- 계정 A에서 다음 AWS CLI 명령을 사용하여 Amazon SNS 표준 주제를 생성합니다.

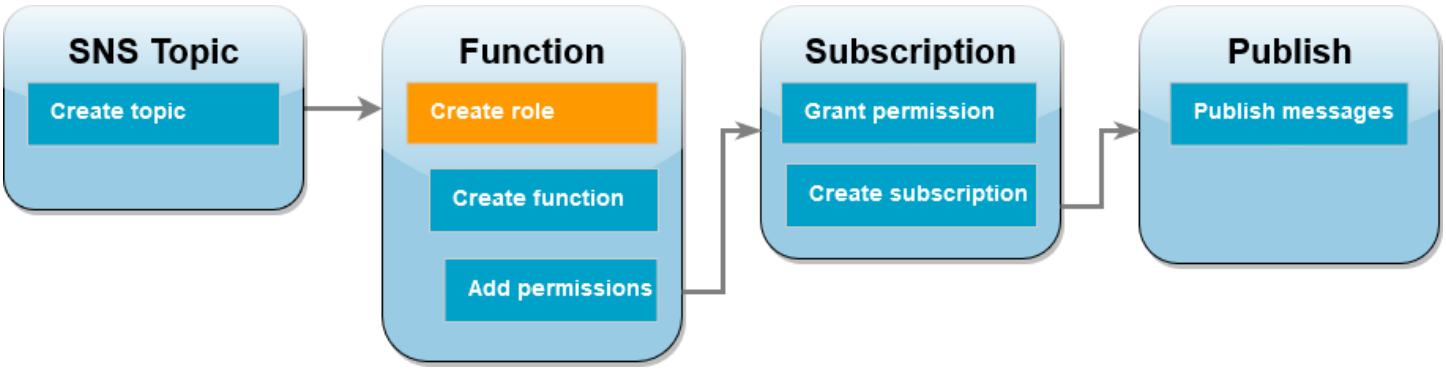
  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

  다음과 유사한 출력 화면이 표시되어야 합니다.

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  }
  ```

  주제의 Amazon 리소스 이름(ARN)을 기록해 둡니다. 이는 나중에 자습서에서 주제를 구독하기 위해 Lambda 함수에 권한을 추가할 때 필요합니다.

함수 실행 역할 생성(계정 B)

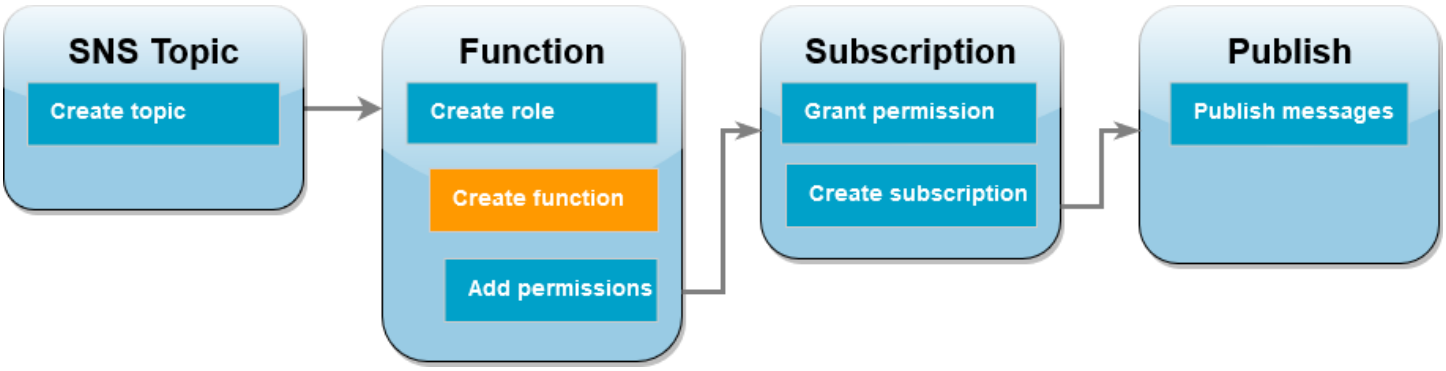


실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 Lambda 함수에 부여하는 IAM 역할입니다. 계정 B에서 함수를 생성하기 전에 CloudWatch Logs에 로그를 쓸 수 있는 기본 권한을 함수에 부여하는 역할을 생성합니다. Amazon SNS 주제에서 읽을 수 있는 권한은 이후 단계에서 추가하겠습니다.

실행 역할을 만들려면

1. 계정 B의 IAM 콘솔에서 역할 페이지를 엽니다.
2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형에서 AWS 서비스를 선택합니다.
4. 사용 사례에서 Lambda를 선택합니다.
5. Next(다음)를 선택합니다.
6. 다음을 수행하여 역할에 기본 권한 정책을 추가합니다.
   a. 권한 정책 검색 상자에 **`AWSLambdaBasicExecutionRole`**을 입력합니다.
   b. Next(다음)를 선택합니다.
7. 다음을 수행하여 역할 생성을 마무리합니다.
   a. 역할 세부 정보의 역할 이름에 **`lambda-sns-role`**을 입력합니다.
   b. 역할 생성을 선택합니다.

Lambda 함수 생성(계정 B)



Amazon SNS 메시지를 처리하는 Lambda 함수를 생성합니다. 함수 코드는 각 레코드의 메시지 콘텐츠를 Amazon CloudWatch Logs에 로그합니다.

이 자습서에서는 Node.js 18.x 런타임을 사용하지만 다른 런타임 언어의 예제 코드도 제공했습니다. 다음 상자에서 탭을 선택하여 관심 있는 런타임에 대한 코드를 볼 수 있습니다. 이 단계에서 사용할 JavaScript 코드는 JavaScript 탭에 표시된 첫 번째 예제입니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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;
  }
}
```

TypeScript를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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
}
```

함수를 만들려면

1. 프로젝트에 대한 디렉터리를 생성하고 해당 디렉터리로 전환합니다.

```
mkdir sns-tutorial
cd sns-tutorial
```

2. 샘플 JavaScript 코드를 새로운 `index.js` 파일에 복사합니다.
3. 다음 `zip` 명령을 사용하여 배포 패키지를 생성합니다.

```
zip function.zip index.js
```

4. 다음 AWS CLI 명령을 실행하여 계정 B에서 Lambda 함수를 생성합니다.

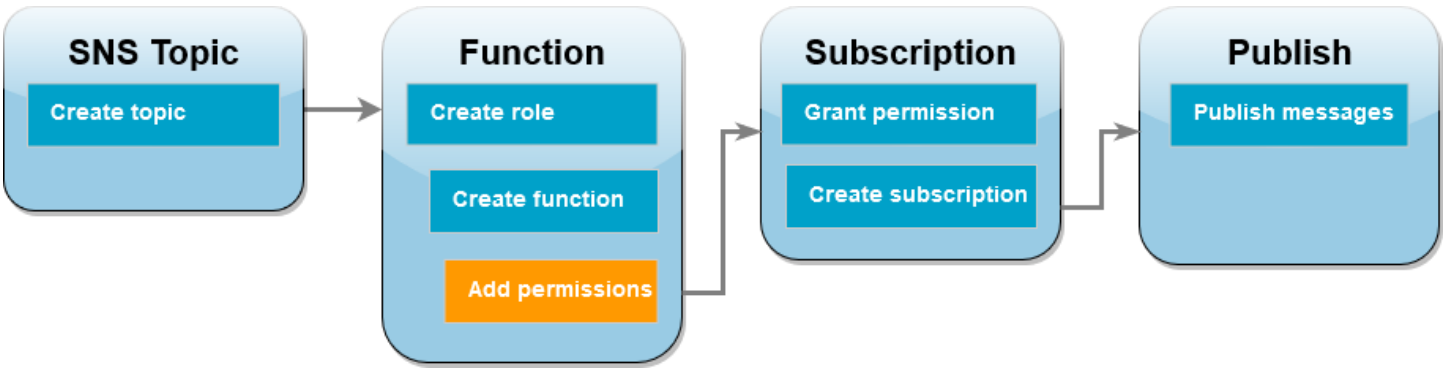
```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs18.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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. 함수의 Amazon 리소스 이름(ARN)을 기록합니다. 이는 나중에 자습서에서 Amazon SNS에 함수를 간접적으로 호출할 권한을 추가할 때 필요합니다.

함수에 권한 추가(계정 B)



Amazon SNS가 함수를 간접적으로 호출하려면 리소스 기반 정책의 문에서 권한을 부여해야 합니다. AWS CLI `add-permission` 명령을 사용하여 이 문을 추가합니다.

Amazon SNS에 함수를 간접적으로 호출할 수 있는 권한 부여

- 계정 B에서 이전에 기록한 Amazon SNS 주제의 ARN을 사용하여 다음 AWS CLI 명령을 실행합니다.

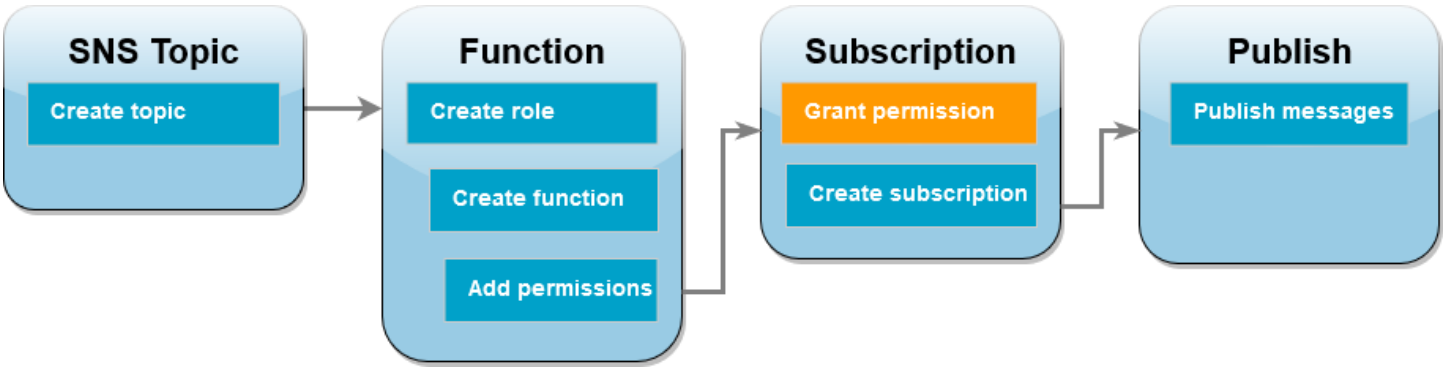
```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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 주제가 있는 계정이 옵트인 AWS 리전에서 호스팅되는 경우 보안 주체에서 리전을 지정해야 합니다. 예를 들어, 아시아 태평양(홍콩) 리전에서 Amazon SNS 주제를 사용하는 경우 보안 주체에 대해 `sns.amazonaws.com` 대신 `sns.ap-east-1.amazonaws.com`을 지정해야 합니다.

Amazon SNS 구독을 위한 크로스 계정 권한 부여(계정 A)



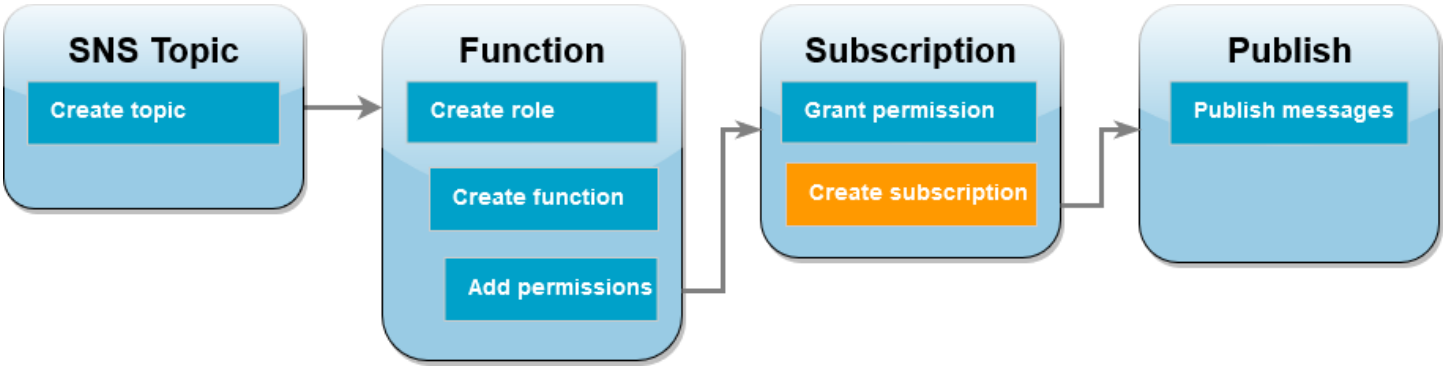
계정 B의 Lambda 함수가 계정 A에서 생성한 Amazon SNS 주제를 구독할 수 있도록 계정 B에 주제를 구독할 수 있는 권한을 부여해야 합니다. AWS CLI `add-permission` 명령을 사용하여 이 권한을 부여합니다.

계정 B가 주제를 구독할 수 있는 권한 부여

- 계정 A에서 다음 AWS CLI 명령을 실행합니다. 이전에 기록한 Amazon SNS 주제의 ARN을 사용합니다.

```
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
```

구독 생성(계정 B)



이제 계정 B에서 계정 A의 자습서 시작 부분에서 생성한 Amazon SNS 주제에 대한 Lambda 함수를 구독합니다. 이 주제(`sns-topic-for-lambda`)로 메시지가 전송되면 Amazon SNS는 계정 B에서 Lambda 함수 `Function-With-SNS`를 호출합니다.

구독 생성

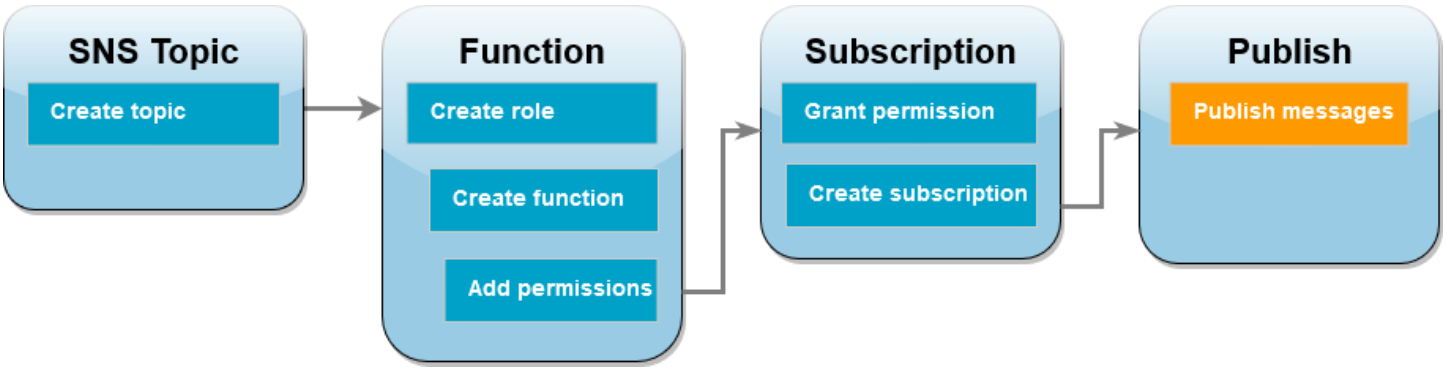
- 계정 B에서 다음 AWS CLI 명령을 실행합니다. 주제를 생성한 기본 리전과 주제 및 Lambda 함수의 ARN을 사용합니다.

```
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
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

주제에 메시지 게시(계정 A 및 계정 B)



이제 계정 B의 Lambda 함수가 계정 A의 Amazon SNS 주제를 구독하므로 주제에 메시지를 게시하여 설정을 테스트할 차례입니다. Amazon SNS가 Lambda 함수를 간접적으로 호출했는지 확인하려면 CloudWatch Logs를 사용하여 함수의 출력을 봅니다.

주제에 메시지 게시 및 함수의 출력 보기

1. 텍스트 파일에 `Hello World`를 입력하고 `message.txt`로 저장합니다.
2. 텍스트 파일을 저장한 디렉터리와 동일한 디렉터리에서 계정 A의 다음 AWS CLI 명령을 실행합니다. 사용자 주제의 ARN을 사용합니다.

```
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
```

그러면 Amazon SNS가 메시지를 수락했음을 나타내는 고유 식별자가 포함된 메시지 ID가 반환됩니다. Amazon SNS는 주제의 구독자에게 메시지를 전달하려고 시도합니다. Amazon SNS가 Lambda 함수를 간접적으로 호출했는지 확인하려면 CloudWatch Logs를 사용하여 함수의 출력을 봅니다.

3. 계정 B에서 Amazon CloudWatch 콘솔의 로그 그룹 페이지를 엽니다.
4. 함수에 대한 로그 그룹(`/aws/lambda/Function-With-SNS`)을 선택합니다.
5. 최신 로그 스트림을 선택합니다.
6. 함수가 간접적으로 제대로 호출되면 주제에 게시한 메시지의 내용을 보여주는 다음과 유사한 출력이 표시됩니다.

```
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
```

리소스 정리

이 자습서 용도로 생성한 리소스를 보관하고 싶지 않다면 지금 삭제할 수 있습니다. 더 이상 사용하지 않는 AWS 리소스를 삭제하면 AWS 계정에 불필요한 요금이 발생하는 것을 방지할 수 있습니다.

계정 A에서 Amazon SNS 주제를 정리합니다.

Amazon SNS 주제를 삭제하려면

1. Amazon SNS 콘솔의 주제 페이지를 엽니다.
2. 생성한 주제를 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 **`delete me`**을 입력합니다.
5. Delete(삭제)를 선택합니다.

계정 B에서 실행 역할, Lambda 함수 및 Amazon SNS 구독을 정리합니다.

집행 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 생성한 실행 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 텍스트 입력 필드에 역할의 이름을 입력하고 Delete(삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **`delete`**를 입력하고 Delete(삭제)를 선택합니다.

Amazon SNS 구독을 삭제하려면

1. Amazon SNS 콘솔의 구독 페이지를 엽니다.
2. 생성한 구독을 선택합니다.
3. 삭제(Delete)와 삭제(Delete)를 차례로 선택합니다.

AWS Lambda 함수 작업의 모범 사례

다음은 AWS Lambda 사용에 대한 권장 모범 사례입니다.

주제

Lambda 애플리케이션의 모범 사례에 대한 자세한 내용은 Serverless Land의 Application design을 참조하세요. AWS 계정 팀에 연락하여 아키텍처 검토를 요청할 수도 있습니다.

함수 코드

- 핵심 로직에서 Lambda 핸들러를 분리합니다. 이를 통해 단위 테스트를 수행할 수 있는 더 많은 함수를 만들 수 있습니다. Node.js에서 이는 다음과 같을 수 있습니다.

```
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
}
```

- 실행 환경 재사용을 활용하여 함수 성능을 향상시킵니다. 함수 핸들러 외부에서 SDK 클라이언트 및 데이터베이스 연결을 초기화하고 정적 자산을 `/tmp` 디렉토리에 로컬로 캐시합니다. 동일한 함수 인스턴스에서 처리하는 후속 호출은 이러한 리소스를 재사용할 수 있습니다. 이를 통해 함수 실행 시간을 줄여 비용을 절감합니다.

호출에서 발생할 수 있는 데이터 유출을 방지하려면 실행 환경을 사용하여 사용자 데이터, 이벤트 또는 보안과 관련된 기타 정보를 저장하지 마세요. 함수가 핸들러 내부 메모리에 저장할 수 없는 변경 가능한 상태에 의존하는 경우 각 사용자에 대해 별도의 함수 또는 별도의 함수 버전을 생성하는 것이 좋습니다.

- 연결 유지 지시문을 사용하여 지속적인 연결을 유지하세요. Lambda는 시간이 지남에 따라 유휴 연결을 제거합니다. 함수를 호출할 때 유휴 연결을 재사용하려고 하면 연결 오류가 발생합니다. 지속적인 연결을 유지하려면 런타임과 관련된 연결 유지 지시문을 사용하세요. 예를 들어, Node.js에서 연결 유지를 이용해 연결 재사용을 참조하세요.
- 환경 변수를 사용하여 함수에 운영 파라미터를 전달합니다. 예를 들어, Amazon S3 버킷에 기록하는 경우 기록하고 있는 버킷 이름을 하드 코딩하는 대신 환경 변수로 구성합니다.
- 함수 배포 패키지의 종속성을 제어합니다. AWS Lambda 실행 환경에는 Node.js 및 Python 런타임용 AWS SDK와 같은 여러 라이브러리가 포함되어 있습니다(전체 목록은 Lambda 런타임에서 찾을 수 있음). 최신 기능 및 보안 업데이트를 활성화하려면 Lambda가 주기적으로 이러한 라이브러리를 업데이트해야 합니다. 이러한 업데이트는 Lambda 함수의 동작에 사소한 변화를 가져올 수 있습니다. 함수가 사용하는 종속성을 완전히 제어하려면 모든 종속성을 배포 패키지로 패키징하세요.
- 배포 패키지 크기를 런타임 필요에 따라 최소화합니다. 이렇게 하면 호출 전에 배포 패키지를 다운로드하고 압축을 풀 때 걸리는 시간이 단축됩니다. Java 또는 .NET Core에서 작성된 함수의 경우 배포 패키지의 일부로 전체 AWS SDK 라이브러리를 업로드하지 마세요. 대신, 필요한 SDK의 구성 요소를 선택하는 모듈을 선택적으로 활용합니다(예: DynamoDB, Amazon S3 SDK 모듈, Lambda 핵심 라이브러리).
- 종속성 `.jar` 파일을 별도의 /lib 디렉터리에 배치하여 Java에서 생성된 배포 패키지의 압축을 푸는 데 Lambda가 소요하는 시간을 단축합니다. 이는 많은 수의 `.class` 파일이 있는 단일 jar에 모든 함수 코드를 배치하는 것보다 빠릅니다. 자세한 내용은 .zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포 섹션을 참조하세요.
- 종속성의 복잡성을 최소화합니다. 실행 환경 시작 시 빠르게 로드되는 더 단순한 프레임워크가 권장됩니다. 예를 들어 Spring Framework와 같은 더 복잡한 프레임워크보다는 Dagger 또는 Guice 같은 더 단순한 Java 종속성 주입(IoC) 프레임워크를 선호합니다.
- 일부 임의 기준이 충족될 때까지 함수가 자동으로 자체 호출이 되는 리커시브 코드를 Lambda 함수에서 사용하지 않도록 합니다. 리커시브 코드를 사용할 경우, 의도하지 않은 함수 호출이 증가하고 비용이 상승할 수 있습니다. 실수로 그렇게 한 경우 코드를 업데이트하는 동안 즉시 함수 예약된 동시성을 `0`으로 설정하여 해당 함수에 대한 모든 호출을 조절합니다.
- Lambda 함수 코드에는 문서화되지 않은 비공개 API를 사용하지 마세요. AWS Lambda 관리형 런타임의 경우, Lambda는 주기적으로 보안 및 기능 업데이트를 Lambda의 내부 API에 적용합니다. 이러한 내부 API 업데이트는 이전 버전과 호환되지 않으므로 함수가 이러한 비공개 API에 종속성을 갖는

경우 호출 실패와 같은 의도하지 않은 결과를 초래할 수 있습니다. 공개적으로 사용 가능한 API의 목록은 API 레퍼런스를 참조하세요.

- 멱등성 코드를 작성합니다. 함수에 멱등성 코드를 작성하면 중복 이벤트가 동일한 방식으로 처리됩니다. 코드는 이벤트를 올바르게 검증하고 중복 이벤트를 정상적으로 처리해야 합니다. 자세한 내용은 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까? 단원을 참조하십시오.
- Java DNS 캐시를 사용하지 마십시오. Lambda 함수는 이미 DNS 응답을 캐싱하고 있습니다. 다른 DNS 캐시를 사용하는 경우 연결 시간 초과가 발생할 수 있습니다.

  `java.util.logging.Logger` 클래스는 JVM DNS 캐시를 간접적으로 활성화할 수 있습니다. 기본 설정을 재정의하려면 `logger` 초기화 전에 networkaddress.cache.ttl을 0으로 설정하십시오. 예제

```
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` 오류를 방지하려면 `networkaddress.cache.negative.ttl`을 0으로 설정하는 것이 좋습니다. `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` 환경 변수를 사용하여 Lambda 함수에 대해 이 속성을 설정할 수 있습니다.

  JVM DNS 캐시를 비활성화해도 Lambda의 관리형 DNS 캐싱은 비활성화되지 않습니다.

함수 구성

- Lambda 함수를 테스트하는 성능은 최적의 메모리 크기 구성을 선택할 때 매우 중요합니다. 메모리 크기가 증가하면 함수에 사용 가능한 상응하는 CPU 사용이 증가합니다. 함수의 메모리 사용은 호출마다 결정되며 Amazon CloudWatch에서 볼 수 있습니다. 매 호출마다 아래와 같이 `REPORT:` 항목이 만들어집니다.

```
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:` 필드를 분석함으로써 함수가 더 많은 메모리를 필요로 하는지 또는 함수의 메모리 크기가 과다 프로비저닝되었는지 확인할 수 있습니다.

기능에 적합한 메모리 구성을 찾으려면 오픈 소스 AWS Lambda Power Tuning 프로젝트를 사용하는 것이 좋습니다. 자세한 내용은 GitHub에서 AWS Lambda Power Tuning을 참조하세요.

함수 성능을 최적화하려면 Advanced Vector Extensions 2(AVX2)를 활용할 수 있는 라이브러리를 배포하는 것이 좋습니다. 이를 통해 기계 학습 추론, 미디어 처리, 고성능 컴퓨팅(HPC), 과학 시뮬레이션, 금융 모델링 등 까다로운 워크로드를 처리할 수 있습니다. 자세한 내용은 Creating faster AWS Lambda functions with AVX2를 참조하세요.

- Lambda 함수를 로드 테스트하여 최적의 제한 시간 값을 확인합니다. 함수의 실행 시간을 분석하여 함수의 동시성을 기대 이상으로 높일 수 있는 종속성 서비스를 통해 문제를 더 효과적으로 확인하는 것이 중요합니다. 이는 Lambda 함수가 Lambda의 조정을 처리하지 못할 수 있는 리소스에 대한 네트워크 호출을 수행할 때 특히 중요합니다.
- IAM 정책을 설정할 때 가장 제한적인 권한을 사용합니다. Lambda 함수에 필요한 리소스와 작업을 이해하고 실행 역할을 이러한 권한으로 제한합니다. 자세한 내용은 AWS Lambda에서 권한 관리 섹션을 참조하세요.
- Lambda 할당량 단원의 내용을 숙지합니다. 페이로드 크기, 파일 설명자 및 /tmp 공간은 런타임 리소스 제한을 결정할 때 자주 간과됩니다.
- 더 이상 사용하지 않는 Lambda 함수를 삭제합니다. 삭제하면 사용되지 않는 함수는 배포 패키지 크기 제한에 불필요하게 포함되지 않습니다.
- 이벤트 소스로 Amazon Simple Queue Service를 사용하는 경우 함수의 예상 호출 시간 값이 대기열의 표시 제한 시간 값을 초과하지 않도록 합니다. 이는 CreateFunction 및 UpdateFunctionConfiguration 모두에 적용됩니다.
  - CreateFunction의 경우 AWS Lambda가 함수 생성 프로세스에 실패합니다.
  - UpdateFunctionConfiguration의 경우, 함수가 중복 호출될 수 있습니다.

함수의 확장성

- 업스트림 및 다운스트림 처리량 제한 사항을 잘 알고 있어야 합니다. Lambda 함수는 로드에 따라 원활하게 규모를 조정할 수 있지만 업스트림 및 다운스트림 종속성의 처리량 용량이 동일하지 않을 수 있습니다. 함수의 규모 조정 가능 범위를 제한해야 하는 경우 함수에서 예약된 동시성을 구성할 수 있습니다.
- 제한 허용치를 구축하세요. Lambda의 조정 속도를 초과하는 트래픽으로 인해 동기 함수에서 제한이 발생하는 경우 다음 전략을 사용하여 제한 허용치를 개선할 수 있습니다.

- 시간 제한, 재시도, 지터가 포함된 백오프를 사용하세요. 이러한 전략을 구현하면 간접 호출의 재시도가 원활해지고 Lambda가 몇 초 내에 스케일 업하여 최종 사용자 제한을 최소화할 수 있습니다.
- 프로비저닝된 동시성을 사용하세요. 프로비저닝된 동시성은 Lambda가 함수에 할당하는 사전 초기화된 실행 환경의 수입니다. Lambda는 가능한 경우 프로비저닝된 동시성을 사용하여 수신 요청을 처리합니다. 또한 필요한 경우 Lambda는 프로비저닝된 동시성 설정 이상으로 함수를 확장할 수도 있습니다. 프로비저닝된 동시성을 구성하는 경우 AWS 계정에 추가 요금이 부과됩니다.

지표 및 경보

- Lambda 함수 코드 내에서 지표를 생성하거나 업데이트하는 대신 Lambda 함수 지표 작업 및 CloudWatch 경보를 사용합니다. 이는 Lambda 함수의 상태를 파악하는 훨씬 더 효율적인 방법이며, 개발 프로세스 초기에 문제를 파악할 수 있습니다. 예를 들어, 함수 코드로 인한 병목 현상이나 지연 시간을 해결하기 위해 예상되는 Lambda 함수 호출 소요 시간을 기준으로 경보를 구성할 수 있습니다.
- 로깅 라이브러리 및 AWS Lambda 지표 및 차원을 사용하여 앱 오류를 파악합니다(ERR, ERROR, WARNING 등).
- AWS Cost Anomaly Detection을 사용하여 계정에서 비정상적인 활동을 감지합니다. Cost Anomaly Detection은 기계 학습을 사용하여 비용과 사용량을 모니터링하고 오탐지 알림을 최소화합니다. Cost Anomaly Detection은 AWS Cost Explorer의 데이터를 사용하며, 최대 24시간의 지연이 있습니다. 따라서 사용량이 발생한 후 이상 탐지에 최대 24시간이 걸릴 수 있습니다. Cost Anomaly Detection을 시작하려면 먼저 Cost Explorer에 가입해야 합니다. 이후 Cost Anomaly Detection에 액세스합니다.

스트림 작업

- 서로 다른 배치 및 레코드 크기로 테스트하여 각 이벤트 소스의 폴링 빈도를 조정하고 해당 함수가 얼마나 빨리 작업을 완료할 수 있는지 확인합니다. CreateEventSourceMapping BatchSize 파라미터는 각 호출에서 함수로 보낼 수 있는 최대 레코드 수를 제어합니다. 배치 크기가 클수록 더 큰 레코드 세트에서 호출 오버헤드를 더 효율적으로 수용하여 처리량을 늘릴 수 있습니다.

  기본적으로, Lambda는 레코드가 사용 가능하게 되는 즉시 함수를 호출합니다. Lambda가 이벤트 소스에서 읽는 배치에 하나의 레코드만 있는 경우, Lambda는 함수에 하나의 레코드만 전송합니다. 소수의 레코드로 함수를 호출하는 것을 피하려면 일괄 처리 기간을 구성하여 이벤트 소스가 최대 5분 동안 레코드를 버퍼링하도록 지정할 수 있습니다. 함수를 호출하기 전에 Lambda는 전체 배치가 수

집되거나, 일괄 처리 기간이 만료되거나, 배치가 페이로드 한도인 6MB에 도달할 때까지 이벤트 소스에서 레코드를 계속 읽습니다. 자세한 내용은 일괄 처리 동작 단원을 참조하십시오.

> **Warning**
>
> Lambda 이벤트 소스 매핑은 각 이벤트를 한 번 이상 처리하므로 레코드가 중복될 수 있습니다. 중복 이벤트와 관련된 잠재적 문제를 방지하려면 함수 코드를 멱등성으로 만드는 것이 좋습니다. 자세한 내용은 AWS 지식 센터의 멱등성 Lambda 함수를 만들려면 어떻게 해야 합니까?를 참조하세요.

- 샤드를 추가하여 Kinesis 스트림 처리량을 늘립니다. Kinesis 스트림은 하나 이상의 샤드로 구성됩니다. Lambda는 최대 한 개의 동시 호출로 각 샤드를 폴링합니다. 예를 들어 스트림에 100개의 활성 샤드가 있으면 최대 100개의 Lambda 함수 호출이 동시에 실행됩니다. 샤드 수를 늘리면 Lambda 함수의 최대 동시 호출 수가 증가하고 Kinesis 스트림 처리량이 증가할 수 있습니다. Kinesis 스트림에서 샤드 수를 늘리는 경우 데이터에 대해 적절한 파티션 키를 선택하여(파티션 키 참조) 관련 레코드가 동일한 샤드에서 끝나며 데이터가 잘 분산되도록 해야 합니다.
- IteratorAge에 대해 Amazon CloudWatch를 사용하여 Kinesis 스트림이 처리 중인지 확인합니다. 예를 들어 최대 값을 30,000(30초)으로 설정하여 CloudWatch 경보를 구성합니다.

보안 모범 사례

- AWS Security Hub을 사용하여 보안 모범 사례와 관련된 AWS Lambda의 사용량을 모니터링하십시오. Security Hub는 보안 제어를 사용하여 리소스 구성 및 보안 표준을 평가하여 다양한 규정 준수 프레임워크를 준수할 수 있도록 지원합니다. Security Hub를 사용하여 Lambda 리소스를 평가하는 방법에 대한 자세한 내용은 AWS Security Hub 사용 설명서의 AWS Lambda 제어를 참조하십시오.
- Amazon GuardDuty Lambda 보호를 사용하여 Lambda 네트워크 활동 로그를 모니터링하세요. GuardDuty Lambda 보호를 사용하면 AWS 계정에서 Lambda 함수가 호출될 때 잠재적인 보안 위협을 식별하는 데 도움이 됩니다. 함수 중 하나가 가상 화폐와 관련된 활동과 연결된 IP 주소를 쿼리하는 경우를 예로 들 수 있습니다. GuardDuty는 Lambda 함수 호출 시 생성되는 네트워크 활동 로그를 모니터링합니다. 자세한 내용은 Amazon GuardDuty 사용 설명서의 Lambda 보호를 참조하세요.

AWS Lambda에서 권한 관리

AWS Lambda에서 AWS Identity and Access Management(IAM)를 사용하여 권한을 관리합니다. Lambda 함수 작업 시 고려해야 할 두 가지 주요 권한 범주가 있습니다.

- Lambda 함수가 API 작업을 수행하고 다른 AWS 리소스에 액세스하는 데 필요한 권한입니다.
- 다른 AWS 사용자 및 엔터티가 Lambda 함수에 액세스하는 데 필요한 권한

Lambda 함수는 종종 다른 AWS 리소스에 액세스하고 해당 리소스에 대해 다양한 API 작업을 수행해야 합니다. 예를 들어 Amazon DynamoDB 데이터베이스의 항목을 업데이트하여 이벤트에 응답하는 Lambda 함수가 있을 수 있습니다. 이 경우 데이터베이스에 액세스할 수 있는 권한과 해당 데이터베이스에 항목을 추가하거나 업데이트할 수 있는 권한이 함수에 필요합니다.

실행 역할이라는 특수 IAM 역할에서 Lambda 함수에 필요한 권한을 정의합니다. 이 역할에서는 함수가 다른 AWS 리소스에 액세스하고 이벤트 소스에서 읽는 데 필요한 모든 권한을 정의하는 정책을 연결할 수 있습니다. 모든 Lambda 함수에는 실행 역할이 있어야 합니다. Lambda 함수는 기본적으로 CloudWatch Logs에 기록되므로 실행 역할에 최소한 Amazon CloudWatch에 대한 액세스 권한이 있어야 합니다. `AWSLambdaBasicExecutionRole` 관리형 정책을 실행 역할에 연결하여 이 요구 사항을 충족할 수 있습니다.

다른 AWS 계정, 조직 및 서비스에 Lambda 리소스에 액세스할 수 있는 권한을 부여하려는 경우 다음과 같은 몇 가지 옵션이 있습니다.

- ID 기반 정책을 사용하여 다른 사용자에게 Lambda 리소스에 대한 액세스 권한을 부여할 수 있습니다. 자격 증명 기반 정책은 사용자에게 직접 적용하거나, 사용자와 연결된 그룹 및 역할에 적용할 수 있습니다.
- 리소스 기반 정책을 사용하여 다른 계정과 AWS 서비스에 Lambda 리소스를 사용할 권한을 부여할 수 있습니다. 사용자가 Lambda 리소스에 액세스하려고 하면 Lambda는 사용자의 자격 증명 기반 정책 및 리소스의 리소스 기반 정책을 모두 고려합니다. Amazon Simple Storage Service(Amazon S3 같은 AWS 서비스가 Lambda 함수를 호출하면 Lambda는 리소스 기반 정책만 고려합니다.
- ABAC(속성 기반 액세스 제어) 모델을 사용하여 Lambda 함수에 대한 액세스를 제어할 수 있습니다. ABAC를 사용하면 태그를 Lambda 함수에 연결하거나, 특정 API 요청에 전달하거나, 요청을 하는 IAM 보안 주체에 연결할 수 있습니다. 함수 액세스를 제어하려면 IAM 정책의 조건 요소에 동일한 태그를 지정하세요.

AWS에서는 작업을 수행하는 데 필요한 권한(least-privilege permissions)만 부여하는 것이 모범 사례입니다. 이를 Lambda에서 구현하려면 AWS 관리형 정책으로 시작하는 것이 좋습니다. 이러한 관리형 정책을 그대로 사용하거나, 이를 바탕으로 보다 제한적인 정책을 직접 만드는 시작점으로 활용할 수 있습니다.

최소 권한 액세스에 대한 권한을 미세 조정할 수 있도록 Lambda는 정책에 포함할 수 있는 몇 가지 추가 조건을 제공합니다. 자세한 내용은 the section called “리소스와 조건” 단원을 참조하십시오.

IAM에 대한 자세한 내용은 IAM 사용 설명서를 참조하세요.

실행 역할로 Lambda 함수 권한 정의

Lambda 함수의 실행 역할은 AWS 서비스 및 리소스에 액세스할 수 있는 권한을 함수에 부여하는 AWS Identity and Access Management(IAM) 역할입니다. Amazon CloudWatch에 로그를 전송하고, AWS X-Ray에 추적 데이터를 업로드하는 권한을 가진 실행 역할을 만들 수 있습니다. 이 페이지에서는 Lambda 함수의 실행 역할을 생성하고 보고 관리하는 방법에 대한 정보를 제공합니다.

자신의 함수를 호출하면 Lambda는 자동으로 실행 역할을 맡습니다. 함수 코드에서 실행 역할을 수임하기 위해 수동으로 `sts:AssumeRole`을 직접 호출하는 것을 피해야 합니다. 사용 사례에서 역할을 맡아야 하는 경우 자신의 역할의 신뢰 정책에, 역할 자체를 신뢰할 수 있는 보안 주체로 포함해야 합니다. 역할 신뢰 정책을 수정하는 방법에 대한 자세한 내용은 IAM 사용 설명서의 역할 신뢰 정책 수정(콘솔)을 참조하세요.

Lambda가 실행 역할을 제대로 수임하려면 역할의 신뢰 정책에서 Lambda 서비스 주체(`lambda.amazonaws.com`)를 신뢰할 수 있는 서비스로 지정해야 합니다.

주제

- IAM 콘솔에서 실행 역할 생성
- AWS CLI로 사용하여 역할 생성 및 관리
- Lambda 실행 역할에 최소 권한 액세스 부여
- 실행 역할의 권한 보기 및 업데이트
- 실행 역할에서 AWS 관리형 정책 작업
- 소스 함수 ARN을 사용하여 함수 액세스 동작 제어

IAM 콘솔에서 실행 역할 생성

기본적으로 Lambda는 Lambda 콘솔에서 함수를 생성할 때 최소한의 권한으로 실행 역할을 생성합니다. 특히 이 실행 역할에는 함수에 Amazon CloudWatch Logs에 이벤트를 기록할 수 있는 기본 권한을 부여하는 `AWSLambdaBasicExecutionRole` 관리형 정책이 포함되어 있습니다.

함수에서 보다 의미 있는 작업을 수행하려면 일반적으로 추가 권한이 필요합니다. 예를 들어 Amazon DynamoDB 데이터베이스의 항목을 업데이트하여 이벤트에 응답하는 Lambda 함수가 있을 수 있습니다. IAM 콘솔을 사용하여 필요한 권한이 있는 실행 역할을 생성할 수 있습니다.

IAM 콘솔에서 실행 역할을 생성하려면

1. IAM 콘솔에서 역할(Roles) 페이지를 엽니다.

2. 역할 생성을 선택합니다.
3. 신뢰할 수 있는 엔터티 유형에서 AWS 서비스를 선택합니다.
4. 사용 사례에서 Lambda를 선택합니다.
5. Next(다음)를 선택합니다.
6. 역할에 연결할 AWS 관리형 정책을 선택합니다. 예를 들어 함수가 DynamoDB에 액세스해야 하는 경우 AWSLambdaDynamoDBExecutionRole 관리형 정책을 선택하세요.
7. Next(다음)를 선택합니다.
8. Role name을 입력하고 Create role을 선택합니다.

자세한 지침은 IAM 사용 설명서의 AWS 서비스에 대한 역할 생성(콘솔)을 참조하세요.

실행 역할을 생성한 후 함수에 연결합니다. Lambda 콘솔에서 함수를 생성할 때 이전에 생성한 모든 실행 역할을 함수에 연결할 수 있습니다. 기존 함수에 새 실행 역할을 연결하려면 해당 단계를 따르세요.

AWS CLI로 사용하여 역할 생성 및 관리

AWS Command Line Interface(AWS CLI)를 사용하여 실행 역할을 생성하려면 create-role 명령을 사용합니다. 이 명령을 사용할 때는 신뢰 정책 인라인을 지정할 수 있습니다. 역할의 신뢰 정책은 지정된 보안 주체에게 역할을 맡을 수 있는 권한을 부여합니다. 다음 예에서는 Lambda 서비스 보안 주체에게 자신의 역할을 맡을 권한을 부여합니다. JSON 문자열의 이스케이프 따옴표 요구 사항은 셸에 따라 다릅니다.

```
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"}]}'
```

별도의 JSON 파일을 사용하여 역할에 대한 신뢰 정책을 정의할 수도 있습니다. 다음 예제에서 `trust-policy.json`은 최신 디렉터리의 파일입니다.

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
```

다음 결과가 표시됩니다.

```
{
    "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"
                }
            ]
        }
    }
}
```

역할에 권한을 추가하려면 attach-policy-to-role 명령을 사용하세요. 다음 명령은 `AWSLambdaBasicExecutionRole` 관리형 정책을 `lambda-ex` 실행 역할에 추가합니다.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

실행 역할을 생성한 후 함수에 연결합니다. Lambda 콘솔에서 함수를 생성할 때 이전에 생성한 모든 실행 역할을 함수에 연결할 수 있습니다. 기존 함수에 새 실행 역할을 연결하려면 해당 단계를 따르세요.

Lambda 실행 역할에 최소 권한 액세스 부여

개발 단계 중에 Lambda에 대한 IAM 역할을 처음 생성하는 경우 간혹 필요 이상의 권한을 부여하게 될 수 있습니다. 모범 사례로 프로덕션 환경에 함수를 게시하기 전에 필요한 권한만 포함하도록 정책을 조정하는 것이 가장 좋습니다. 자세한 정보는 IAM 사용 설명서의 최소 권한 적용을 참조하세요.

IAM Access Analyzer를 사용하면 IAM 실행 역할 정책에 필요한 권한을 식별하는 데 도움이 될 수 있습니다. IAM Access Analyzer는 지정한 날짜 범위에 걸친 AWS CloudTrail 로그를 검토하여 해당 기간 동안 함수가 사용했던 권한만을 포함한 정책 템플릿을 생성합니다. 템플릿을 사용하여 세분화된 권한을 가진 관리형 정책을 생성한 다음 IAM 역할에 연결할 수 있습니다. 이렇게 하면 특정 사용 사례에 따라 역할이 AWS 리소스와 상호 작용하는 데 필요한 권한만 부여할 수 있습니다.

자세한 내용은 IAM 사용 설명서에서 액세스 활동을 기반으로 정책 생성을 참조하세요.

실행 역할의 권한 보기 및 업데이트

이 항목에서는 함수의 실행 역할을 보고 업데이트하는 방법을 다룹니다.

주제

- 함수의 실행 역할 보기
- 함수의 실행 역할 업데이트

함수의 실행 역할 보기

함수의 실행 역할을 보려면 Lambda 콘솔을 사용하세요.

함수의 실행 역할을 보려면 다음을 수행하세요(콘솔).

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration)을 선택한 다음 권한(Permissions)을 선택합니다.
4. 실행 역할에서 함수의 실행 역할로 현재 사용되고 있는 역할을 볼 수 있습니다. 편의를 위해 리소스 요약 섹션에서 함수가 액세스할 수 있는 모든 리소스와 작업을 볼 수 있습니다. 드롭다운 목록에서 서비스를 선택하여 해당 서비스와 관련된 모든 권한을 확인할 수 있습니다.

함수의 실행 역할 업데이트

언제든지 함수의 실행 역할에서 원한을 추가하거나 제거할 수 있으며, 다른 역할에 사용하도록 함수를 구성할 수 있습니다. 함수가 다른 서비스나 리소스에 액세스해야 하는 경우 실행 역할에 필수 권한을 추가해야 합니다.

함수에 권한을 추가할 때 코드나 구성에 대한 간단한 업데이트도 수행하세요. 그러면 자격 증명이 오래된 함수의 인스턴스 실행이 중지 및 대체됩니다.

함수의 실행 역할을 업데이트하려면 Lambda 콘솔을 사용하세요.

함수의 실행 역할을 업데이트하려면 다음을 수행하세요(콘솔).

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수의 이름을 선택합니다.
3. 구성(Configuration)을 선택한 다음 권한(Permissions)을 선택합니다.
4. 실행 역할에서 편집을 선택합니다.
5. 다른 역할을 실행 역할로 사용하도록 함수를 업데이트하려면 기존 역할 아래의 드롭다운 메뉴에서 새 역할을 선택합니다.

   > **Note**
   >
   > 기존 실행 역할 내에서 권한을 업데이트하려는 경우 AWS Identity and Access Management(IAM) 콘솔에서만 가능합니다.

   실행 역할로 사용할 새 역할을 생성하려면 실행 역할 아래의 AWS 정책 템플릿에서 새 역할 생성을 선택합니다. 그런 다음 역할 이름 아래에 새 역할의 이름을 입력하고 정책 템플릿 아래에서 새 역할에 연결할 정책을 지정합니다.
6. Save(저장)를 선택합니다.

실행 역할에서 AWS 관리형 정책 작업

다음 AWS 관리형 정책은 Lambda 기능을 사용하는 데 필요한 권한을 제공합니다.

변경 사항	설명	날짜
AWSLambdaMSKExecutionRole – Lambda에서 이 정책에 kafka:DescribeClusterV2 권한을 추가했습니다.	AWSLambdaMSKExecutionRole 은 Amazon Managed Streaming for Apache Kafka(Amazon MSK) 클러스터에서 레코드를 읽고 액세스하고, 탄력적 네트워크 인터페이스(ENI)를 관리하고, CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 6월 17일
AWSLambdaBasicExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaBasicExecutionRole 은 CloudWatch에 로그를 업로드할 수 있는 권한을 부여합니다.	2022년 2월 14일
AWSLambdaDynamoDBExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaDynamoDBExecutionRole 은 Amazon DynamoDB Stream에서 레코드를 읽고 CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일
AWSLambdaKinesisExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaKinesisExecutionRole 은 Amazon Kinesis 데이터 스트림에서 이벤트를 읽고 CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일
AWSLambdaMSKExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaMSKExecutionRole 은 Amazon Managed Streaming for Apache Kafka(Amazon MSK) 클러스터에서 레코드를 읽고 액세스하고, 탄력적 네트워크 인터페이스(ENI)를 관리하고,	2022년 2월 14일

변경 사항	설명	날짜
	CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	
AWSLambdaSQSQueueExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaSQSQueueExecutionRole 은 Amazon Simple Queue Service(Amazon SQS) 대기열에서 메시지를 읽고 CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일
AWSLambdaVPCAccessExecutionRole – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSLambdaVPCAccessExecutionRole 은 Amazon VPC 내에서 ENI를 관리하고 CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일
AWSXRayDaemonWriteAccess – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AWSXRayDaemonWriteAccess 는 추적 데이터를 X-Ray에 업로드할 수 있는 권한을 부여합니다.	2022년 2월 14일
CloudWatchLambdaInsightsExecutionRolePolicy – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	CloudWatchLambdaInsightsExecutionRolePolicy 는 CloudWatch Lambda Insights에 런타임 지표를 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일
AmazonS3ObjectLambdaExecutionRolePolicy – Lambda가 이 정책의 변경 사항 추적을 시작했습니다.	AmazonS3ObjectLambdaExecutionRolePolicy 는 Amazon Simple Storage Service(Amazon S3) 객체 Lambda와 상호 작용하고 CloudWatch 로그에 쓸 수 있는 권한을 부여합니다.	2022년 2월 14일

일부 기능의 경우 Lambda 콘솔은 고객 관리형 정책의 실행 역할에 누락된 권한을 추가하려고 시도합니다. 이러한 정책은 많아질 수 있습니다. 추가 정책을 만들지 않도록 기능을 활성화하기 전에 실행 역할에 관련된 AWS 관리형 정책을 추가합니다.

이벤트 소스 매핑을 사용하여 함수를 호출할 경우 Lambda는 실행 역할을 사용해 이벤트 데이터를 읽습니다. 예를 들어 Kinesis에 대한 이벤트 소스 매핑은 데이터 스트림의 이벤트를 읽고 이를 함수에 배치(batch)로 전송합니다.

서비스가 계정에서 역할을 맡은 경우, 역할 신뢰 정책의 `aws:SourceAccount` 및 `aws:SourceArn` 전역 조건 컨텍스트 키를 포함하여 역할에 대한 액세스를 예상 리소스에 의해 생성된 요청으로만 제한할 수 있습니다. 자세한 내용은 AWS Security Token Service에 대한 교차 서비스 혼동된 대리자 예방을 참조하세요.

AWS 관리형 정책 외에도 Lambda 콘솔은 추가 사용 사례와 관련된 권한을 가진 사용자 지정 정책을 생성하는 템플릿을 제공합니다. Lambda 콘솔에서 함수를 생성할 때 하나 이상의 템플릿의 권한을 사용하여 새로운 실행 역할을 만들 수 있습니다. 블루프린트에서 함수를 생성하거나, 다른 서비스에 액세스해야 하는 옵션을 구성할 때 이러한 템플릿도 자동으로 적용됩니다. 예제 템플릿은 이 설명서의 GitHub 리포지토리에서 구할 수 있습니다.

소스 함수 ARN을 사용하여 함수 액세스 동작 제어

Lambda 함수 코드가 다른 AWS 서비스에 API 요청을 보내는 일은 흔히 발생합니다. 이러한 요청을 수행하기 위해 Lambda는 함수의 실행 역할을 가정하여 휘발성 자격 증명 세트를 생성합니다. 이러한 보안 인증 정보는 함수를 호출할 때 환경 변수로 사용할 수 있습니다. AWS SDK로 작업할 때는 코드에서 직접 SDK의 보안 인증 정보를 제공할 필요가 없습니다. 기본적으로 보안 인증 공급자 체인은 사용자가 보안 인증 정보를 설정할 수 있는 각 위치를 순차적으로 확인하고, 일반적으로 환경 변수(`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` 및 `AWS_SESSION_TOKEN`)인 사용 가능한 첫 번째 위치를 선택합니다.

Lambda는 요청이 실행 환경 내에서 오는 AWS API 요청인 경우 소스 함수 ARN을 보안 인증 컨텍스트에 삽입합니다. 또한 Lambda는 실행 환경 외부에서 Lambda가 사용자를 대신하여 실행하는 다음 AWS API 요청의 소스 함수 ARN를 주입합니다.

Service	작업	이유
CloudWatch Logs	`CreateLogGroup` , `CreateLogStream` , `PutLogEvents`	CloudWatch Logs 로그 그룹에 로그를 저장하려면

Service	작업	이유
X-Ray	`PutTraceSegments`	추적 데이터를 X-Ray로 전송하려면
Amazon EFS	`ClientMount`	함수를 Amazon Elastic File System(Amazon EFS) 파일 시스템에 연결하려면

Lambda가 같은 실행 역할을 사용하여 사용자를 대신하여 실행 환경 외부에서 실행하는 다른 AWS API 호출은 소스 함수 ARN를 포함하지 않습니다. 실행 환경 외부에서의 해당 API 호출의 예는 다음과 같습니다.

- AWS Key Management Service(AWS KMS)을(를) 호출하여 환경 변수를 자동으로 암호화 및 복호화합니다.
- Amazon Elastic Compute Cloud(Amazon EC2)를 호출하여 VPC 지원 함수에 대해 탄력적 네트워크 인터페이스(ENI)를 생성합니다.
- Amazon Simple Queue Service(Amazon SQS)와 같은AWS 서비스를 호출하여 이벤트 소스 매핑으로 설정된 이벤트 소스에서 데이터를 읽습니다.

자격 증명 콘텍스트에서 소스 함수 ARN를 사용하여 리소스에 대한 호출이 특정 Lambda 함수의 코드에서 왔는지 여부를 확인할 수 있습니다. 이를 확인하려면 IAM ID 기반 정책 또는 서비스 제어 정책(SCP)의 `lambda:SourceFunctionArn` 조건 키를 사용하세요.

> **Note**
>
> 리소스 기반 정책의 `lambda:SourceFunctionArn` 조건 키를 사용할 수 없습니다.

자격 증명 기반 정책 또는 SCP에서 이 조건 키를 사용하여 함수 코드가 다른 AWS 서비스에 수행하는 API 작업에 대한 보안 제어를 구현할 수 있습니다. 여기에는 자격 증명 유출의 원인을 식별하는 데 도움이 되는 몇 가지 주요 보안 애플리케이션이 있습니다.

> **Note**
>
> `lambda:SourceFunctionArn` 조건 키가 `lambda:FunctionArn` 및 `aws:SourceArn` 조건 키와 다릅니다. `lambda:FunctionArn` 조건 키는 이벤트 소스 매핑에만 적용되고 이벤

트 소스가 호출할 수 있는 함수를 정의하는 데 도움이 됩니다. `aws:SourceArn` 조건 키는 Lambda 함수가 대상 리소스인 정책에만 적용되며 AWS 서비스와 리소스가 해당 함수를 호출하도록 정의하도록 돕습니다. `lambda:SourceFunctionArn` 조건 키는 모든 자격 증명 기반 정책 또는 SCP에 적용하여 다른 리소스에 대한 특정 AWS API 호출 권한을 가진 특정 Lambda 함수를 정의할 수 있습니다.

정책에서 `lambda:SourceFunctionArn`을(를) 사용하려면 ARN 조건 연산자 중 하나를 포함하는 조건으로 포함하십시오. 키 값은 유효한 ARN이어야 합니다.

예를 들어, Lambda 함수 코드가 특정 Amazon S3 버킷을 대상으로 하는 `s3:PutObject` 호출을 한다고 합시다. 하나의 특정 Lambda 함수만 해당 버킷에 `s3:PutObject` 액세스하게 허용한다고 합시다. 이 경우 함수의 실행 역할에 다음과 같은 정책이 연결되어 있어야 합니다.

Example Amazon S3 리소스에 대한 특정 Lambda 함수 액세스 권한을 부여하는 정책

```
{
    "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"
                }
            }
        }
    ]
}
```

이 정책은 소스가 ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda`을(를) 사용하는 Lambda 함수인 경우만 `s3:PutObject` 액세스를 허용합니다. 이 정책은 다른 호출 자격 증명에 대한 `s3:PutObject` 액세스를 허용하지 않습니다. 이것은 다른 함수 또는 엔티티가 동일한 실행 역할로 `s3:PutObject`을(를) 호출할 때도 마찬가지입니다.

> **Note**
>
> `lambda:SourceFunctionARN` 조건 키는 Lambda 함수 버전 또는 함수 별칭을 지원하지 않습니다. 특정 함수 버전 또는 별칭에 ARN을 사용하는 경우 함수는 지정한 작업을 수행할 권한이 없습니다. 버전 또는 별칭 접미사 없이 함수에 정규화되지 않은 ARN을 사용해야 합니다.

SCP에서도 `lambda:SourceFunctionArn`을 사용할 수 있습니다. 예를 들어 버킷에 대한 액세스를 단일 Lambda 함수의 코드 또는 특정 Amazon Virtual Private Cloud(VPC)의 호출로 제한하려고 한다고 가정해 보겠습니다. 다음 SCP에서 이 방법을 보여줍니다.

Example 특정 조건에서 Amazon S3 대한 액세스를 거부하는 정책

```
{
    "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"
                }
            }
```

```
        }
    ]
}
```

이 정책은 ARN `arn:aws:lambda:*:123456789012:function:source_lambda`을(를) 사용하는 특정 Lambda 함수에서 오지 않는 한 또는 지정된 VPC에서 오지 않는 한 모든 S3 작업을 거부합니다. `StringNotEqualsIfExists` 연산자는 `aws:SourceVpc` 키가 요청에 있는 경우에만 이 조건을 처리하도록 IAM에 지시합니다. 마찬가지로 IAM은 `lambda:SourceFunctionArn`이(가) 있는 경우에만 `ArnNotEqualsIfExists` 연산자를 고려합니다.

다른 AWS 엔터티에 Lambda 함수에 대한 액세스 권한 부여

다른 AWS 계정, 조직 및 서비스에 Lambda 리소스에 액세스할 수 있는 권한을 부여하려는 경우 다음과 같은 몇 가지 옵션이 있습니다.

- ID 기반 정책을 사용하여 다른 사용자에게 Lambda 리소스에 대한 액세스 권한을 부여할 수 있습니다. 자격 증명 기반 정책은 사용자에게 직접 적용하거나, 사용자와 연결된 그룹 및 역할에 적용할 수 있습니다.
- 리소스 기반 정책을 사용하여 다른 계정과 AWS 서비스에 Lambda 리소스를 사용할 권한을 부여할 수 있습니다. 사용자가 Lambda 리소스에 액세스하려고 하면 Lambda는 사용자의 자격 증명 기반 정책 및 리소스의 리소스 기반 정책을 모두 고려합니다. Amazon Simple Storage Service(Amazon S3 같은 AWS 서비스가 Lambda 함수를 호출하면 Lambda는 리소스 기반 정책만 고려합니다.
- ABAC(속성 기반 액세스 제어) 모델을 사용하여 Lambda 함수에 대한 액세스를 제어할 수 있습니다. ABAC를 사용하면 태그를 Lambda 함수에 연결하거나, 특정 API 요청에 전달하거나, 요청을 하는 IAM 보안 주체에 연결할 수 있습니다. 함수 액세스를 제어하려면 IAM 정책의 조건 요소에 동일한 태그를 지정하세요.

최소 권한 액세스에 대한 권한을 미세 조정할 수 있도록 Lambda는 정책에 포함할 수 있는 몇 가지 추가 조건을 제공합니다. 자세한 내용은 the section called “리소스와 조건” 단원을 참조하십시오.

Lambda에서 ID 기반 IAM 정책 작업

AWS Identity and Access Management(IAM)에서 자격 증명 기반 정책을 사용하여 계정의 사용자에게 Lambda에 대한 액세스 권한을 부여할 수 있습니다. 자격 증명 기반 정책은 사용자에게 직접 적용하거나, 사용자와 연결된 그룹 및 역할에 적용할 수 있습니다. 다른 계정의 사용자에게 내 계정의 역할을 수행할 수 있는 권한 및 Lambda 리소스에 대한 액세스 권한을 부여할 수도 있습니다. 이 페이지는 ID 기반 정책을 함수 개발에 사용하는 방법의 예를 보여줍니다.

Lambda는 Lambda API 작업에 대해 액세스 권한을 부여하는 AWS 관리형 정책을 제공합니다. 경우에 따라 이 정책은 Lambda 리소스를 개발하고 관리하는 데 사용되는 AWS 다른 서비스에 대한 액세스 권한도 부여합니다. Lambda는 필요에 따라 이러한 관리형 정책을 업데이트하여 정책 릴리스 시 사용자가 새 기능에 대한 액세스 권한을 가질 수 있도록 보장합니다.

- AWSLambda_FullAccess - Lambda 작업에 대한 모든 액세스 권한 및 Lambda 리소스를 개발 및 유지하는 데 사용되는 다른 AWS 서비스에 대한 모든 액세스 권한을 부여합니다. 이 정책은 이전 정책 AWSLambdaFullAccess의 범위를 줄이는 방식으로 만들어졌습니다.

- AWSLambda_ReadOnlyAccess – Lambda 리소스에 대한 읽기 전용 액세스 권한을 부여합니다. 이 정책은 이전 정책 AWSLambdaReadOnlyAccess의 범위를 줄이는 방식으로 만들어졌습니다.
- AWSLambdaRole – Lambda 함수를 호출할 수 있는 권한을 부여합니다.

AWS 관리형 정책은 사용자가 수정할 수 있는 Lambda 함수나 계층을 제한하지 않고 API 작업에 대한 권한을 부여합니다. 보다 세부적으로 제어하기 위해 사용자의 권한을 제한하는 정책을 직접 만들 수 있습니다.

Sections

사용자에게 함수에 대한 권한을 부여하는 예제 정책 작성

자격 증명 기반 정책을 사용해 사용자에게 Lambda 함수에 대한 작업을 실행할 수 있도록 허용합니다.

> **Note**
>
> 컨테이너 이미지로 정의된 함수의 경우 이미지에 액세스할 수 있는 사용자 권한은 Amazon Elastic Container Registry에서 구성해야 합니다. 예시는 Amazon ECR 권한을 참조하세요.

다음은 범위를 제한한 권한 정책의 예입니다. 사용자가 지정된 접두사(`intern-`)로 이름이 지정되고 지정된 실행 역할로 구성된 Lambda 함수를 생성하고 관리할 수 있도록 허용합니다.

Example 함수 개발 정책

```
{
    "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-*"
        }
    ]
}
```

정책의 권한은 구문이 지원하는 리소스 및 조건에 따라 구문으로 구성됩니다.

- `ReadOnlyPermissions` – 찾아보기 및 보기 기능을 사용할 때 Lambda 콘솔은 이러한 권한을 사용합니다. 이들 권한은 리소스 패턴 또는 조건을 지원하지 않습니다.

```
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
```

- `DevelopFunctions` – `AddPermission` 및 `PutFunctionConcurrency`를 제외하고 `intern-`으로 접두사가 지정된 함수에서 작동하는 Lambda 작업을 사용합니다. `AddPermission`은 함수에 대한 리소스 기반 정책을 수정하여 보안에 영향을 줄 수 있습니다. `PutFunctionConcurrency`는 함수에 대한 조정 용량을 예약하여 다른 함수로부터 용량을 가져올 수 있습니다.

```
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
```

```
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- DevelopEventSourceMappings - intern-으로 접두사가 지정된 함수에 대한 이벤트 소스 매핑을 관리합니다. 이러한 작업은 이벤트 소스 매핑에 대해 작용하지만 조건과 함께 함수를 제한할 수 있습니다.

```
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
```

- PassExecutionRole – IAM 권한을 가진 사용자가 생성하고 관리해야 하는 intern-lambda-execution-role이라는 역할만 조회하고 전달합니다. 함수에 실행 역할을 할당할 경우 PassRole이 사용됩니다.

```
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- ViewLogs – CloudWatch Logs를 사용하여, intern-으로 접두사가 지정된 함수에 대한 로그를 조회합니다.

```
            "Action": [
                "logs:*"
            ],
```

```
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

이 정책은 사용자가 다른 사용자의 리소스를 위험하게 만들지 않고 Lambda를 시작할 수 있도록 합니다. 사용자가 더 광범위한 IAM 권한이 필요한 다른 AWS 서비스에 의해 트리거되는 기능을 구성하거나 호출하는 것은 허용하지 않습니다. 또한 CloudWatch 및 X-Ray와 같이 범위 제한 정책을 지원하지 않는 서비스에 대한 권한은 포함하지 않습니다. 사용자에게 지표 및 추적 데이터에 대한 액세스 권한을 부여하려면 이러한 서비스에 대해 읽기 전용 정책을 사용합니다.

함수에 대한 트리거를 구성할 경우 해당 함수를 호출하는 AWS 서비스를 사용하기 위한 액세스 권한이 필요합니다. 예를 들어, Amazon S3 트리거를 구성하려면 버킷 알림을 관리하는 Amazon S3 작업을 사용할 수 있는 권한이 필요합니다. 이러한 권한의 대부분은 AWSLambdaFullAccess 관리형 정책에 포함됩니다. 예제 정책은 이 설명서의 GitHub 리포지토리에서 사용할 수 있습니다.

계층을 사용할 수 있는 권한을 부여하는 예제 정책 작성

다음 정책은 계층을 만들고 이를 함수에 사용할 수 있는 권한을 사용자에게 부여합니다. 이 리소스 패턴에 따르면 계층 이름이 AWS로 시작되는 한 사용자는 원하는 `test-` 리전에서 원하는 계층 버전으로 작업할 수 있습니다.

Example 계층 개발 정책

```
{
    "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-*:*"
        }
```

```
    ]
}
```

함수 생성 및 구성 중 lambda:Layer 조건을 사용하여 계층 사용을 적용할 수도 있습니다. 예를 들어 다른 계정에서 게시한 계층을 사용자가 사용하지 못하게 할 수 있습니다. 다음 정책은 CreateFunction 및 UpdateFunctionConfiguration 작업에 조건을 추가하여 지정된 계층이 123456789012 계정에 속한 계층이어야 하도록 요구합니다.

```
{
    "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:*:*"
                    ]
                }
            }
        }
    ]
}
```

조건이 적용되도록 하려면, 다른 구문이 이러한 작업에 대한 권한을 사용자에게 부여하지 않는지 확인하세요.

ID 기반 정책으로 크로스 계정 액세스 구현

이전의 정책과 구문을 역할에 적용할 수 있으며, 그런 다음 다른 계정과 공유하여 Lambda 리소스에 대한 액세스 권한을 부여할 수 있습니다. 사용자와 달리, 역할에는 인증을 위한 자격 증명이 없습니다. 대신, 누가 역할을 맡을 수 있고 해당 권한을 사용할 수 있는지를 지정하는 트러스트 정책이 있습니다.

신뢰하는 계정에 교차 계정 역할을 사용하여 Lambda 작업 및 리소스에 대한 액세스 권한을 부여할 수 있습니다. 함수를 호출하거나 계층을 사용하는 권한만 부여하려면 리소스 기반 정책을 사용하세요.

자세한 내용은 IAM 사용 설명서에서 IAM 역할을 참조하세요.

Lambda에서 리소스 기반 정책 작업

Lambda는 Lambda 함수 및 계층에 대해 리소스 기반 권한 정책을 지원합니다. 리소스 기반 정책을 사용하여 리소스별로 다른 AWS 계정 또는 조직에 사용 권한을 부여할 수 있습니다. 리소스 기반 정책을 사용하여 AWS 서비스가 자동으로 해당 함수를 호출할 수 있도록 허용할 수도 있습니다.

Lambda 함수의 경우 함수를 호출하거나 관리하기 위한 권한을 계정에 부여할 수 있습니다. 또한 단일 리소스 기반 정책을 사용하여 AWS Organizations의 전체 조직에 사용 권한을 부여할 수 있습니다. 계정에서의 활동에 대한 응답으로 함수를 호출하는 AWS 서비스에 호출 권한을 부여하도록 리소스 기반 정책을 사용할 수도 있습니다.

함수의 리소스 기반 정책을 보려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 권한(Permissions)을 선택합니다.
4. 리소스 기반 정책(Resource-based policy)까지 아래로 스크롤한 다음 정책 문서 보기(View policy document)를 선택합니다. 리소스 기반 정책에는 다른 계정 또는 AWS 서비스가 해당 함수에 액세스하려고 할 때 적용되는 권한이 표시됩니다. 다음 예제에서는 Amazon S3가 123456789012 계정에서 DOC-EXAMPLE-BUCKET 버킷의 my-function이라는 함수를 호출할 수 있도록 허용하는 명령문을 보여 줍니다.

   Example 리소스 기반 정책

```
{
    "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 계층의 경우 전체 계층 대신 특정 계층 버전에서만 리소스 기반 정책을 사용할 수 있습니다. 단일 계정 또는 여러 계정에 권한을 부여하는 정책 외에도 계층에 대해 조직의 모든 계정에 권한을 부여할 수도 있습니다.

> **Note**
>
> AddPermission 및 AddLayerVersionPermission API 작업의 범위 내에 있는 Lambda 리소스에 대한 리소스 기반 정책만 업데이트할 수 있습니다. 현재, Lambda 리소스에 대한 정책을 JSON으로 작성할 수 없으며, 해당 작업에 대한 파라미터로 매핑되지 않는 조건은 사용할 수 없습니다.

리소스 기반 정책은 한 버전의 함수, 버전, 별칭 또는 계층에 적용됩니다. 이 정책은 하나 이상의 서비스 및 계정에 권한을 부여합니다. 여러 리소스에 대한 액세스 권한을 갖게 하거나 리소스 기반 정책이 지원하지 않는 API 작업을 사용하게 하려는 신뢰할 수 있는 계정의 경우, 교차 계정 역할을 사용할 수 있습니다.

주제

- 지원되는 API 작업
- 함수에 AWS 서비스에 대한 액세스 권한 부여
- 조직에 함수 액세스 권한 부여
- 함수에 다른 계정에 대한 액세스 권한 부여
- 계층에 다른 계정에 대한 액세스 권한 부여
- 리소스 기반 정책 정리

지원되는 API 작업

다음 Lambda API 작업은 리소스 기반 정책을 지원합니다.

- 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

함수에 AWS 서비스에 대한 액세스 권한 부여

AWS 서비스를 사용해 함수를 호출하는 경우 리소스 기반 정책의 문에서 권한을 부여하십시오. 호출하거나 관리할 전체 함수에 문을 적용하거나, 문을 단일 버전 또는 별칭으로 제한할 수 있습니다.

> **Note**
>
> Lambda 콘솔을 사용하여 함수에 트리거를 추가하는 경우, 콘솔은 함수의 리소스 기반 정책을 업데이트하여 서비스가 함수를 호출할 수 있도록 합니다. Lambda 콘솔에서 사용할 수 없는 다른 계정이나 서비스에 권한을 부여하기 위해 AWS CLI를 사용할 수 있습니다.

`add-permission` 명령을 사용하여 구문을 추가합니다. 가장 간단한 리소스 기반 정책 구문은 서비스가 함수를 호출할 수 있도록 허용합니다. 다음 명령은 `my-function`이라는 함수를 호출할 수 있는 권한을 Amazon SNS에 부여합니다.

```
aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --
statement-id sns \
    --principal sns.amazonaws.com --output text
```

다음 결과가 표시됩니다:

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

이렇게 하면 Amazon SNS가 함수에 대해 `lambda:Invoke` API를 호출하지만, 호출을 트리거하는 Amazon SNS 주제를 제한하지는 않습니다. 함수를 특정 리소스에서만 호출하도록 하려면, `source-arn` 옵션을 사용하여 리소스의 Amazon 리소스 이름(ARN)을 지정합니다. 다음 명령은 Amazon SNS가 `my-topic`이라는 주제 구독에 대해서만 함수를 호출할 수 있도록 허용합니다.

```
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
```

일부 서비스는 다른 계정의 함수를 호출할 수 있습니다. 해당 계정 ID를 소유하는 소스 ARN을 지정하면 문제가 되지 않습니다. 하지만 Amazon S3의 경우 소스는 ARN에 버킷의 계정 ID가 포함되지 않은

버킷입니다. 버킷을 삭제하고, 다른 계정에서 동일한 이름으로 버킷을 생성할 수 있습니다. 계정의 리소스만 함수를 호출할 수 있게 하려면 계정 ID와 함께 source-account 옵션을 사용합니다.

```
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
```

조직에 함수 액세스 권한 부여

AWS Organizations의 조직에 권한을 부여하려면 조직 ID를 principal-org-id로 지정합니다. 다음 AddPermission AWS CLI 명령은 o-a1b2c3d4e5f 조직의 모든 사용자에게 호출 액세스 권한을 부여합니다.

```
aws lambda add-permission --function-name example \
    --statement-id PrincipalOrgIDExample --action lambda:InvokeFunction \
    --principal * --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> 이 명령에서 Principal은 *입니다. 즉, o-a1b2c3d4e5f 조직의 모든 사용자에게 함수 호출 권한이 부여됩니다. AWS 계정 또는 역할을 Principal로 지정하면 해당 보안 주체만 함수 호출 권한을 얻게 됩니다. 단, 보안 주체가 o-a1b2c3d4e5f 조직에도 속하는 경우에만 그렇습니다.

이 명령은 다음과 같은 리소스 기반 정책을 생성합니다.

```
{
    "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"
```

```
                }
            }
        }
    ]
}
```

자세한 내용은 AWS Identity and Access Management 사용 설명서의 aws:PrincipalOrgID 섹션을 참조하세요.

함수에 다른 계정에 대한 액세스 권한 부여

다른 AWS 계정에 권한을 부여하려면 계정 ID를 `principal`로 지정합니다. 다음 예제는 `111122223333` 별칭을 가진 `my-function`을 호출할 수 있는 권한을 `prod` 계정에 부여합니다.

```
aws lambda add-permission --function-name my-function:prod --statement-id xaccount --
action lambda:InvokeFunction \
    --principal 111122223333 --output text
```

다음 결과가 표시됩니다:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-2:123456789012:function:my-function"}
```

리소스 기반 정책은 다른 계정에서 함수에 액세스할 수 있는 권한을 부여하지만, 해당 계정의 사용자가 권한을 초과하도록 허용하지 않습니다. 다른 계정의 사용자는 Lambda API를 사용할 수 있는 해당 사용자 권한이 있어야 합니다.

사용자 또는 다른 계정의 역할에 대한 액세스 제한을 위해서는 ID의 전체 ARN을 보안 주체로 지정합니다. 예: `arn:aws:iam::123456789012:user/developer`.

별칭은 다른 계정에서 호출할 수 있는 버전을 제한합니다. 다른 계정에서 함수 ARN에 별칭을 포함시켜야 합니다.

```
aws lambda invoke --function-name arn:aws:lambda:us-west-2:123456789012:function:my-
function:prod out
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
```

```
    "ExecutedVersion": "1"
}
```

그러면 함수 소유자는 호출자가 함수를 호출하는 방법을 변경할 필요없이 새 버전을 가리키도록 별칭을 업데이트할 수 있습니다. 이렇게 하면 다른 계정은 새 버전을 사용하기 위해 코드를 변경할 필요가 없으며, 별칭과 연결된 함수 버전을 호출할 수 있는 권한만 갖습니다.

기존 함수에서 작업을 수행하는 대부분의 API 작업에 대해 교차 계정 액세스 권한을 부여할 수 있습니다. 예를 들어 계정에서 별칭 목록을 확인할 수 있게 하려면 `lambda:ListAliases`에 대한 액세스를 부여하고, 함수 코드를 다운로드할 수 있게 하려면 `lambda:GetFunction`에 대한 액세스를 부여할 수 있습니다. 각 권한을 별도로 추가하거나, `lambda:*`를 사용하여 특정 함수에 대한 모든 작업에 대한 액세스를 부여합니다.

여러 함수에 대한 권한 또는 함수에서 작업을 수행하지 않는 작업에 대한 권한을 다른 계정에 부여하려면 IAM 역할을 사용하는 것이 좋습니다.

계층에 다른 계정에 대한 액세스 권한 부여

다른 계정에 계층 사용 권한을 부여하려면 add-layer-version-permission 명령을 사용해 해당 계층 버전의 권한 정책에 명령문을 추가합니다. 각 명령문에서 단일 계정, 모든 계정 또는 조직을 대상으로 권한을 부여할 수 있습니다.

다음 예시에서는 111122223333 계정에 `bash-runtime` 계층의 버전 2에 대한 액세스 권한을 부여합니다.

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

다음과 유사한 출력 화면이 표시되어야 합니다.

```
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"}
```

권한은 하나의 계층 버전에만 적용됩니다. 새 계층 버전을 만들 때마다 해당 과정을 반복합니다.

조직 내 모든 계정에 권한을 부여하려면 `organization-id` 옵션을 사용합니다. 다음 예제에서는 버전 3의 계층을 사용할 권한을 조직의 모든 계정에 부여합니다.

```
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
```

다음 결과가 표시됩니다:

```
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"}}}"
```

모든 AWS 계정에 권한을 부여하려면 보안 주체에 대해 *를 사용하고 조직 ID는 생략하세요. 계정 또는 조직이 다수인 경우, 여러 가지 명령문을 추가해야 합니다.

리소스 기반 정책 정리

함수의 리소스 기반 정책을 보려면 `get-policy` 명령을 사용합니다.

```
aws lambda get-policy --function-name my-function --output text
```

다음 결과가 표시됩니다:

```
{"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
```

버전 및 별칭의 경우 버전 번호나 별칭을 함수 이름에 추가합니다.

```
aws lambda get-policy --function-name my-function:PROD
```

함수에서 권한을 제거하려면 `remove-permission`을 사용합니다.

```
aws lambda remove-permission --function-name example --statement-id sns
```

계층에 대한 권한을 보려면 `get-layer-version-policy` 명령을 사용합니다.

```
aws lambda get-layer-version-policy --layer-name my-layer --version-number 3 --output
 text
```

다음 결과가 표시됩니다:

```
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"}}}"
```

정책에서 명령문을 제거하려면 `remove-layer-version-permission`을 사용합니다.

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3 --
statement-id engineering-org
```

Lambda에서 속성 기반 액세스 제어 사용

ABAC(속성 기반 액세스 제어)를 사용하여 Lambda 함수에 대한 액세스를 제어할 수 있습니다. Lambda 함수에 태그를 연결하거나, 특정 API 요청에서 태그를 전달하거나,AWS Identity and Access Management(IAM) 보안 주체에 연결하여 요청할 수 있습니다. AWS 속성 기반 액세스를 부여하는 방법에 대한 자세한 내용은 IAM 사용 설명서에서 태그를 사용하여 AWS 리소스에 대한 액세스 제어를 참조하세요.

IAM 정책에서 Amazon 리소스 이름(ARN) 또는 IAM 정책을 지정하지 않고 ABAC를 사용하여 최소 권한 부여할 수 있습니다. 대신 IAM 정책 조건 요소에서 태그를 지정하여 액세스를 제어할 수 있습니다. ABAC를 사용하면 새 함수를 생성할 때 IAM 정책을 업데이트할 필요가 없으므로 확장이 더 쉬워집니다. 대신 새 함수에 태그를 추가하여 액세스를 제어하십시오.

Lambda에서 태그는 함수 수준에서 작동합니다. 레이어, 코드 서명 구성 또는 이벤트 소스 매핑에는 태그가 지원되지 않습니다. 함수에 태그를 지정하면 해당 태그가 함수와 연결된 모든 버전 및 별칭에 적용됩니다. 함수에 태그를 지정하는 방법은 Lambda 함수에서 태그 사용을(를) 참조하십시오.

다음 조건 키를 사용하여 함수 작업을 제어할 수 있습니다.

- aws:ResourceTag/tag-key: Lambda 함수에 연결된 태그를 기반으로 액세스를 제어합니다.
- aws:RequestTag/tag-key: 새 함수를 만들 때와 같이 요청에 태그가 있어야 합니다.
- aws:PrincipalTag/tag-key: IAM 보안 주체(요청을 하는 사람)이 자신의 IAM 사용자 또는 역할에 연결된 태그를 기반으로 수행할 수 있는 권한을 제어합니다.

- aws:TagKeys: 요청에서 특정 키를 사용할 수 있는지 여부를 제어합니다.

ABAC를 지원하는 Lambda 작업의 전체 목록은 지원되는 함수 작업 단원 및 테이블의 조건(Condition) 열을 참조하십시오.

다음 단계에서는 ABAC를 사용하여 권한을 설정하는 한 가지 방법을 보여 줍니다. 이 예제 시나리오에서는 4개의 IAM 권한 정책을 생성합니다. 그런 다음 이러한 정책을 새 IAM 역할에 연결합니다. 마지막으로 IAM 사용자를 만들고 해당 사용자에게 새 역할을 수임할 권한을 부여합니다.

주제

필수 조건

Lambda 실행 역할이 있는지 확인하십시오. IAM 권한을 부여할 때와 Lambda 함수를 만들 때 이 역할을 사용합니다.

1단계: 새 함수에 태그 필요

Lambda와 함께 ABAC를 사용하는 경우 모든 함수에 태그가 필요한 것이 모범 사례입니다. 이렇게 하면 ABAC 권한 정책이 예상대로 작동할 수 있습니다.

다음과 비슷한 IAM 정책을 생성합니다. 이 정책은 aws:RequestTag/tag-key, aws:ResourceTag/tag-key 및 aws:TagKeys 조건 키를 사용하여 새 함수와 해당 함수를 생성하는 IAM 보안 주체가 모두 `project` 태그를 갖도록 요구합니다. `ForAllValues` 수정자는 `project` 태그만 허용되게 합니다. `ForAllValues` 수정자를 포함하지 않는 경우 사용자는 `project`을(를) 통과하는 한 함수에 다른 태그를 추가할 수 있습니다.

Example - 새 함수에 태그 필요

```
{
  "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"
      }
    }
  }
}
```

2단계: Lambda 함수 및 IAM 보안 주체에 연결된 태그를 기반으로 작업 허용

aws:ResourceTag/tag-key 조건 키를 사용하여 보안 주체의 태그가 함수에 연결된 태그와 일치할 것을 요구하는 두 번째 IAM 정책을 생성합니다.. 다음 예제 정책에서는 `project` 태그가 있는 보안 주체가 `project` 태그가 있는 함수를 호출하도록 허용합니다. 함수에 다른 태그가 있으면 작업이 거부됩니다.

Example — 함수와 IAM 보안 주체에 일치하는 태그 필요

```
{
  "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}"
        }
      }
    }
  ]
}
```

3단계: 목록 권한 부여

보안 주체가 Lambda 함수 및 IAM 역할을 나열할 수 있도록 허용하는 정책을 생성합니다. 이렇게 하면 보안 주체가 콘솔에서 그리고 API 작업을 호출할 때 모든 Lambda 함수와 IAM 역할을 볼 수 있습니다.

Example — Lambda 및 IAM 목록 권한 부여

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllResourcesLambdaNoTags",
      "Effect": "Allow",
      "Action": [
        "lambda:GetAccountSettings",
        "lambda:ListFunctions",
        "iam:ListRoles"
      ],
      "Resource": "*"
    }
  ]
}
```

4단계: IAM 권한 부여

iam:PassRole을 허용하는 정책을 생성합니다. 이 권한은 함수에 실행 역할을 할당할 때 필요합니다. 다음 예제 정책에서는 예제 ARN을 Lambda 실행 역할의 ARN으로 바꿉니다.

> **Note**
>
> `ResourceTag` 작업을 포함하는 정책에는 `iam:PassRole` 조건 키를 사용하지 마세요. IAM 역할에서는 태그를 사용하여 누가 해당 역할을 전달할 수 있는지 액세스 권한을 제어할 수 없습니다. 서비스에 역할을 전달할 수 있는 권한 부여에 대한 자세한 내용은 사용자에게 AWS 서비스에 역할을 전달할 권한 부여를 참조하세요.

Example — 실행 역할을 전달할 수 있는 권한 부여

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
    }
  ]
}
```

5단계 - IAM 역할 생성

역할을 사용하여 권한을 위임하는 모범 사례입니다. `abac-project-role`이라고 하는 IAM 역할 생성:

- 1단계: 신뢰할 수 있는 엔터티 선택(Step 1: Select trusted entity)에서 AWS 계정(account)을 선택한 다음 이 계정(This account)을 선택합니다.
- 2단계: 권한 추가(Step 2: Add permissions)에서는 이전 단계에서 생성한 네 가지 IAM 정책을 연결합니다.
- 3단계: 이름, 검토 및 생성(Step 3: Name, review, and create)에서는 태그 추가(Add tag)를 선택합니다. 키에 `project`를 입력합니다. 값(Value)을 입력하지 마십시오.

6단계: IAM 사용자 생성

`abac-test-user`(이)라고 하는 IAM 사용자를 생성합니다. 권한 설정(Set permissions) 섹션에서 기존 정책 직접 연결(Attach existing policies directly)을 선택한 후 정책 생성(Create policy)을 선택합니다. 다음 정책 정의를 입력합니다. *111122223333*을 AWS 계정 ID로 바꿉니다. 이 정책은 `abac-test-user`이(가) `abac-project-role`을(를) 수임할 수 있게 허용합니다.

Example — IAM 사용자가 ABAC 역할을 수임할 수 있게 허용

```
{
  "Version": "2012-10-17",
  "Statement": {
```

```
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
  }
}
```

7단계: 권한 테스트

1. AWS 콘솔에 abac-test-user(으)로 로그인합니다. 자세한 정보는 IAM 사용자로 로그인을 참조하십시오.
2. abac-project-role 역할로 전환합니다. 자세한 내용은 역할(콘솔) 전환을 참조하십시오.
3. Lambda 함수 생성:
   - 권한(Permissions)에서 기본 실행 역할 변경(Change default execution role)을 선택하고 실행 역할(Execution role)을 선택한 후 기존 역할 사용(Use an existing role)을 선택합니다. 4단계: IAM 권한 부여에서 사용한 것과 동일한 실행 역할을 선택합니다.
   - 고급 설정(Advanced settings)에서 태그 사용(Enable tags)를 선택한 다음 새 태그 추가(Add new tag)를 선택합니다. 키에 project를 입력합니다. 값(Value)을 입력하지 마십시오.
4. 함수를 테스트합니다.
5. 두 번째 Lambda 함수를 생성하고 다른 태그(예: environment)를 추가합니다. 1단계: 새 함수에 태그 필요에서 만든 ABAC 정책은 보안 주체가 project 태그를 사용하여 함수를 만들 수만 있도록 허용하기 때문에 이 작업이 실패합니다.
6. 태그 없이 세 번째 함수를 만듭니다. 1단계: 새 함수에 태그 필요에서 만든 ABAC 정책은 보안 주체가 태그 없이 함수를 만들 수만 있도록 허용하지 않기 때문에 이 작업이 실패합니다.

이 권한 부여 전략을 사용하면 각 새 사용자에 대해 새 정책을 만들지 않고도 액세스를 제어할 수 있습니다. 새 사용자에게 액세스 권한을 부여하려면 할당된 프로젝트에 해당하는 역할을 수임할 권한을 부여하기만 하면 됩니다.

8단계: 리소스 정리

IAM 역할 삭제

1. IAM 콘솔에서 역할 페이지를 엽니다.
2. 5단계에서 생성한 역할을 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 삭제를 확인하려면 텍스트 입력 필드에 역할 이름을 입력합니다.

5. Delete(삭제)를 선택합니다.

IAM 사용자를 삭제하려면 다음을 수행하세요.

1. IAM 콘솔의 사용자 페이지를 엽니다.
2. 6단계에서 생성한 IAM 사용자를 선택합니다.
3. Delete(삭제)를 선택합니다.
4. 삭제를 확인하려면 텍스트 입력 필드에 사용자 이름을 입력합니다.
5. Delete user(사용자 삭제)를 선택합니다.

Lambda 함수를 삭제하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 생성한 함수를 선택합니다.
3. 작업, 삭제를 선택합니다.
4. 텍스트 입력 필드에 **delete**를 입력하고 Delete(삭제)를 선택합니다.

정책의 리소스 및 조건 섹션 미세 조정

AWS Identity and Access Management(IAM) 정책에 리소스와 조건을 지정하여 사용자 권한의 범위를 제한할 수 있습니다. 정책에서 각각의 작업은 해당 작업의 동작에 따라 다양한 조합의 리소스와 조건을 지원합니다.

각 IAM 정책 구문은 리소스에 대해 수행되는 작업에 대한 권한을 부여합니다. 지명된 리소스에서 이루어지는 작업이 아니거나 모든 리소스에 대해 그 작업을 수행할 수 있도록 권한을 부여하는 경우, 정책에서 해당 리소스의 값은 와일드카드(*)가 됩니다. 대부분의 작업에서는 리소스의 Amazon 리소스 이름(ARN) 또는 복수의 리소스에 맞는 ARN 패턴을 지정함으로써 사용자 수정이 가능한 리소스를 제한할 수 있습니다.

리소스별 권한을 제한하려면 ARN으로 리소스를 지정하세요.

Lambda 리소스 ARN 형식

- 함수 - `arn:aws:lambda:us-west-2:123456789012:function:my-function`
- 함수 버전 - `arn:aws:lambda:us-west-2:123456789012:function:my-function:1`
- 함수 별칭 - `arn:aws:lambda:us-west-2:123456789012:function:my-function:TEST`

- 이벤트 소스 매핑 - arn:aws:lambda:*us-west-2*:*123456789012*:event-source-mapping:*fa123456-14a1-4fd2-9fec-83de64ad683de6d47*
- 계층 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*
- 계층 버전 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*:*1*

예를 들어 다음 정책을 통해 AWS 계정 123456789012 사용자는 미국 서부(오리건) AWS 리전에서 my-function이라는 함수를 호출할 수 있습니다.

Example 함수 정책 호출

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

이것은 작업 ID(lambda:InvokeFunction)가 API 작업(호출)과 다른 특수한 경우입니다. 다른 작업의 경우, 작업 이름 앞에 lambda:를 붙인 것이 작업 ID입니다.

Sections

- 정책의 조건 섹션 이해
- 정책의 리소스 섹션에서 함수 참조
- 지원되는 함수 작업
- 지원되는 이벤트 소스 매핑 작업
- 지원되는 계층 작업

정책의 조건 섹션 이해

조건이란 허용되는 작업인지 여부를 판단하기 위해 로직을 추가로 적용하는 선택적 정책 요소를 말합니다. 모든 작업에서 지원되는 일반 조건 외에, Lambda는 일부 작업에서 추가 파라미터 값을 제한하는 데 사용할 수 있는 조건 유형도 정의합니다.

예를 들어, `lambda:Principal` 조건으로는 사용자가 함수의 리소스 기반 정책에 대한 호출 액세스 권한을 부여할 수 있는 서비스 또는 계정을 제한합니다. 다음 정책에 따르면 사용자는 `test` 함수 호출 권한을 Amazon Simple Notification Service(SNS) 주제에 부여할 수 있습니다.

Example 함수 정책 권한 관리

```
{
    "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"
                }
            }
        }
    ]
}
```

이 조건은 보안 주체가 다른 서비스 또는 계정이 아닌 Amazon SNS여야 합니다. 리소스 패턴에 따르면 함수 이름은 `test`이고 버전 번호나 별칭이 포함되어야 합니다. 예: `test:v1`.

Lambda 및 기타 AWS 서비스의 리소스 및 조건에 대한 자세한 내용은 서비스 승인 참조에서 AWS 서비스에 사용되는 작업, 리소스 및 조건 키를 참조하세요.

정책의 리소스 섹션에서 함수 참조

Amazon 리소스 이름(ARN)을 사용하여 정책 설명에서 Lambda 함수를 참조합니다. 함수 ARN의 형식은 전체 함수를 참조(비정규화)하는지, 함수 버전 또는 별칭을 참조(정규화)하는지에 따라 다릅니다.

Lambda API를 호출할 때 사용자는 GetFunction `FunctionName` 파라미터의 버전 ARN 또는 별칭 ARN을 전달하거나 GetFunction `Qualifier` 파라미터의 값을 설정하여 버전 또는 별칭을 지정할 수 있습니다. Lambda는 IAM 정책의 리소스 요소와 API 호출로 전달된 `FunctionName`과 `Qualifier`를 비교하여 권한 부여를 결정합니다. 불일치가 있는 경우 Lambda는 요청을 거부합니다.

함수에 대한 작업을 허용하든지 거부하든지 간에 예상한 결과를 얻으려면 정책 설명에서 올바른 함수 ARN 유형을 사용해야 합니다. 예를 들어, 정책이 정규화되지 않은 ARN을 참조하는 경우 Lambda는 정규화되지 않은 ARN을 참조하는 요청을 수락하지만 정규화된 ARN을 참조하는 요청은 거부합니다.

> **Note**
>
> 와일드카드 문자(*)를 사용하여 계정 ID와 일치시킬 수 없습니다. 허용되는 정책에 대한 자세한 내용은 IAM 사용 설명서의 IAM JSON 정책 참조를 참조하세요.

Example 정규화되지 않은 ARN의 호출 허용

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction"
        }
    ]
}
```

정책이 정규화된 특정 ARN을 참조하는 경우, Lambda는 해당 ARN을 참조하는 요청을 수락하지만 `myFunction:2`와 같이 정규화되지 않은 ARN 또는 정규화된 다른 ARN을 참조하는 요청은 거부합니다.

Example 정규화된 특정 ARN의 호출 허용

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:1"
```

```
        }
    ]
}
```

정책이 :*를 사용하는 정규화된 ARN을 참조하는 경우, Lambda는 정규화된 모든 ARN을 수락하지만 정규화되지 않은 ARN을 참조하는 요청은 거부합니다.

Example 정규화된 모든 ARN의 호출 허용

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

정책이 *를 사용하는 모든 ARN을 참조하는 경우, Lambda는 정규화된 ARN 또는 정규화되지 않은 ARN을 수락합니다.

Example 정규화된 ARN 또는 정규화되지 않은 ARN의 호출 허용

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:myFunction*"
        }
    ]
}
```

지원되는 함수 작업

아래 표에 설명된 것과 같이, 함수에 대한 작업도 특정 함수에만 실행되도록 함수, 버전, 별칭 ARN 등으로 제한할 수 있습니다. 리소스 제한이 지원되지 않는 작업은 모든 리소스(*)를 대상으로 부여할 수 있습니다.

함수 작업

작업	리소스	Condition
AddPermission RemovePermission	함수 함수 버전 함수 별칭	lambda:Principal aws:ResourceTag/${TagKey} lambda:FunctionUrlAuthType
Invoke 권한: lambda:InvokeFunction	함수 함수 버전 함수 별칭	aws:ResourceTag/${TagKey} lambda:EventSourceToken
CreateFunction	함수	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds lambda:SecurityGroupIds aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys
UpdateFunctionConfiguration	함수	lambda:CodeSigningConfigArn lambda:Layer lambda:VpcIds lambda:SubnetIds

작업	리소스	Condition

		`lambda:SecurityGroupIds`
		
		`aws:ResourceTag/${TagKey}`

작업	리소스	Condition
CreateAlias	함수	`aws:ResourceTag/${TagKey}`
DeleteAlias		
DeleteFunction		
DeleteFunctionCodeSigningConfig		
DeleteFunctionConcurrency		
GetAlias		
GetFunction		
GetFunctionCodeSigningConfig		
GetFunctionConcurrency		
GetFunctionConfiguration		
GetPolicy		
ListProvisionedConcurrencyConfigs		
ListAliases		
ListTags		
ListVersionsByFunction		
PublishVersion		
PutFunctionCodeSigningConfig		
PutFunctionConcurrency		
UpdateAlias		
UpdateFunctionCode		

작업	리소스	Condition
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	함수 함수 별칭	lambda:FunctionUrlAuthType lambda:FunctionArn aws:ResourceTag/${TagKey}
ListFunctionUrlConfigs	함수	lambda:FunctionUrlAuthType
DeleteFunctionEventInvokeConfig GetFunctionEventInvokeConfig ListFunctionEventInvokeConfigs PutFunctionEventInvokeConfig UpdateFunctionEventInvokeConfig	함수	aws:ResourceTag/${TagKey}
DeleteProvisionedConcurrencyConfig GetProvisionedConcurrencyConfig PutProvisionedConcurrencyConfig	함수 별칭 함수 버전	aws:ResourceTag/${TagKey}
GetAccountSettings ListFunctions	*	None
TagResource	함수	aws:ResourceTag/${TagKey} aws:RequestTag/${TagKey} aws:TagKeys
UntagResource	함수	aws:ResourceTag/${TagKey} aws:TagKeys

지원되는 이벤트 소스 매핑 작업

이벤트 소스 매핑에서는 삭제 및 업데이트 권한을 특정 이벤트 소스로 제한할 수 있습니다. 사용자가 이벤트 소스를 호출하기 위해 구성할 수 있는 함수를 `lambda:FunctionArn` 조건으로 제한할 수 있습니다.

이러한 작업에서는 리소스가 이벤트 소스 매핑이며, 따라서 해당 이벤트 소스 매핑으로 호출되는 함수에 따라 Lambda의 조건으로 권한을 제어할 수 있습니다.

이벤트 소스 매핑 작업

작업	리소스	Condition
DeleteEventSourceMapping UpdateEventSourceMapping	이벤트 소스 매핑	`lambda:FunctionArn`
CreateEventSourceMapping GetEventSourceMapping	*	`lambda:FunctionArn`
ListEventSourceMappings	*	None

지원되는 계층 작업

계층 작업으로 사용자가 함수에 사용하거나 관리할 수 있는 계층을 제한할 수 있습니다. 계층 사용과 관련된 작업 및 권한은 계층의 버전에 적용되며, `PublishLayerVersion`은 계층 이름에 적용됩니다. 어느 쪽이든 와일드카드를 사용하여 사용자가 작업할 수 있는 계층을 이름별로 제한할 수 있습니다.

> **Note**
>
> GetLayerVersion은 작업은 GetLayerVersionByArn도 포괄합니다. Lambda는 IAM 작업으로 `GetLayerVersionByArn`을 지원하지 않습니다.

계층 작업

작업	리소스	Condition
AddLayerVersionPermission	계층 버전	없음

작업	리소스	Condition
RemoveLayerVersionPermission GetLayerVersion GetLayerVersionPolicy DeleteLayerVersion		
ListLayerVersions PublishLayerVersion	계층	없음
ListLayers	*	없음

AWS Lambda의 보안

AWS에서는 클라우드 보안을 가장 중요하게 생각합니다. 여러분은 AWS 고객으로서 보안에 민감한 기관의 요구 사항을 충족하도록 구축된 데이터 센터 및 네트워크 아키텍처의 혜택을 누릴 수 있습니다.

보안은 AWS와(과) 귀하의 공동 책임입니다. 공동 책임 모델은 이 사항을 클라우드 내 보안 및 클라우드의 보안으로 설명합니다.

- 클라우드의 보안 - AWS는 AWS클라우드에서 AWS서비스를 실행하는 인프라를 보호합니다.AWS는 또한 안전하게 사용할 수 있는 서비스를 제공합니다. 타사 감사원은 정기적으로 AWS 규제 준수 프로그램의 일환으로 보안 효과를 테스트하고 검증합니다. AWS Lambda에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 규정 준수 프로그램 제공 범위 내의 AWS 서비스를 참조하십시오.
- 클라우드 내 보안 – 귀하의 책임은 사용하는 AWS 서비스에 따라 결정됩니다. 또한 귀하는 데이터의 민감도, 회사 요구 사항, 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 Lambda 사용 시 책임 분담 모델을 적용하는 방법을 이해하는 데 도움이 됩니다. 다음 항목에서는 보안 및 규정 준수 목표를 충족하도록 Lambda를 구성하는 방법을 보여줍니다. 또한 Lambda 리소스를 모니터링하고 보호하는 데 도움이 되는 다른 AWS 서비스를 사용하는 방법을 알아봅니다.

Lambda 애플리케이션에 보안 원칙을 적용하는 방법에 대한 자세한 내용은 Serverless Land의 Security를 참조하세요.

주제

- AWS Lambda의 데이터 보호
- AWS Lambda의 ID 및 액세스 관리
- Lambda 함수 및 계층에 대한 거버넌스 전략 생성
- AWS Lambda의 규정 준수 확인
- AWS Lambda의 복원성
- AWS Lambda에서 인프라 보안

AWS Lambda의 데이터 보호

AWS 공동 책임 모델은 AWS Lambda의 데이터 보호에 적용됩니다. 이 모델에서 설명하는 것처럼 AWS는 모든 AWS 클라우드를 실행하는 글로벌 인프라를 보호할 책임이 있습니다. 이 인프라에서 호

스팅되는 콘텐츠에 대한 제어를 유지하는 것은 사용자의 책임입니다. 사용하는 AWS 서비스의 보안 구성과 관리 작업에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 대한 자세한 내용은 데이터 프라이버시 FAQ를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 AWS 보안 블로그의 AWS 공동 책임 모델 및 GDPR 블로그 게시물을 참조하세요.

데이터를 보호하려면 AWS 계정 보안 인증 정보를 보호하고 AWS IAM Identity Center 또는(AWS Identity and Access ManagementIAM)를 통해 개별 사용자 계정을 설정하는 것이 좋습니다. 이 방식을 사용하면 각 사용자에게 자신의 직무를 충실히 이행하는 데 필요한 권한만 부여됩니다. 또한 다음과 같은 방법으로 데이터를 보호하는 것이 좋습니다.

- 각 계정에 다중 인증(MFA)을 사용합니다.
- SSL/TLS를 사용하여 AWS리소스와 통신합니다. TLS 1.2가 필수이며 TLS 1.3을 권장합니다.
- AWS CloudTrail(으)로 API 및 사용자 활동 로깅을 설정합니다.
- AWS 암호화 솔루션을 AWS 서비스 내의 모든 기본 보안 제어와 함께 사용합니다.
- Amazon Macie와 같은 고급 관리형 보안 서비스를 사용하여 Amazon S3에 저장된 민감한 데이터를 검색하고 보호합니다.
- 명령줄 인터페이스 또는 API를 통해 AWS에 액세스할 때 FIPS 140-2 검증된 암호화 모듈이 필요한 경우 FIPS 엔드포인트를 사용합니다. 사용 가능한 FIPS 엔드포인트에 대한 자세한 내용은 Federal Information Processing Standard(FIPS) 140-2를 참조하십시오.

고객의 이메일 주소와 같은 기밀 정보나 중요한 정보는 태그나 이름 필드와 같은 자유 양식 필드에 입력하지 않는 것이 좋습니다. 여기에는 Lambda 또는 기타 AWS 서비스에서 콘솔, API, AWS CLI 또는 AWS SDK를 사용하여 작업하는 경우가 포함됩니다. 이름에 사용되는 태그 또는 자유 형식 텍스트 필드에 입력하는 모든 데이터는 청구 또는 진단 로그에 사용될 수 있습니다. 외부 서버에 URL을 제공할 때 해당 서버에 대한 요청을 검증하기 위해 보안 인증 정보를 URL에 포함시켜서는 안 됩니다.

단원

- 전송 중 데이터 암호화
- 저장 중 암호화

전송 중 데이터 암호화

Lambda API 엔드포인트는 HTTPS를 통한 보안 연결만을 지원합니다. AWS Management Console, AWS SDK 또는 Lambda API를 사용하여 Lambda 리소스를 관리하면 모든 통신이 TLS(전송 계층 보안)로 암호화됩니다. API 엔드포인트의 전체 목록은 AWS 일반 참조의 AWS 리전 및 엔드포인트를 참조하세요.

함수를 파일 시스템에 연결하면 Lambda에서는 모든 연결에 대해 전송 중 암호화를 사용합니다. 자세한 내용은 Amazon Elastic File System 사용 설명서에서 Amazon EFS의 데이터 암호화를 참조하세요.

환경 변수를 사용하면, 콘솔 암호화 도우미가 클라이언트 측 암호화를 사용하여 전송 중인 환경 변수를 보호하도록 할 수 있습니다. 자세한 내용은 Lambda 환경 변수 보안 단원을 참조하십시오.

저장 중 암호화

Lambda는 유휴 상태에서 환경 변수를 항시 암호화합니다. 기본적으로 Lambda는 계정에 생성한 AWS KMS key를 사용하여 환경 변수를 암호화합니다. 이 AWS 관리형 키의 이름은 `aws/lambda`입니다.

원한다면 함수별로 Lambda를 구성하여 기본 AWS 관리형 키 대신에 고객 관리형 키를 사용하여 환경 변수를 암호화할 수 있습니다. 자세한 내용은 Lambda 환경 변수 보안 단원을 참조하십시오.

Lambda는 배포 패키지와 계층 아카이브 등 사용자가 Lambda에 업로드하는 파일을 항상 암호화합니다.

Amazon CloudWatch Logs 및 AWS X-Ray의 경우에도 데이터를 기본적으로 암호화하며, 고객 관리형 키를 사용하도록 구성할 수도 있습니다. 자세한 내용은 CloudWatch Logs의 로그 데이터 암호화와 AWS X-Ray의 데이터 보호를 참조하세요.

AWS Lambda의 ID 및 액세스 관리

AWS Identity and Access Management(IAM)은 관리자가 AWS 리소스에 대한 액세스를 안전하게 제어할 수 있도록 지원하는 AWS 서비스입니다. IAM 관리자는 어떤 사용자가 Lambda 리소스를 사용할 수 있는 인증(로그인) 및 권한(권한 있음)을 받을 수 있는지 제어합니다. IAM은 추가 비용 없이 사용할 수 있는 AWS 서비스입니다.

주제

- 고객
- 자격 증명을 통한 인증
- 정책을 사용한 액세스 관리
- AWS Lambda에서 IAM을 사용하는 방식
- AWS Lambda에 대한 자격 증명 기반 정책 예시
- AWS Lambda의 AWS 관리형 정책
- AWS Lambda 보안 인증 및 액세스 문제 해결

고객

AWS Identity and Access Management(IAM)를 사용하는 방법은 Lambda에서 수행하는 작업에 따라 달라집니다.

서비스 사용자 - Lambda 서비스를 사용하여 작업을 수행하는 경우 필요한 자격 증명과 권한을 관리자가 제공합니다. 더 많은 Lambda 기능을 사용하여 작업을 수행하게 되면 추가 권한이 필요할 수 있습니다. 액세스 권한 관리 방식을 이해하면 적절한 권한을 관리자에게 요청할 수 있습니다. Lambda의 기능에 액세스할 수 없는 경우 AWS Lambda 보안 인증 및 액세스 문제 해결 단원을 참조하세요.

서비스 관리자 - 회사에서 Lambda 리소스를 책임지고 있는 경우 Lambda에 대한 전체 액세스 권한을 가지고 있을 것입니다. 서비스 관리자는 서비스 사용자가 액세스해야 하는 Lambda 기능과 리소스를 결정합니다. 그런 다음, IAM 관리자에게 요청을 제출하여 서비스 사용자의 권한을 변경해야 합니다. 이 페이지의 정보를 검토하여 IAM의 기본 개념을 이해하세요. 회사가 Lambda에서 IAM을 사용하는 방법에 대해 자세히 알아보려면 AWS Lambda에서 IAM을 사용하는 방식 단원을 참조하세요.

IAM 관리자 - IAM 관리자라면 Lambda에 대한 액세스 권한 관리 정책 작성 방법을 자세히 알고 싶을 것입니다. IAM에서 사용할 수 있는 Lambda 자격 증명 기반 정책 예제를 보려면 AWS Lambda에 대한 자격 증명 기반 정책 예시 단원을 참조하세요.

자격 증명을 통한 인증

인증은 ID 자격 증명을 사용하여 AWS에 로그인하는 방식입니다. AWS 계정 루트 사용자이나 IAM 사용자로 또는 IAM 역할을 수임하여 인증(AWS에 로그인)되어야 합니다.

자격 증명 소스를 통해 제공된 보안 인증 정보를 사용하여 페더레이션형 ID로 AWS에 로그인할 수 있습니다. AWS IAM Identity Center (IAM Identity Center) 사용자, 회사의 Single Sign-On 인증, Google 또는 Facebook 보안 인증이 페더레이션형 ID의 예입니다. 페더레이션형 ID로 로그인할 때 관리자가 이전에 IAM 역할을 사용하여 ID 페더레이션을 설정했습니다. 페더레이션을 사용하여 AWS에 액세스하면 간접적으로 역할을 수임합니다.

사용자 유형에 따라 AWS Management Console 또는 AWS 액세스 포털에 로그인할 수 있습니다. AWS에 로그인하는 방법에 대한 자세한 내용은 AWS 로그인 사용 설명서의 AWS 계정에 로그인하는 방법을 참조하십시오.

AWS에 프로그래밍 방식으로 액세스하는 경우, AWS에서는 보안 인증 정보를 사용하여 요청에 암호화 방식으로 서명할 수 있는 소프트웨어 개발 키트(SDK) 및 명령줄 인터페이스(CLI)를 제공합니다. AWS 도구를 사용하지 않는 경우 요청에 직접 서명해야 합니다. 권장 방법을 사용하여 요청에 직접 서명하는 방법에 대한 자세한 내용은 IAM 사용 설명서의 AWS API 요청에 서명을 참조하십시오.

사용하는 인증 방법에 상관없이 추가 보안 정보를 제공해야 할 수도 있습니다. 예를 들어, AWS는 다중 인증(MFA)을 사용하여 계정의 보안을 강화하는 것을 권장합니다. 자세한 내용은 AWS IAM Identity Center 사용 설명서의 다중 인증 및 IAM 사용 설명서의 AWS에서 다중 인증(MFA) 사용을 참조하십시오.

AWS 계정 루트 사용자

AWS 계정를 생성할 때는 해당 계정의 모든 AWS 서비스 및 리소스에 대해 완전한 액세스 권한이 있는 단일 로그인 ID로 시작합니다. 이 자격 증명은 AWS 계정루트 사용자라고 하며, 계정을 생성할 때 사용한 이메일 주소와 암호로 로그인하여 액세스합니다. 일상적인 작업에는 루트 사용자를 가급적 사용하지 않는 것이 좋습니다. 루트 사용자 보안 인증 정보를 보호하고 루트 사용자만 수행할 수 있는 작업을 수행하는 데 이 정보를 사용합니다. 루트 사용자로 로그인해야 하는 태스크의 전체 목록은 IAM 사용자 안내서의 루트 사용자 보안 인증이 필요한 태스크를 참조하십시오.

페더레이션 ID

가장 좋은 방법은 관리자 액세스가 필요한 사용자를 포함한 사용자가 자격 증명 공급자와의 페더레이션을 통해 임시 보안 인증을 사용하여 AWS 서비스에 액세스하도록 요구하는 것입니다.

연동 자격 증명은 엔터프라이즈 사용자 디렉터리, 웹 자격 증명 공급자, AWS Directory Service, Identity Center 디렉터리의 사용자 또는 자격 증명 소스를 통해 제공된 자격 증명을 사용하여 AWS 서비스에 액세스하는 모든 사용자입니다. 페더레이션 보안 인증 정보는 AWS 계정에 액세스할 때 역할을 수임하고 역할은 임시 보안 인증 정보를 제공합니다.

중앙 집중식 액세스 관리를 위해 AWS IAM Identity Center을 사용하는 것이 좋습니다. IAM Identity Center에서 사용자 및 그룹을 생성하거나 모든 AWS 계정및 애플리케이션에서 사용하기 위해 고유한 자격 증명 소스의 사용자 및 그룹 집합에 연결하고 동기화할 수 있습니다. IAM Identity Center에 대한 자세한 내용은 AWS IAM Identity Center 사용 설명서에서 IAM Identity Center란 무엇입니까?를 참조하십시오.

IAM 사용자 및 그룹

IAM 사용자는 단일 개인 또는 애플리케이션에 대한 특정 권한을 가지고 있는 AWS 계정내 자격 증명입니다. 가능하다면 암호 및 액세스 키와 같은 장기 보안 인증 정보가 있는 IAM 사용자를 생성하는 대신, 임시 보안 인증 정보를 사용하는 것이 좋습니다. 하지만 IAM 사용자의 장기 보안 인증 정보가 필요한 특정 사용 사례가 있는 경우, 액세스 키를 교체하는 것이 좋습니다. 자세한 내용은 IAM 사용 설명서의 장기 보안 인증이 필요한 사용 사례의 경우 정기적으로 액세스 키 교체를 참조하십시오.

IAM 그룹은 IAM 사용자 컬렉션을 지정하는 자격 증명입니다. 귀하는 그룹으로 로그인할 수 없습니다. 그룹을 사용하여 여러 사용자의 권한을 한 번에 지정할 수 있습니다. 그룹을 사용하면 대규모 사용자

집합의 권한을 더 쉽게 관리할 수 있습니다. 예를 들어, IAMAdmins(이)라는 그룹이 있고 이 그룹에 IAM 리소스를 관리할 권한을 부여할 수 있습니다.

사용자는 역할과 다릅니다. 사용자는 한 사람 또는 애플리케이션과 고유하게 연결되지만, 역할은 해당 역할이 필요한 사람이라면 누구나 수임할 수 있습니다. 사용자는 영구적인 장기 보안 인증 정보를 가지고 있지만, 역할은 임시 보안 인증 정보만 제공합니다. 자세한 정보는 IAM 사용 설명서의 IAM 사용자를 만들어야 하는 경우(역할이 아님)를 참조하십시오.

IAM 역할

IAM 역할은 특정 권한을 가지고 있는 AWS 계정계정 내 ID입니다. IAM 사용자와 유사하지만, 특정 개인과 연결되지 않습니다. 역할 전환하여 AWS Management Console에서 IAM 역할을 임시로 수임할 수 있습니다. AWS CLI 또는 AWSAPI 태스크를 호출하거나 사용자 지정 URL을 사용하여 역할을 수임할 수 있습니다. 역할 사용 방법에 대한 자세한 정보는 IAM 사용 설명서의 IAM 역할 사용을 참조하십시오.

임시 보안 인증 정보가 있는 IAM 역할은 다음과 같은 상황에서 유용합니다.

- 페더레이션 사용자 액세스 - 페더레이션형 ID에 권한을 부여하려면 역할을 생성하고 해당 역할의 권한을 정의합니다. 페더레이션형 ID가 인증되면 이 ID는 역할과 연결되며 역할에 의해 정의된 권한이 부여됩니다. 페더레이션 역할에 대한 자세한 내용은 IAM 사용 설명서의 Creating a role for a third-party Identity Provider(서드 파티 자격 증명 공급자의 역할 만들기) 부분을 참조하십시오. IAM Identity Center를 사용하는 경우 권한 세트를 구성합니다. 인증 후 보안 인증 정보에서 액세스할 수 있는 항목을 제어하기 위해 IAM Identity Center는 권한 집합을 IAM의 역할과 연결합니다. 권한 세트에 대한 자세한 내용은 AWS IAM Identity Center 사용 설명서의 권한 세트를 참조하십시오.
- 임시 IAM 사용자 권한 - IAM 사용자 또는 역할은 IAM 역할을 수임하여 특정 작업에 대한 다양한 권한을 임시로 받을 수 있습니다.
- 크로스 계정 액세스: IAM 역할을 사용하여 다른 계정의 사용자(신뢰할 수 있는 보안 주체)가 내 계정의 리소스에 액세스하도록 허용할 수 있습니다. 역할은 계정 간 액세스를 부여하는 기본적인 방법입니다. 그러나 일부 AWS 서비스를 사용하면 역할을(프록시로 사용하는 대신) 리소스에 정책을 직접 연결할 수 있습니다. 교차 계정 액세스를 위한 역할과 리소스 기반 정책의 차이점을 알아보려면 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하십시오.
- 교차 서비스 액세스 - 일부 AWS 서비스는 다른 AWS 서비스의 기능을 사용합니다. 예를 들어, 서비스에서 호출을 수행하면 일반적으로 해당 서비스는 Amazon EC2에서 애플리케이션을 실행하거나 Amazon S3에 객체를 저장합니다. 서비스는 호출하는 보안 주체의 권한을 사용하거나, 서비스 역할을 사용하거나, 또는 서비스 연결 역할을 사용하여 이 작업을 수행할 수 있습니다.
  - 전달 액세스 세션(FAS) - IAM 사용자 또는 역할을 사용하여 AWS에서 작업을 수행하는 사람은 보안 주체로 간주됩니다. 일부 서비스를 사용하는 경우 다른 서비스에서 다른 작업을 시작하는 작업

을 수행할 수 있습니다. FAS는 AWS 서비스를 직접 호출하는 보안 주체의 권한과 요청하는 AWS 서비스를 함께 사용하여 다운스트림 서비스에 대한 요청을 수행합니다. FAS 요청은 서비스에서 완료를 위해 다른 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 받은 경우에만 이루어집니다. 이 경우 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하십시오.

- 서비스 역할 - 서비스 역할은 서비스가 사용자를 대신하여 태스크를 수행하기 위해 맡는 IAM 역할입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성을 참조하십시오.
- 서비스 연결 역할 - 서비스 연결 역할은 AWS 서비스에 연결된 서비스 역할의 한 유형입니다. 서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 연결 역할은 AWS 계정에 나타나고, 서비스가 소유합니다. IAM 관리자는 서비스 연결 역할의 권한을 볼 수 있지만 편집할 수는 없습니다.

- Amazon EC2에서 실행 중인 애플리케이션 – IAM 역할을 사용하여 EC2 인스턴스에서 실행되고 AWS CLI 또는 AWS API 요청을 수행하는 애플리케이션의 임시 보안 인증을 관리할 수 있습니다. 이는 EC2 인스턴스 내에 액세스 키를 저장할 때 권장되는 방법입니다. EC2 인스턴스에 AWS역할을 할당하고 해당 역할을 모든 애플리케이션에서 사용할 수 있도록 하려면 인스턴스에 연결된 인스턴스 프로파일을 생성합니다. 인스턴스 프로파일에는 역할이 포함되어 있으며 EC2 인스턴스에서 실행되는 프로그램이 임시 보안 인증을 얻을 수 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM 역할을 사용하여 Amazon EC2 인스턴스에서 실행되는 애플리케이션에 권한 부여를 참조하십시오.

IAM 역할을 사용할지 또는 IAM 사용자를 사용할지를 알아보려면 IAM 사용 설명서의 IAM 역할(사용자 대신)을 생성하는 경우를 참조하십시오.

정책을 사용한 액세스 관리

정책을 생성하고 AWSID 또는 리소스에 연결하여 AWS내 액세스를 제어합니다. 정책은 자격 증명 또는 리소스와 연결될 때 해당 권한을 정의하는 AWS의 객체입니다. AWS는 보안 주체(사용자, 루트 사용자 또는 역할 세션)가 요청을 보낼 때 이러한 정책을 평가합니다. 정책에서 권한은 요청이 허용되는지 또는 거부되는지를 결정합니다. 대부분의 정책은 AWS에 JSON 설명서로서 저장됩니다. JSON 정책 문서의 구조와 콘텐츠에 대한 자세한 정보는 IAM 사용 설명서의 JSON 정책 개요를 참조하십시오.

관리자는 AWSJSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

기본적으로, 사용자와 역할에는 어떠한 권한도 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 수임할 수 있습니다.

IAM 정책은 작업을 수행하기 위해 사용하는 방법과 상관없이 작업에 대한 권한을 정의합니다. 예를 들어, `iam:GetRole`태스크를 허용하는 정책이 있다고 가정합니다. 해당 정책이 있는 사용자는 AWS Management Console, AWS CLI또는 AWSAPI에서 역할 정보를 가져올 수 있습니다.

ID 기반 정책

ID 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 자격 증명에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자와 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하십시오.

ID 기반 정책은 인라인 정책 또는 관리형 정책으로 분류할 수 있습니다. 인라인 정책은 단일 사용자, 그룹 또는 역할에 직접 포함됩니다. 관리형 정책은 AWS 계정에 속한 다수의 사용자, 그룹 및 역할에 독립적으로 추가할 수 있는 정책입니다. 관리형 정책에는 AWS관리형 정책과 고객 관리형 정책이 포함되어 있습니다. 관리형 정책 또는 인라인 정책을 선택하는 방법을 알아보려면 IAM 사용 설명서의 관리형 정책과 인라인 정책의 선택을 참조하십시오.

리소스 기반 정책

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 리소스 기반 정책의 예는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 보안 주체가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 정의합니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 페더레이션 사용자 또는 AWS 서비스가 포함될 수 있습니다.

리소스 기반 정책은 해당 서비스에 있는 인라인 정책입니다. 리소스 기반 정책에서는 IAM의 AWS관리형 정책을 사용할 수 없습니다.

액세스 제어 목록(ACL)

액세스 제어 목록(ACL)은 어떤 보안 주체(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는지를 제어합니다. ACLs는 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

Amazon S3, AWS WAF및 Amazon VPC는 ACL을 지원하는 대표적인 서비스입니다. ACL에 대해 자세히 알아보려면 Amazon Simple Storage Service 개발자 안내서의 ACL(액세스 제어 목록) 개요를 참조하십시오.

기타 정책 유형

AWS은(는) 비교적 일반적이지 않은 추가 정책 유형을 지원합니다. 이러한 정책 유형은 더 일반적인 정책 유형에 따라 사용자에게 부여되는 최대 권한을 설정할 수 있습니다.

- 권한 경계 – 권한 경계는 ID 기반 정책에 따라 IAM 엔터티(IAM 사용자 또는 역할)에 부여할 수 있는 최대 권한을 설정하는 고급 기능입니다. 개체에 대한 권한 경계를 설정할 수 있습니다. 그 결과로 얻는 권한은 엔터티의 ID 기반 정책 및 해당 권한 경계의 교집합입니다. `Principal` 필드에서 사용자나 역할을 보안 주체로 지정하는 리소스 기반 정책은 권한 경계를 통해 제한되지 않습니다. 이러한 정책 중 하나에 포함된 명시적 거부는 허용을 재정의합니다. 권한 경계에 대한 자세한 정보는 IAM 사용 설명서의 IAM 엔터티에 대한 권한 경계를 참조하십시오.
- 서비스 제어 정책(SCP) – SCP는 AWS Organizations에서 조직 또는 조직 단위(OU)에 최대 권한을 지정하는 JSON 정책입니다. AWS Organizations은 기업이 소유하는 여러 개의 AWS 계정를 그룹화하고 중앙에서 관리하기 위한 서비스입니다. 조직에서 모든 기능을 활성화할 경우, 서비스 제어 정책(SCP)을 임의의 계정 또는 모든 계정에 적용할 수 있습니다. SCP는 각 AWS 계정 루트 사용자을 비롯하여 멤버 계정의 엔터티에 대한 권한을 제한합니다. 조직 및 SCP에 대한 자세한 정보는 AWS Organizations 사용 설명서의 SCP 작동 방식을 참조하십시오.
- 세션 정책 – 세션 정책은 역할 또는 페더레이션 사용자에 대해 임시 세션을 프로그래밍 방식으로 생성할 때 파라미터로 전달하는 고급 정책입니다. 결과적으로 얻는 세션의 권한은 사용자 또는 역할의 ID 기반 정책과 세션 정책의 교집합입니다. 또한 권한을 리소스 기반 정책에서 가져올 수도 있습니다. 이러한 정책 중 하나에 포함된 명시적 거부는 허용을 재정의합니다. 자세한 정보는 IAM 사용 설명서의 세션 정책을 참조하십시오.

여러 정책 유형

여러 정책 유형이 요청에 적용되는 경우, 결과 권한은 이해하기가 더 복잡합니다. 여러 정책 유형이 관련될 때 AWS가 요청을 허용할지 여부를 결정하는 방법을 알아보려면 IAM 사용 설명서의 정책 평가 로직을 참조하세요.

AWS Lambda에서 IAM을 사용하는 방식

IAM을 사용하여 Lambda에 대한 액세스를 관리하기 전에 Lambda에서 사용할 수 있는 IAM 기능에 대해 알아보십시오.

AWS Lambda을 통해 사용할 수 있는 IAM 기능

IAM 특성	람다 지원
ID 기반 정책	예
리소스 기반 정책	예
정책 작업	예
정책 리소스	예
정책 조건 키(서비스별)	예
ACLs	아니요
ABAC(정책 내 태그)	부분
임시 보안 인증	예
전달 액세스 세션(FAS)	아니요
서비스 역할	예
서비스 연결 역할	부분

Lambda 및 AWS 기타 서비스가 대부분의 IAM 기능과 어떻게 작동하는지 자세히 알아보려면 IAM 사용 설명서에서 IAM과 함께 작동하는 서비스를 AWS 참조하십시오.

Lambda에 대한 자격 증명 기반 정책

ID 기반 정책 지원	예

ID 기반 정책은 IAM 사용자, 사용자 그룹 또는 역할과 같은 자격 증명에 연결할 수 있는 JSON 권한 정책 문서입니다. 이러한 정책은 사용자와 역할이 어떤 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 제어합니다. 자격 증명 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하십시오.

IAM 자격 증명 기반 정책을 사용하면 허용되거나 거부되는 작업과 리소스뿐 아니라 작업이 허용되거나 거부되는 조건을 지정할 수 있습니다. ID 기반 정책에서는 보안 주체가 연결된 사용자 또는 역할에 적용되므로 보안 주체를 지정할 수 없습니다. JSON 정책에서 사용하는 모든 요소에 대해 알아보려면 IAM 사용 설명서의 IAM JSON 정책 요소 참조를 참조하십시오.

Lambda의 자격 증명 기반 정책 예제

Lambda 자격 증명 기반 정책의 예를 보려면 을 참조하십시오. AWS Lambda에 대한 자격 증명 기반 정책 예시

Lambda 내의 리소스 기반 정책

리소스 기반 정책 지원	예

리소스 기반 정책은 리소스에 연결하는 JSON 정책 설명서입니다. 리소스 기반 정책의 예는 IAM 역할 신뢰 정책과 Amazon S3 버킷 정책입니다. 리소스 기반 정책을 지원하는 서비스에서 서비스 관리자는 이러한 정책을 사용하여 특정 리소스에 대한 액세스를 통제할 수 있습니다. 정책이 연결된 리소스의 경우 정책은 지정된 보안 주체가 해당 리소스와 어떤 조건에서 어떤 작업을 수행할 수 있는지를 정의합니다. 리소스 기반 정책에서 보안 주체를 지정해야 합니다. 보안 주체에는 계정, 사용자, 역할, 페더레이션 사용자 또는 AWS 서비스이(가) 포함될 수 있습니다.

교차 계정 액세스를 활성화하려는 경우 전체 계정이나 다른 계정의 IAM 엔터티를 리소스 기반 정책의 보안 주체로 지정할 수 있습니다. 리소스 기반 정책에 크로스 계정 보안 주체를 추가하는 것은 트러스트 관계 설정의 절반밖에 되지 않는다는 것을 유념하십시오. 보안 주체와 리소스가 서로 다른 AWS 계정에 있는 경우 신뢰할 수 있는 계정의 IAM 관리자는 보안 주체 개체(사용자 또는 역할)에도 리소스 액세스 권한을 부여해야 합니다. 엔터티에 ID 기반 정책을 연결하여 권한을 부여합니다. 하지만 리소스 기반 정책이 동일 계정의 보안 주체에 액세스를 부여하는 경우 추가 ID 기반 정책이 필요하지 않습니다. 자세한 정보는 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하십시오.

리소스 기반 정책을 Lambda 함수 또는 계층에 연결할 수 있습니다. 이 정책은 함수 또는 계층에서 작업을 수행할 수 있는 보안 주체를 정의합니다.

리소스 기반 정책을 함수 또는 계층에 연결하는 방법을 알아보려면 을 참조하십시오. Lambda에서 리소스 기반 정책 작업

Lambda에 대한 정책 조치

정책 작업 지원	예

관리자는 AWSJSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지를 지정할 수 있습니다.

JSON 정책의 `Action`요소는 정책에서 액세스를 허용하거나 거부하는 데 사용할 수 있는 태스크를 설명합니다. 일반적으로 정책 작업의 이름은 연결된 AWSAPI 작업의 이름과 동일합니다. 일치하는 API 작업이 없는 권한 전용 작업 같은 몇 가지 예외도 있습니다. 정책에서 여러 작업이 필요한 몇 가지 작업도 있습니다. 이러한 추가 작업을 일컬어 종속 작업이라고 합니다.

연결된 작업을 수행할 수 있는 권한을 부여하기 위한 정책에 작업을 포함합니다.

Lambda 작업 목록을 보려면 서비스 권한 부여 AWS Lambda 참조에 정의된 작업을 참조하십시오.

Lambda의 정책 작업은 작업 앞에 다음 접두사를 사용합니다.

```
lambda
```

단일 명령문에서 여러 작업을 지정하려면 다음과 같이 쉼표로 구분합니다.

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

Lambda 자격 증명 기반 정책의 예를 보려면 을 참조하십시오. AWS Lambda에 대한 자격 증명 기반 정책 예시

Lambda용 정책 리소스

정책 리소스 지원	예

관리자는 AWSJSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지 지정할 수 있습니다.

`Resource` JSON 정책 요소는 작업이 적용되는 하나 이상의 개체를 지정합니다. 문장에는 `Resource` 또는 `NotResource` 요소가 반드시 추가되어야 합니다. 모범 사례에 따라 Amazon 리소스 이름(ARN)을 사용하여 리소스를 지정합니다. 리소스 수준 권한이라고 하는 특정 리소스 유형을 지원하는 작업에 대해 이 태스크를 수행할 수 있습니다.

작업 나열과 같이 리소스 수준 권한을 지원하지 않는 작업의 경우, 와일드카드(*)를 사용하여 해당 문이 모든 리소스에 적용됨을 나타냅니다.

```
"Resource": "*"
```

Lambda 리소스 유형 및 해당 ARN 목록을 보려면 서비스 권한 부여 참조에 정의된 AWS Lambda 리소스 유형을 참조하십시오. 각 리소스의 ARN을 지정할 수 있는 작업을 알아보려면 AWS Lambda가 정의한 작업을 참조하세요.

Lambda 자격 증명 기반 정책의 예를 보려면 을 참조하십시오. AWS Lambda에 대한 자격 증명 기반 정책 예시

Lambda의 정책 조건 키

서비스별 정책 조건 키 지원	예

관리자는 AWSJSON 정책을 사용하여 누가 무엇에 액세스할 수 있는지를 지정할 수 있습니다. 즉, 어떤 보안 주체가 어떤 리소스와 어떤 조건에서 작업을 수행할 수 있는지 지정할 수 있습니다.

`Condition` 요소(또는 `Condition` 블록)를 사용하면 문장이 발효되는 조건을 지정할 수 있습니다. `Condition` 요소는 옵션입니다. 같거나 작음과 같은 조건 연산자를 사용하여 정책의 조건을 요청의 값과 일치시키는 조건식을 생성할 수 있습니다.

한 문장에서 여러 `Condition` 요소를 지정하거나 단일 `Condition` 요소에서 여러 키를 지정하는 경우 AWS는 논리적 `AND` 태스크를 사용하여 평가합니다. 단일 조건 키의 여러 값을 지정하는 경우 AWS는 논리적 `OR`태스크를 사용하여 조건을 평가합니다. 문장의 권한을 부여하기 전에 모든 조건을 충족해야 합니다.

조건을 지정할 때 자리표시자 변수를 사용할 수도 있습니다. 예를 들어, IAM 사용자에게 IAM 사용자 이름으로 태그가 지정된 경우에만 리소스에 액세스할 수 있는 권한을 부여할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM 정책 요소: 변수 및 태그를 참조하십시오.

AWS는 전역 조건 키와 서비스별 조건 키를 지원합니다. 모든 AWS 전역 조건 키를 보려면 IAM 사용 설명서의 AWS 전역 조건 컨텍스트 키를 참조하십시오.

Lambda 조건 키 목록을 보려면 서비스 권한 부여 참조의 조건 키를 AWS Lambda 참조하십시오. 조건 키를 사용할 수 있는 작업과 리소스를 알아보려면 AWS Lambda가 정의한 작업 단원을 참조하세요.

Lambda 자격 증명 기반 정책의 예를 보려면 을 참조하십시오. AWS Lambda에 대한 자격 증명 기반 정책 예시

람다의 ACL

ACL 지원	아니요

액세스 제어 목록(ACLs)은 어떤 보안 주체(계정 멤버, 사용자 또는 역할)가 리소스에 액세스할 수 있는 권한을 가지고 있는 지를 제어합니다. ACL은 JSON 정책 문서 형식을 사용하지 않지만 리소스 기반 정책과 유사합니다.

람다를 사용한 ABAC

ABAC(정책 내 태그) 지원	부분

속성 기반 액세스 제어(ABAC)는 속성을 기반으로 권한을 정의하는 권한 부여 전략입니다. AWS에서는 이러한 속성을 태그라고 합니다. IAM 엔터티(사용자 또는 역할) 및 많은 AWS리소스에 태그를 연결할 수 있습니다. ABAC의 첫 번째 단계로 개체 및 리소스에 태그를 지정합니다. 그런 다음 보안 주체의 태그가 액세스하려는 리소스의 태그와 일치할 때 작업을 허용하도록 ABAC 정책을 설계합니다.

ABAC는 빠르게 성장하는 환경에서 유용하며 정책 관리가 번거로운 상황에 도움이 됩니다.

태그를 기반으로 액세스를 제어하려면 `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`또는 `aws:TagKeys`조건 키를 사용하여 정책의 조건 요소에 태그 정보를 제공합니다.

서비스가 모든 리소스 유형에 대해 세 가지 조건 키를 모두 지원하는 경우 값은 서비스에 대해 예입니다. 서비스가 일부 리소스 유형에 대해서만 세 가지 조건 키를 모두 지원하는 경우 값은 부분입니다.

ABAC에 대한 자세한 정보는 IAM 사용 설명서의 ABAC란 무엇인가요?를 참조하십시오. ABAC 설정 단계가 포함된 자습서를 보려면 IAM 사용 설명서의 속성 기반 액세스 제어(ABAC) 사용을 참조하십시오.

Lambda 리소스 태깅에 대한 자세한 내용은 을 참조하십시오. Lambda에서 속성 기반 액세스 제어 사용

Lambda에서 임시 자격 증명 사용

임시 보안 인증 지원	예

일부 AWS 서비스는 임시 보안 인증 정보를 사용하여 로그인할 때 작동하지 않습니다. 임시 자격 증명으로 작동하는 AWS 서비스를 비롯한 추가 정보는 IAM 사용 설명서의 IAM으로 작업하는 AWS 서비스를 참조하십시오.

사용자 이름과 암호를 제외한 다른 방법을 사용하여 AWS Management Console에 로그인하면 임시 보안 인증 정보를 사용하는 것입니다. 예를 들어 회사의 Single Sign-On(SSO) 링크를 사용하여 AWS에 액세스하면 해당 프로세스에서 자동으로 임시 보안 인증 정보를 생성합니다. 또한 콘솔에 사용자로 로그인한 다음 역할을 전환할 때 임시 보안 인증 정보를 자동으로 생성합니다. 역할 전환에 대한 자세한 정보는 IAM 사용 설명서의 역할로 전환(콘솔)을 참조하십시오.

AWS CLI 또는 AWSAPI를 사용하여 임시 보안 인증 정보를 수동으로 만들 수 있습니다 그런 다음 이러한 임시 보안 인증 정보를 사용하여 AWS에 액세스할 수 있습니다. AWS에서는 장기 액세스 키를 사용하는 대신 임시 보안 인증 정보를 동적으로 생성할 것을 권장합니다. 자세한 정보는 IAM의 임시 보안 자격 증명 섹션을 참조하십시오.

Lambda용 포워드 액세스 세션

전달 액세스 세션(FAS) 지원	아니요

IAM 사용자 또는 역할을 사용하여 AWS에서 작업을 수행하는 사람은 보안 주체로 간주됩니다. 일부 서비스를 사용하는 경우 다른 서비스에서 다른 작업을 시작하는 작업을 수행할 수 있습니다. FAS는 AWS 서비스를 직접 호출하는 보안 주체의 권한과 요청하는 AWS 서비스를 함께 사용하여 다운스트림 서비스에 대한 요청을 수행합니다. FAS 요청은 서비스에서 완료를 위해 다른 AWS 서비스 또는 리소스와의 상호 작용이 필요한 요청을 받은 경우에만 이루어집니다. 이 경우 두 작업을 모두 수행할 수 있는 권한이 있어야 합니다. FAS 요청 시 정책 세부 정보는 전달 액세스 세션을 참조하십시오.

Lambda의 서비스 역할

서비스 역할 지원	예

서비스 역할은 서비스가 사용자를 대신하여 작업을 수행하는 것으로 가정하는 IAM 역할입니다. IAM 관리자는 IAM 내에서 서비스 역할을 생성, 수정 및 삭제할 수 있습니다. 자세한 정보는 IAM 사용 설명서의 AWS 서비스에 대한 권한을 위임할 역할 생성을 참조하십시오.

Lambda에서는 서비스 역할을 실행 역할이라고 합니다.

> **Warning**
>
> 실행 역할에 대한 권한을 변경하면 Lambda 기능이 손상될 수 있습니다.

Lambda의 서비스 연결 역할

서비스 연결 역할 지원	부분

서비스 연결 역할은 AWS 서비스에 연결된 서비스 역할의 한 타입입니다. 서비스는 사용자를 대신하여 작업을 수행하기 위해 역할을 수임할 수 있습니다. 서비스 연결 역할은 AWS 계정에 나타나고, 서비스가 소유합니다. IAM 관리자는 서비스 연결 역할의 권한을 볼 수 있지만 편집할 수는 없습니다.

Lambda에는 서비스 연결 역할이 없지만 Lambda@Edge에는 있습니다. 자세한 내용은 Amazon 개발자 안내서의 Lambda @Edge 서비스 연결 역할을 참조하십시오. CloudFront

서비스 연결 역할 생성 또는 관리에 대한 자세한 내용은 IAM으로 작업하는 AWS 서비스 단원을 참조하십시오. 서비스 연결 역할 열에서 `Yes`이(가) 포함된 서비스를 테이블에서 찾습니다. 해당 서비스에 대한 서비스 연결 역할 설명서를 보려면 Yes(네) 링크를 선택합니다.

AWS Lambda에 대한 자격 증명 기반 정책 예시

기본적으로 사용자 및 역할은 Lambda 리소스를 생성하거나 수정할 수 있는 권한이 없습니다. 또한 AWS Management Console, AWS Command Line Interface(AWS CLI) 또는 AWS API를 사용하여 작업을 수행할 수 없습니다. 사용자에게 사용자가 필요한 리소스에서 작업을 수행할 권한을 부여하려면 IAM 관리자가 IAM 정책을 생성하면 됩니다. 그런 다음 관리자가 IAM 정책을 역할에 추가하고, 사용자가 역할을 수임할 수 있습니다.

이러한 예제 JSON 정책 문서를 사용하여 IAM ID 기반 정책을 생성하는 방법을 알아보려면 IAM 사용 설명서의 IAM 정책 생성을 참조하십시오.

각 리소스 유형의 ARN 형식을 비롯하여 Lambda에서 정의한 작업 및 리소스 유형에 대한 자세한 내용은 서비스 권한 부여 참조의 작업, 리소스 및 조건 키를 AWS Lambda 참조하십시오.

주제

- 정책 모범 사례
- Lambda 콘솔 사용
- 사용자가 자신이 권한을 볼 수 있도록 허용

정책 모범 사례

ID 기반 정책에 따라 계정에서 사용자가 Lambda 리소스를 생성, 액세스 또는 삭제할 수 있는지 여부가 결정됩니다. 이 작업으로 인해 AWS 계정에 비용이 발생할 수 있습니다. 자격 증명 기반 정책을 생성하거나 편집할 때는 다음 지침과 권장 사항을 따르십시오.

- AWS 관리형 정책으로 시작하고 최소 권한을 향해 나아가기 - 사용자 및 워크로드에 권한 부여를 시작하려면 많은 일반 사용 사례에 대한 권한을 부여하는 AWS관리형 정책을 사용합니다. 관리형 정책은 AWS 계정에서 사용할 수 있습니다. 사용 사례에 고유한 AWS고객 관리형 정책을 정의하여 권한을 줄이는 것이 좋습니다. 자세한 정보는 IAM 사용 설명서의 AWS managed policies(관리형 정책) 또는 AWS managed policies for job functions(직무에 대한 관리형 정책)를 참조하십시오.
- 최소 권한 적용 – IAM 정책을 사용하여 권한을 설정하는 경우 작업을 수행하는 데 필요한 권한만 부여합니다. 이렇게 하려면 최소 권한으로 알려진 특정 조건에서 특정 리소스에 대해 수행할 수 있는 작업을 정의합니다. IAM을 사용하여 권한을 적용하는 방법에 대한 자세한 정보는 IAM 사용 설명서에 있는 Policies and permissions in IAM(IAM의 정책 및 권한)을 참조하십시오.
- IAM 정책의 조건을 사용하여 액세스 추가 제한: 정책에 조건을 추가하여 작업 및 리소스에 대한 액세스를 제한할 수 있습니다. 예를 들어 SSL을 사용하여 모든 요청을 전송해야 한다고 지정하는 정책 조건을 작성할 수 있습니다. 특정 AWS 서비스(예: AWS CloudFormation)을(를) 통해 사용되는 경우에만 서비스 작업에 대한 액세스 권한을 부여할 수도 있습니다. 자세한 정보는 IAM 사용 설명서의 IAM JSON 정책 요소: 조건을 참조하십시오.
- IAM Access Analyzer를 통해 IAM 정책을 검증하여 안전하고 기능적인 권한 보장 – IAM Access Analyzer에서는 IAM 정책 언어(JSON)와 IAM 모범 사례가 정책에서 준수되도록 신규 및 기존 정책을 검증합니다. IAM Access Analyzer는 100개 이상의 정책 확인 항목과 실행 가능한 권장 사항을 제공하여 안전하고 기능적인 정책을 작성하도록 돕습니다. 자세한 정보는 IAM 사용 설명서의 IAM Access Analyzer policy validation(IAM Access Analyzer 정책 검증)을 참조하십시오.
- 다중 인증(MFA) 필요 – AWS 계정 계정에 IAM 사용자 또는 루트 사용자가 필요한 시나리오가 있는 경우, 추가 보안을 위해 MFA를 설정합니다. API 작업을 직접적으로 호출할 때 MFA가 필요하면 정책에 MFA 조건을 추가합니다. 자세한 정보는 IAM 사용 설명서의 Configuring MFA-protected API access(MFA 보호 API 액세스 구성)를 참조하십시오.

IAM의 모범 사례에 대한 자세한 내용은 IAM 사용 설명서의 IAM의 보안 모범 사례를 참조하십시오.

Lambda 콘솔 사용

AWS Lambda 콘솔에 액세스하려면 최소 권한 세트가 있어야 합니다. 이러한 권한을 통해 내 Lambda 리소스를 나열하고 세부 정보를 볼 수 있어야 합니다. AWS 계정 최소 필수 권한보다 더 제한적인 자격

증명 기반 정책을 만들면 콘솔이 해당 정책에 연결된 엔터티(사용자 또는 역할)에 대해 의도대로 작동하지 않습니다.

AWS CLI 또는 AWSAPI만 호출하는 사용자에게 최소 콘솔 권한을 허용할 필요가 없습니다. 그 대신, 수행하려는 API 작업과 일치하는 작업에만 액세스할 수 있도록 합니다.

함수 개발을 위해 최소한의 액세스를 부여하는 정책 예제는 단원을 참조하세요사용자에게 함수에 대한 권한을 부여하는 예제 정책 작성 Lambda API에 더해 Lambda 콘솔은 다른 서비스를 사용하여 트리거 구성을 표시하고 사용자의 신규 트리거 추가를 돕습니다. 사용자가 다른 서비스와 함께 Lambda를 사용하는 경우 해당 서비스에도 액세스해야 합니다. Lambda로 다른 서비스를 구성하는 방법에 대한 자세한 내용은 다른 AWS 서비스의 이벤트로 Lambda 간접 호출 단원을 참조하세요.

사용자가 자신이 권한을 볼 수 있도록 허용

이 예제는 IAM 사용자가 자신의 사용자 ID에 연결된 인라인 및 관리형 정책을 볼 수 있도록 허용하는 정책을 생성하는 방법을 보여줍니다. 이 정책에는 콘솔에서 또는 AWS CLI나 AWS API를 사용하여 프로그래밍 방식으로 이 작업을 완료할 수 있는 권한이 포함됩니다.

```
{
    "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 Lambda의 AWS 관리형 정책

AWS 관리형 정책은 AWS에서 생성되고 관리되는 독립형 정책입니다. AWS 관리형 정책은 사용자, 그룹 및 역할에 권한 할당을 시작할 수 있도록 많은 일반 사용 사례에 대한 권한을 제공하도록 설계되었습니다.

AWS 관리형 정책은 모든 AWS 고객이 사용할 수 있기 때문에 특정 사용 사례에 대해 최소 권한을 부여하지 않을 수 있습니다. 사용 사례에 고유한 고객 관리형 정책을 정의하여 권한을 줄이는 것이 좋습니다.

AWS 관리형 정책에서 정의한 권한은 변경할 수 없습니다. AWS에서 AWS 관리형 정책에 정의된 권한을 업데이트할 경우 정책이 연결되어 있는 모든 보안 주체 엔터티(사용자, 그룹 및 역할)에도 업데이트가 적용됩니다. 새로운 AWS 서비스를 시작하거나 새로운 API 작업을 기존 서비스에 이용하는 경우 AWS가 AWS 관리형 정책을 업데이트할 가능성이 높습니다.

자세한 내용은 IAM 사용 설명서의 AWS 관리형 정책을 참조하십시오.

주제

- AWS관리형 정책: AWSLambda_FullAccess
- AWS관리형 정책: AWSLambda_ReadOnlyAccess
- AWS관리형 정책: AWSLambdaBasicExecutionRole
- AWS관리형 정책: AWSLambdaDynamoDBExecutionRole
- AWS관리형 정책: AWSLambdaENIManagementAccess
- AWS관리형 정책: AWSLambdaExecute
- AWS관리형 정책: AWSLambdaInvocation -DynamoDB
- AWS관리형 정책: AWSLambdaKinesisExecutionRole

- AWS관리형 정책: AWSLambdaMSKExecutionRole
- AWS관리형 정책: AWSLambdaRole
- AWS관리형 정책: AWSLambdaSQSQueueExecutionRole
- AWS관리형 정책: AWSLambdaVPCAccessExecutionRole
- AWS 관리형 정책에 대한 Lambda 업데이트

AWS관리형 정책: AWSLambda_FullAccess

이 정책은 Lambda 작업에 대한 전체 액세스 권한을 부여합니다. 또한 Lambda 리소스를 개발하고 유지 관리하는 데 사용되는 다른 AWS 서비스에 대한 권한도 부여합니다.

사용자, 그룹 및 역할에 `AWSLambda_FullAccess` 정책을 연결할 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `lambda` – 보안 주체에게 Lambda에 대한 전체 액세스 권한을 허용합니다.
- `cloudformation` – 보안 주체가 AWS CloudFormation 스택을 설명하고 해당 스택의 리소스를 나열하도록 허용합니다.
- `cloudwatch`— 보안 주체가 Amazon CloudWatch 메트릭을 나열하고 지표 데이터를 가져올 수 있습니다.
- `ec2` - 보안 주체가 보안 그룹, 서브넷 및 VPC를 설명하도록 허용합니다.
- `iam` - 보안 주체가 정책, 정책 버전, 역할, 역할 정책, 연결된 역할 정책 및 역할 목록을 가져오도록 허용합니다. 이 정책은 또한 보안 주체가 Lambda에 역할을 전달할 수 있도록 허용합니다. `PassRole` 권한은 함수에 실행 역할을 할당할 때 사용됩니다.
- `kms` - 보안 주체가 별칭을 나열하도록 허용합니다.
- `logs`— 주도자가 Amazon CloudWatch 로그 그룹을 설명할 수 있습니다. Lambda 함수와 연결된 로그 그룹의 경우 이 정책은 보안 주체가 로그 스트림을 설명하고, 로그 이벤트를 가져오고, 로그 이벤트를 필터링하도록 허용합니다.
- `states` – 보안 주체가 AWS Step Functions 상태 시스템을 설명하고 나열하도록 허용합니다.
- `tag` - 보안 주체가 태그를 기반으로 리소스를 가져오도록 허용합니다.
- `xray` - 보안 주체가 AWS X-Ray 트레이스 요약을 가져오고 ID로 지정된 트레이스 목록을 검색하도록 허용합니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambda_FullAccess.

AWS관리형 정책: AWSLambda_ReadOnlyAccess

이 정책은 Lambda 리소스 및 Lambda 리소스를 개발하고 유지 관리하는 데 사용되는 다른 AWS 서비스에 대한 읽기 전용 액세스 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambda_ReadOnlyAccess` 정책을 연결할 수 있습니다.

권한 세부 정보

이 정책에는 다음 권한이 포함되어 있습니다.

- `lambda` - 보안 주체가 모든 리소스를 가져오고 나열하도록 허용합니다.
- `cloudformation` – 보안 주체가 AWS CloudFormation 스택을 설명 및 나열하고 해당 스택의 리소스를 나열하도록 허용합니다.
- `cloudwatch`— 보안 주체가 Amazon CloudWatch 메트릭을 나열하고 지표 데이터를 가져올 수 있습니다.
- `ec2` - 보안 주체가 보안 그룹, 서브넷 및 VPC를 설명하도록 허용합니다.
- `iam` - 보안 주체가 정책, 정책 버전, 역할, 역할 정책, 연결된 역할 정책 및 역할 목록을 가져오도록 허용합니다.
- `kms` - 보안 주체가 별칭을 나열하도록 허용합니다.
- `logs`— 주도자가 Amazon CloudWatch 로그 그룹을 설명할 수 있습니다. Lambda 함수와 연결된 로그 그룹의 경우 이 정책은 보안 주체가 로그 스트림을 설명하고, 로그 이벤트를 가져오고, 로그 이벤트를 필터링하도록 허용합니다.
- `states` – 보안 주체가 AWS Step Functions 상태 시스템을 설명하고 나열하도록 허용합니다.
- `tag` - 보안 주체가 태그를 기반으로 리소스를 가져오도록 허용합니다.
- `xray` - 보안 주체가 AWS X-Ray 트레이스 요약을 가져오고 ID로 지정된 트레이스 목록을 검색하도록 허용합니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambda_ReadOnlyAccess.

AWS관리형 정책: AWSLambdaBasicExecutionRole

이 정책은 로그를 Logs에 업로드할 CloudWatch 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaBasicExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaBasicExecutionRole.

AWS관리형 정책: AWSLambdaDynamoDBExecutionRole

이 정책은 Amazon DynamoDB 스트림에서 레코드를 읽고 로그에 쓸 수 있는 권한을 부여합니다. CloudWatch

사용자, 그룹 및 역할에 `AWSLambdaDynamoDBExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaDynamoDBExecutionRole.

AWS관리형 정책: AWSLambdaENIManagementAccess

이 정책은 VPC 지원 Lambda 함수에서 사용하는 탄력적 네트워크 인터페이스를 생성, 설명 및 삭제할 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaENIManagementAccess` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaENIManagementAccess.

AWS관리형 정책: AWSLambdaExecute

이 정책은 Amazon 심플 스토리지 서비스에 `GET` 대한 액세스 `PUT` 및 CloudWatch 로그에 대한 전체 액세스 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaExecute` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaExecute.

AWS관리형 정책: AWSLambdaInvocation -DynamoDB

이 정책은 Amazon DynamoDB Streams에 대한 읽기 액세스 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaInvocation-DynamoDB` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 가이드의 AWSLambdaInvocation-DynamoDB를 참조하십시오.

AWS관리형 정책: AWSLambdaKinesisExecutionRole

이 정책은 Amazon Kinesis 데이터 스트림에서 이벤트를 읽고 로그에 쓸 수 있는 권한을 부여합니다 CloudWatch .

사용자, 그룹 및 역할에 `AWSLambdaKinesisExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaKinesisExecutionRole.

AWS관리형 정책: AWSLambdaMSKExecutionRole

이 정책은 Amazon Managed Streaming for Apache Kafka 클러스터에서 레코드를 읽고 액세스하고, 엘라스틱 네트워크 인터페이스를 관리하고, Logs에 쓸 수 있는 권한을 부여합니다. CloudWatch

사용자, 그룹 및 역할에 `AWSLambdaMSKExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 AWSLambdaMSKExecutionRole안내서를 참조하십시오.

AWS관리형 정책: AWSLambdaRole

이 정책은 Lambda 함수를 간접적으로 호출할 수 있는 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaRole.

AWS관리형 정책: AWSLambdaSQSQueueExecutionRole

이 정책은 Amazon Simple Queue Service 대기열에서 메시지를 읽고 삭제할 권한을 부여하고, CloudWatch 로그에는 쓰기 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaSQSQueueExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaSQSQueueExecutionRole.

AWS관리형 정책: AWSLambdaVPCAccessExecutionRole

이 정책은 Amazon Virtual Private Cloud 내에서 엘라스틱 네트워크 인터페이스를 관리하고 CloudWatch Logs에 쓸 수 있는 권한을 부여합니다.

사용자, 그룹 및 역할에 `AWSLambdaVPCAccessExecutionRole` 정책을 연결할 수 있습니다.

JSON 정책 문서 및 정책 버전을 포함하여 이 정책에 대한 자세한 내용은 AWS관리형 정책 참조 안내서를 참조하십시오 AWSLambdaVPCAccessExecutionRole.

AWS 관리형 정책에 대한 Lambda 업데이트

변경 사항	설명	날짜
AWSLambdaVPCAccessExecutionRole— 변경	Lambda는 작업을 허용하도록 정책을 `AWSLambdaVPCAccessExecutionRole` 업데이트했습니다. `ec2:DescribeSubnets`	2024년 1월 5일
AWSLambda_ReadOnlyAccess— 변경	Lambda는 보안 주체가 AWS CloudFormation 스택을 나열할 수 있도록 `AWSLambda_ReadOnlyAccess` 정책을 업데이트했습니다.	2023년 7월 27일
AWS Lambda에서 변경 사항 추적 시작	AWS Lambda이(가) AWS 관리형 정책에 대한 변경 내용 추적을 시작했습니다.	2023년 7월 27일

AWS Lambda 보안 인증 및 액세스 문제 해결

다음 정보를 사용하여 Lambda 및 IAM으로 작업할 때 발생할 수 있는 일반적인 문제를 진단하고 수정할 수 있습니다.

주제

- Lambda에서 작업을 수행할 권한이 없음
- 저는 IAM을 수행할 권한이 없습니다. PassRole

- 외부 사용자가 Lambda 리소스에 액세스할 AWS 계정 수 있도록 허용하고 싶습니다.

Lambda에서 작업을 수행할 권한이 없음

작업을 수행할 권한이 없다는 오류가 수신되면, 작업을 수행할 수 있도록 정책을 업데이트해야 합니다.

다음 예제 오류는 mateojackson IAM 사용자가 콘솔을 사용하여 가상 *my-example-widget* 리소스에 대한 세부 정보를 보려고 하지만 가상 lambda:*GetWidget* 권한이 없을 때 발생합니다.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

이 경우 lambda:*GetWidget* 작업을 사용하여 *my-example-widget* 리소스에 액세스할 수 있도록 mateojackson 사용자 정책을 업데이트해야 합니다.

도움이 필요한 경우 AWS 관리자에게 문의하십시오. 관리자는 로그인 보안 인증을 제공한 사용자입니다.

저는 IAM을 수행할 권한이 없습니다. PassRole

iam:PassRole 작업을 수행할 수 있는 권한이 없다는 오류가 수신되면 Lambda에 역할을 전달할 수 있도록 정책을 업데이트해야 합니다.

일부 AWS 서비스에서는 새 서비스 역할 또는 서비스 연결 역할을 생성하는 대신 해당 서비스에 기존 역할을 전달할 수 있습니다. 이렇게 하려면 사용자가 서비스에 역할을 전달할 수 있는 권한을 가지고 있어야 합니다.

다음 예제 오류는 marymajor라는 IAM 사용자가 콘솔을 사용하여 Lambda에서 작업을 수행하려고 하는 경우에 발생합니다. 하지만 작업을 수행하려면 서비스 역할이 부여한 권한이 서비스에 있어야 합니다. Mary는 서비스에 역할을 전달할 수 있는 권한을 가지고 있지 않습니다.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

이 경우 Mary가 iam:PassRole작업을 수행할 수 있도록 Mary의 정책을 업데이트해야 합니다.

도움이 필요한 경우 AWS 관리자에게 문의하십시오. 관리자는 로그인 보안 인증을 제공한 사용자입니다.

외부 사용자가 Lambda 리소스에 액세스할 AWS 계정 수 있도록 허용하고 싶습니다.

다른 계정의 사용자 또는 조직 외부의 사람이 리소스에 액세스할 때 사용할 수 있는 역할을 생성할 수 있습니다. 역할을 수임할 신뢰할 수 있는 사람을 지정할 수 있습니다. 리소스 기반 정책 또는 액세스 제어 목록(ACL)을 지원하는 서비스의 경우 이러한 정책을 사용하여 다른 사람에게 리소스에 대한 액세스 권한을 부여할 수 있습니다.

자세히 알아보려면 다음을 참조하세요.

- Lambda에서 이러한 기능을 지원하는지 여부를 알아보려면 AWS Lambda에서 IAM을 사용하는 방식 단원을 참조하세요.
- 소유하고 있는 AWS 계정의 리소스에 대한 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 자신이 소유한 다른 AWS 계정의 IAM 사용자에 대한 액세스 권한 제공을 참조하십시오.
- 리소스에 대한 액세스 권한을 서드 파티 AWS 계정에게 제공하는 방법을 알아보려면 IAM 사용 설명서의 서드 파티가 소유한 AWS 계정에 대한 액세스 제공을 참조하십시오.
- 자격 증명 연동을 통해 액세스 권한을 제공하는 방법을 알아보려면 IAM 사용 설명서의 외부에서 인증된 사용자에게 액세스 권한 제공(자격 증명 연동)을 참조하십시오.
- 크로스 계정 액세스를 위한 역할과 리소스 기반 정책 사용의 차이점을 알아보려면 IAM 사용 설명서의 IAM 역할과 리소스 기반 정책의 차이를 참조하십시오.

Lambda 함수 및 계층에 대한 거버넌스 전략 생성

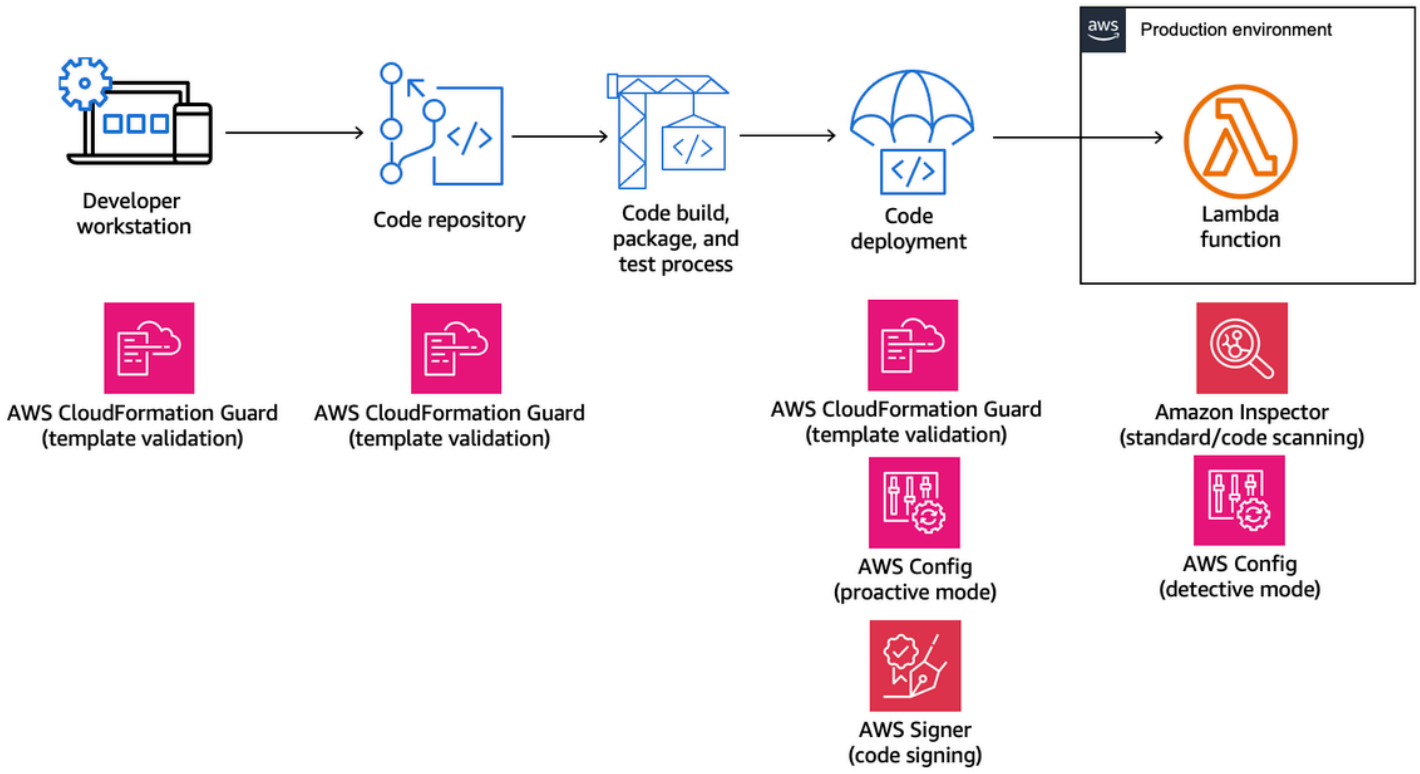
서버리스 클라우드 네이티브 애플리케이션을 빌드하고 배포하려면 적절한 거버넌스 및 가드레일을 통해 민첩성과 시장 출시 속도를 높여야 합니다. 비즈니스 수준의 우선순위를 설정합니다. 민첩성을 최우선으로 강조하거나 거버넌스, 가드레일 및 제어를 통한 위험 회피를 강조할 수 있습니다. 소프트웨어 개발 라이프사이클에서 민첩성과 가드레일의 균형을 맞출 때 현실적으로 '둘 중 하나'의 전략이 아니라 '둘 모두'를 고려해야 합니다. 이러한 요구 사항이 회사 라이프사이클의 어느 단계에 속하든, 프로세스와 도구 체인의 구현 요구 사항으로 거버넌스 기능이 포함될 가능성이 큽니다.

다음은 조직에서 Lambda에 대해 구현할 수 있는 거버넌스 제어의 몇 가지 예제입니다.

- Lambda 함수는 공개적으로 액세스할 수 없어야 합니다.
- Lambda 함수는 VPC에 연결되어야 합니다.
- Lambda 함수는 더 이상 사용되지 않는 런타임을 사용해서는 안 됩니다.
- Lambda 함수는 필수 태그 세트로 태그를 지정해야 합니다.
- Lambda 계층은 조직 외부에서 액세스할 수 없어야 합니다.

- 보안 그룹이 연결된 Lambda 함수에는 함수와 보안 그룹 간에 일치하는 태그가 있어야 합니다.
- 연결된 계층이 있는 Lambda 함수는 승인된 버전을 사용해야 합니다.
- Lambda 환경 변수는 고객 관리형 키로 저장 시 암호화되어야 합니다.

다음 다이어그램은 소프트웨어 개발 및 배포 프로세스 전반에 걸쳐 제어 및 정책을 구현하는 심층 거버넌스 전략의 예제입니다.



다음 주제에서는 스타트업과 엔터프라이즈 모두를 위해 조직에서 Lambda 함수를 개발하고 배포하기 위한 제어를 구현하는 방법을 설명합니다. 조직에 관련 도구가 이미 있을 수도 있습니다. 다음 주제에서는 이러한 제어에 대한 모듈식 접근 방식을 사용하므로 실제로 필요한 구성 요소를 선택할 수 있습니다.

주제

- AWS CloudFormation Guard를 사용하는 Lambda의 사전 예방적 제어
- AWS Config를 사용하여 Lambda에 대한 예방적 제어 구현
- AWS Config를 사용하여 규정 미준수 Lambda 배포 및 구성 감지
- AWS Signer를 사용하여 Lambda 코드 서명
- Amazon Inspector를 사용하여 Lambda에 대한 보안 평가 자동화

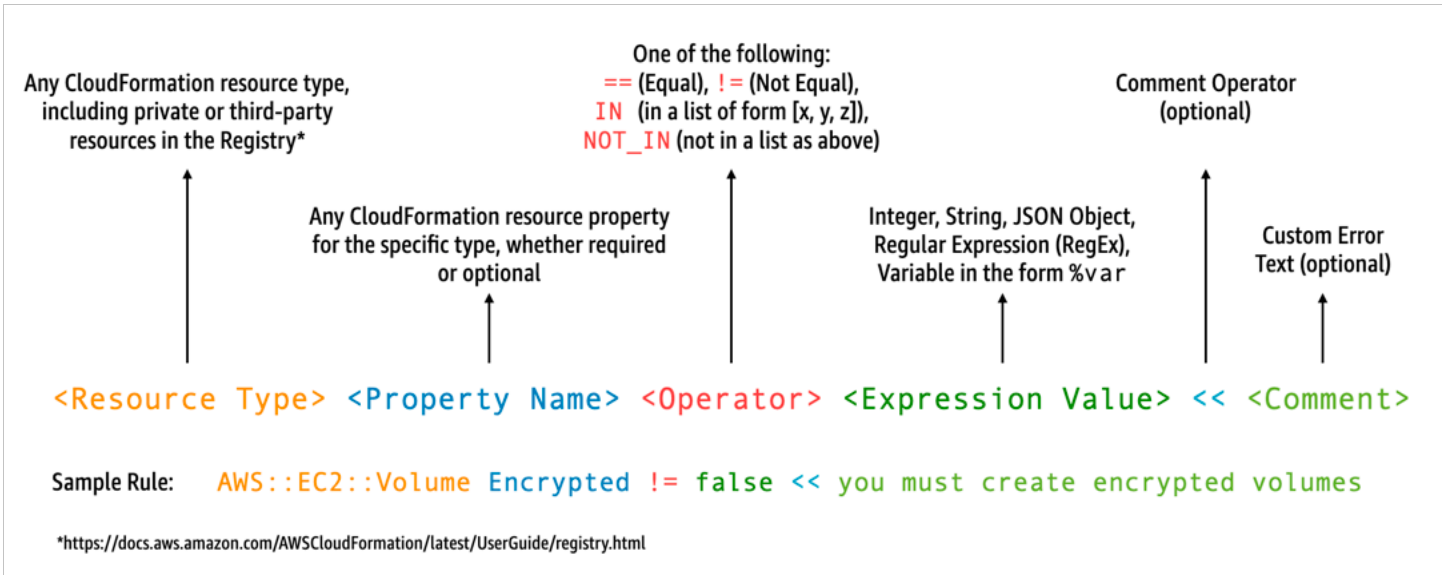
- Lambda 보안 및 규정 준수를 위한 관찰성 구현

AWS CloudFormation Guard를 사용하는 Lambda의 사전 예방적 제어

AWS CloudFormation Guard는 오픈 소스의 범용 평가 도구입니다. policy-as-code 정책 규칙을 기준으로 코드형 인프라(IaC) 템플릿 및 서비스 구성을 검증함으로써 예방적 거버넌스 및 규정 준수를 위해 사용할 수 있습니다. 이러한 규칙은 팀 또는 조직의 요구 사항에 따라 사용자 지정할 수 있습니다. Lambda 함수의 경우 Guard 규칙을 사용하여 Lambda 함수를 생성 또는 업데이트하는 동안 필요한 필수 속성 설정을 정의함으로써 리소스 생성 및 구성 업데이트를 제어할 수 있습니다.

규정 준수 관리자는 Lambda 함수를 배포하고 업데이트하는 데 필요한 제어 및 거버넌스 정책 목록을 정의합니다. 플랫폼 관리자는 코드 리포지토리가 포함된 사전 커밋 검증 웹후크로 CI/CD 파이프라인에서 제어를 구현하고, 개발자에게 로컬 워크스테이션에서 템플릿 및 코드를 검증하기 위한 명령줄 도구를 제공합니다. 개발자는 코드를 작성하고 명령줄 도구를 사용해 템플릿을 검증한 후 리포지토리에 코드를 커밋합니다. 그러면 리포지토리를 AWS 환경에 배포하기 전에 CI/CD 파이프라인을 통해 코드가 자동으로 검증됩니다.

Guard를 사용하면 다음과 같이 도메인별 언어로 규칙을 작성하고 제어를 구현할 수 있습니다.



예를 들어 개발자가 최신 런타임만 선택하려고 한다고 가정합니다. 두 가지 정책을 지정할 수 있습니다. 하나는 이미 지원이 중단된 런타임을 식별하는 정책이고 다른 하나는 곧 지원 중단 예정인 런타임을 식별하는 정책입니다. 이를 위해 다음 `etc/rules.guard` 파일을 작성할 수 있습니다.

```
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.>>
            }
        }
    }
}
```

이제 Lambda 함수를 정의하는 다음 `iac/lambda.yaml` CloudFormation 템플릿을 작성한다고 가정해 보겠습니다.

```
  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 유틸리티를 설치한 후 템플릿을 검증합니다.

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

출력은 다음과 같습니다.

```
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

      }
    }
  }
}
```

Guard를 사용하면 개발자가 로컬 개발자 워크스테이션에서 조직이 허용하는 런타임을 사용하도록 템플릿을 업데이트해야 한다는 사실을 확인할 수 있습니다. 코드 리포지토리를 커밋하고 이후 CI/CD 파이프라인 내에서 검사에 실패하기 전에 확인할 수 있습니다. 따라서 개발자는 규정을 준수하는 템플릿을 개발하고 비즈니스 가치를 전달하는 코드를 작성하는 데 시간을 더 할애하는 방법에 대한 피드백을

받을 수 있습니다. 이 제어는 배포 전에 로컬 개발자 워크스테이션, 사전 커밋 검증 웹후크 및/또는 CI/CD 파이프라인에 적용할 수 있습니다.

경고

AWS Serverless Application Model(AWS SAM) 템플릿을 사용하여 Lambda 함수를 정의하는 경우 다음과 같이 `AWS::Serverless::Function` 리소스 유형을 검색하도록 Guard 규칙을 업데이트해야 합니다.

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

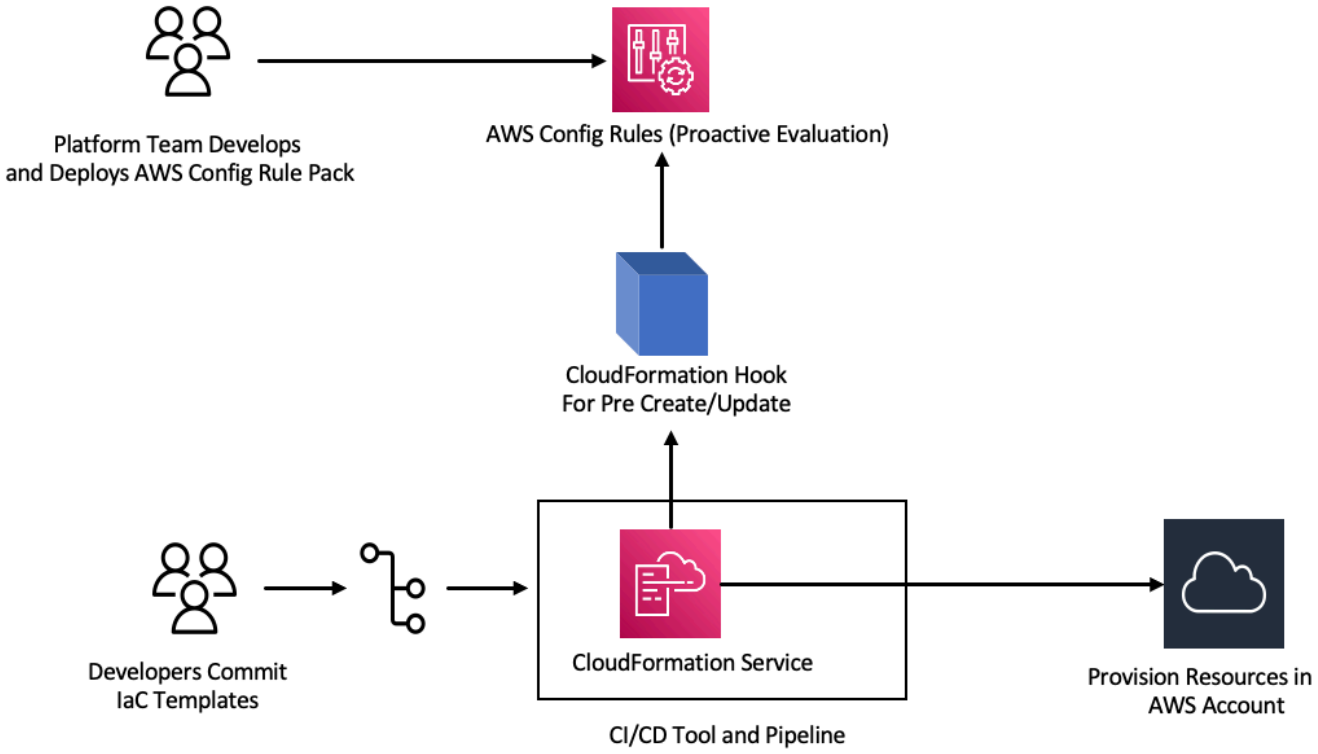
또한 Guard에서는 리소스 정의에 속성을 포함한다고 예상합니다. 한편, AWS SAM 템플릿을 사용하면 별도의 글로벌 섹션에서 속성을 지정할 수 있습니다. 글로벌 섹션에 정의된 속성은 Guard 규칙으로 검증되지 않습니다.

Guard 문제 해결 설명서에 설명된 대로, Guard에서는 짧은 양식의 내장 함수(예: `!GetAtt` 또는 `!Sub`)를 지원하지 않으며, 대신 확장된 양식(`Fn::GetAtt` 및 `Fn::Sub`)을 사용해야 합니다. (이전 예제에서는 역할 속성을 평가하지 않으므로 단순한 설명을 위해 짧은 양식의 내장 함수를 사용했습니다.)

AWS Config를 사용하여 Lambda에 대한 예방적 제어 구현

가능하면 개발 프로세스 초기에 서버리스 애플리케이션의 규정 준수를 보장하는 것이 중요합니다. 이 주제에서는 AWS Config를 사용하여 사전 예방적 제어를 구현하는 방법을 다룹니다. 이를 통해 개발 프로세스 초기에 규정 준수 검사를 구현하고 CI/CD 파이프라인에서 동일한 제어를 구현할 수 있습니다. 또한 AWS 계정 전체에서 제어를 일관되게 적용할 수 있도록 중앙에서 관리되는 규칙 리포지토리에서 제어를 표준화합니다.

예를 들어 규정 준수 관리자가 모든 Lambda 함수에 AWS X-Ray 추적 기능을 포함하는 요구 사항을 정의한다고 가정합니다. AWS Config의 사전 예방 모드를 사용하면 배포 전에 Lambda 함수 리소스에서 규정 준수 검사를 실행하여 잘못 구성된 Lambda 함수를 배포할 위험을 줄이고 코드형 인프라 템플릿에서 피드백을 더 빠르게 제공하여 개발자의 시간을 절약할 수 있습니다. 다음은 AWS Config를 사용하는 사전 예방적 제어의 흐름을 시각화한 것입니다.



모든 Lambda 함수에 추적 기능이 활성화되어 있어야 하는 요구 사항을 고려합니다. 이에 대한 응답으로 플랫폼 팀은 모든 계정에서 사전 예방적으로 특정 AWS Config 규칙을 실행해야 할 필요성을 식별합니다. 이 규칙은 구성된 X-Ray 트레이싱 구성이 없는 모든 Lambda 함수에 규정 미준수 리소스로 플래그를 지정합니다. 팀은 규칙을 개발하고, 이를 적합성 팩으로 패키징한 후, 조직의 모든 계정이 이

러한 제어를 균일하게 적용하도록 모든 AWS 계정에 적합성 팩을 배포합니다. 다음 형식을 사용하는 AWS CloudFormation Guard 2.x.x 구문으로 규칙을 작성할 수 있습니다.

```
rule name when condition { assertion }
```

다음은 Lambda 함수에 트레이싱 기능이 활성화되어 있는지 확인하는 샘플 Guard 규칙입니다.

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

플랫폼 팀은 모든 AWS CloudFormation 배포에서 사전 생성 또는 업데이트 후크를 간접 호출하도록 요구하여 추가 조치를 수행합니다. 이 후크를 개발하고 파이프라인을 구성하며 규정 준수 규칙의 중앙 집중식 제어를 강화하고 모든 배포에서 일관된 적용을 유지하는 일은 플랫폼 팀의 책임입니다. 후크를 개발, 패키징 및 등록하려면 CloudFormation 명령줄(CFN-CLI) 설명서의 AWS CloudFormation 후크 개발을 참조하세요. CloudFormation CLI를 사용하여 후크 프로젝트를 생성할 수 있습니다.

```
cfn init
```

이 명령은 후크 프로젝트에 대한 몇 가지 기본 정보를 요청하고 다음 파일이 포함된 프로젝트를 생성합니다.

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

후크 개발자인 경우 <hook-name>.json 구성 파일에 원하는 대상 리소스 유형을 추가해야 합니다. 아래 구성에서는 CloudFormation을 사용하여 Lambda 함수를 생성하기 전에 후크가 실행되도록 구성되어 있습니다. preUpdate 및 preDelete 작업에도 유사한 핸들러를 추가할 수 있습니다.

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
```

```
            "permissions": []
        }
    }
```

또한 CloudFormation 후크가 AWS Config API를 직접 호출할 수 있는 적절한 권한을 보유하는지도 확인해야 합니다. `hook-role.yaml`이라는 역할 정의 파일을 업데이트하여 이 작업을 수행할 수 있습니다. 역할 정의 파일에는 기본적으로 다음과 같은 신뢰 정책이 있으며, 이를 통해 CloudFormation이 역할을 수임할 수 있습니다.

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

이 후크가 구성 API를 직접 호출하도록 허용하려면 정책 명령에 다음 권한을 추가해야 합니다. 그런 다음, `cfn submit` 명령을 사용하여 후크 프로젝트를 제출합니다. 그러면 CloudFormation에서 필요한 권한을 가진 역할을 자동으로 생성합니다.

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

다음으로 `src/handler.py` 파일에 Lambda 함수를 작성해야 합니다. 이 파일에는 프로젝트 시작 시 이름이 `preCreate`, `preUpdate`, `preDelete`인 메서드가 이미 생성되어 있습니다. AWS SDK for Python (Boto3)을 사용하여 사전 예방적 모드에서 AWS Config `StartResourceEvaluation` API를 직접 호출하는 재사용 가능한 공통 함수를 작성하는 것을 목표로 합니다. 이 API 직접 호출은 리소스 속성을 입력으로 사용하고 규칙 정의를 기준으로 리소스를 평가합니다.

```
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.")
```

이제 사전 생성 후크의 핸들러에서 공통 함수를 직접 호출할 수 있습니다. 다음은 핸들러 예제입니다.

```
@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}")
```

이 단계 이후에 후크를 등록하고 모든 AWS Lambda 함수 생성 이벤트를 수신하도록 구성할 수 있습니다.

개발자가 Lambda를 사용하여 서버리스 마이크로서비스를 위한 코드형 인프라(IaC) 템플릿을 준비합니다. 이 준비 작업에는 내부 표준을 준수한 후 템플릿을 로컬에서 테스트하고 리포지토리에 커밋하는 작업이 포함됩니다. 다음은 예제 IaC 템플릿입니다.

```
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 프로세스의 일부로 CloudFormation 템플릿이 배포되면 CloudFormation 서비스는 `AWS::Lambda::Function` 리소스 유형을 프로비저닝하기 바로 전에 사전 생성 또는 업데이트 후크를 간접 호출합니다. 후크는 사전 예방적 모드에서 실행되는 AWS Config 규칙을 활용하여 Lambda 함수 구성에 필수 트레이싱 구성이 포함되어 있는지 확인합니다. 후크의 응답에 따라 다음 단계가 결정됩니다. 규정을 준수하는 경우 후크는 성공 신호를 보내고 CloudFormation은 리소스 프로비저닝을 진행합니다. 그렇지 않으면 CloudFormation 스택 배포가 실패하고 파이프라인이 즉시 중단되며 시스템은 후속 검토를 위해 세부 정보를 기록합니다. 규정 준수 알림이 관련 이해관계자에게 전송됩니다.

CloudFormation 콘솔에서 후크 성공 또는 실패 정보를 찾을 수 있습니다.

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 후크에 대해 로그가 활성화된 경우 후크 평가 결과를 캡처할 수 있습니다. 다음은 실패 상태의 후크에 대한 샘플 로그로, Lambda 함수에서 X-Ray가 활성화되지 않았음을 나타냅니다.

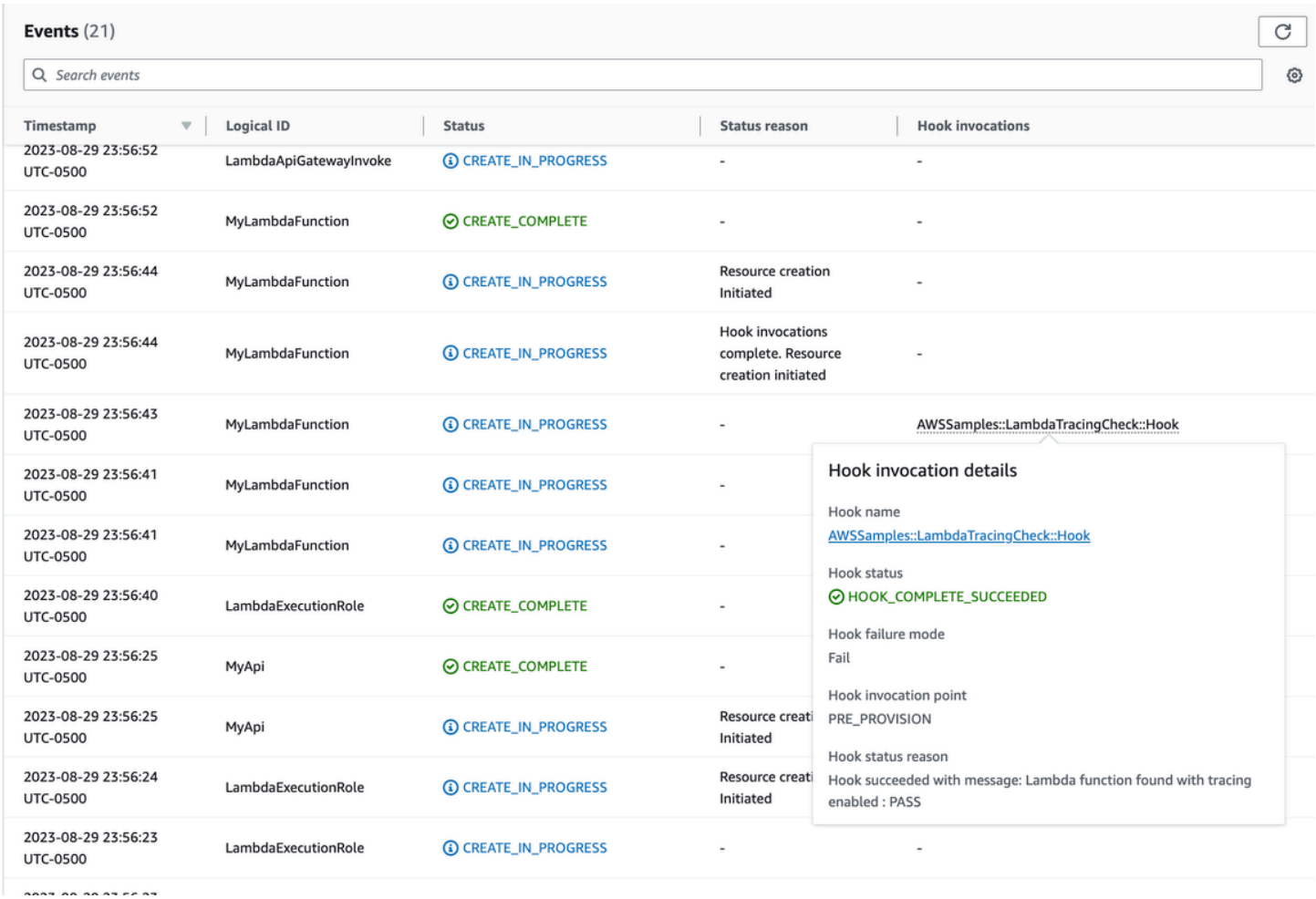
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

개발자가 IaC를 변경하여 `TracingConfig Mode` 값을 `Active`로 업데이트하고 다시 배포하도록 선택하면 후크가 성공적으로 실행되고 스택에서 Lambda 리소스 생성을 진행합니다.



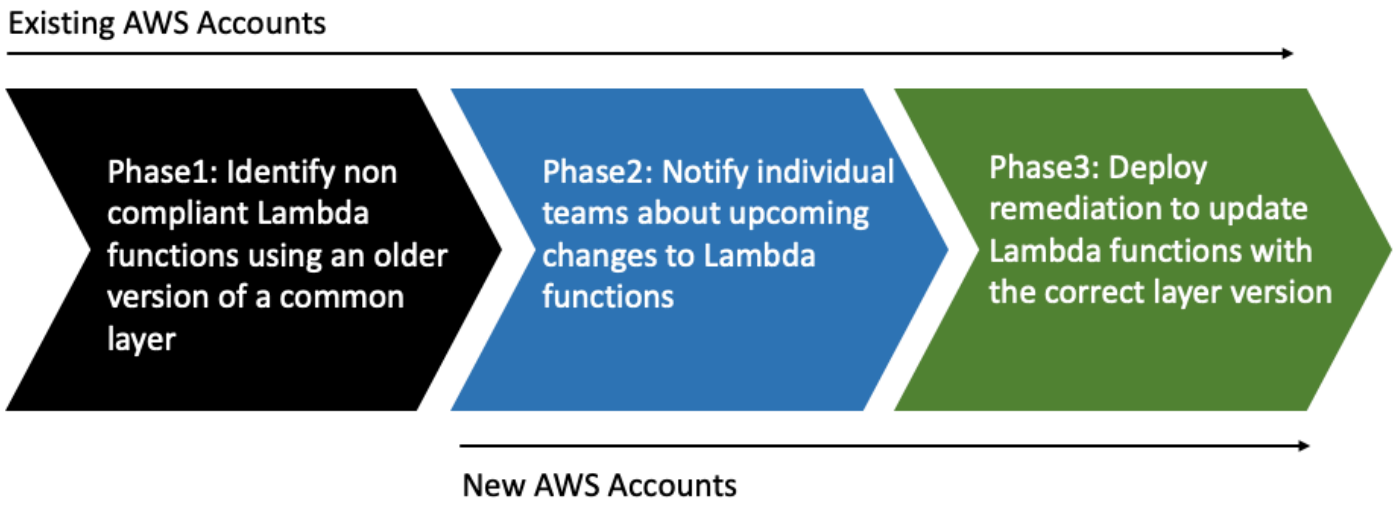
이렇게 하면 AWS 계정에서 서버리스 리소스를 개발하고 배포할 때 AWS Config에서 사전 예방적 모드로 예방 제어를 구현할 수 있습니다. AWS Config 규칙을 CI/CD 파이프라인에 통합하면 활성 트레이싱 구성이 없는 Lambda 함수와 같이 규정 미준수 리소스 배포를 식별하고 선택적으로 차단할 수 있습니다. 이렇게 하면 최신 거버넌스 정책을 준수하는 리소스만 AWS 환경에 배포할 수 있습니다.

AWS Config를 사용하여 규정 미준수 Lambda 배포 및 구성 감지

선제적 평가 외에도 AWS Config에서는 거버넌스 정책을 준수하지 않는 리소스 배포 및 구성을 사후 대응적으로 감지할 수 있습니다. 이 기능은 조직이 새로운 모범 사례를 학습하고 구현함에 따라 거버넌스 정책도 발전하기 때문에 중요합니다.

Lambda 함수를 배포하거나 업데이트할 때 완전히 새로운 정책을 설정하는 시나리오를 고려합니다. 모든 Lambda 함수는 항상 승인된 특정 Lambda 계층 버전을 사용해야 합니다. 계층 구성을 위해 새 함수 또는 업데이트된 함수를 모니터링하도록 AWS Config를 구성할 수 있습니다. AWS Config에서 승인된 계층 버전을 사용하지 않는 함수를 탐지하면 해당 함수에 비준수 리소스로 플래그를 지정합니다. 선택적으로 AWS Systems Manager 자동화 문서를 사용하여 수정 조치를 지정함으로써 리소스를 자동으로 수정하도록 AWS Config를 구성할 수 있습니다. 예를 들어, AWS SDK for Python (Boto3)을 사용하여 Python에서 승인된 계층 버전을 가리키도록 규정 미준수 함수를 업데이트하는 자동화 문서를 작성할 수 있습니다. 이를 통해 AWS Config에서는 감지 및 수정 제어 기능을 모두 지원하며 규정 준수 관리를 자동화합니다.

이 프로세스를 세 가지 중요한 구현 단계로 나누어 보겠습니다.



1단계: 액세스 리소스 식별

먼저 여러 계정을 AWS Config를 활성화하고 AWS Lambda 함수를 기록하도록 구성합니다. 그러면 AWS Config에서 Lambda 함수가 생성되거나 업데이트되는 시점을 관찰할 수 있습니다. 그런 다음, 특정 정책 위반을 검사하도록 사용자 지정 정책 규칙을 구성합니다. 이때 AWS CloudFormation Guard 구문을 사용합니다. Guard 규칙은 다음과 같은 일반적인 양식을 사용합니다.

```
rule name when condition { assertion }
```

다음은 계층이 이전 계층 버전으로 설정되지 않았는지 확인하는 샘플 규칙입니다.

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

규칙 구문과 구조를 살펴봅니다.

- 규칙 이름: 제공된 예제의 규칙 이름은 `desiredlayer`입니다.
- 조건: 이 절은 규칙을 확인해야 하는 조건을 지정합니다. 제공된 예제에서 조건은 `configuration.layers !empty`입니다. 즉, 구성의 `layers` 속성이 비어 있지 않은 경우에만 리소스를 평가해야 합니다.
- 어설션: `when` 절 뒤의 어설션은 규칙이 검사하는 대상을 결정합니다. `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` 어설션은 Lambda 계층 ARN이 `OldLayerArn` 값과 일치하지 않는지 확인합니다. 일치하지 않으면 어설션은 `true`이고 규칙은 통과됩니다. 그렇지 않으면 실패합니다.

`CONFIG_RULE_PARAMETERS`는 AWS Config 규칙으로 구성된 특별한 파라미터 세트입니다. 이 경우 `OldLayerArn`은 `CONFIG_RULE_PARAMETERS` 내부의 파라미터입니다. 이를 통해 사용자는 오래되었거나 더 이상 사용되지 않는 것으로 간주되는 특정 ARN 값을 제공할 수 있으며, 이후 이 규칙은 Lambda 함수가 이 오래된 ARN을 사용하고 있는지 확인합니다.

2단계: 시각화 및 설계

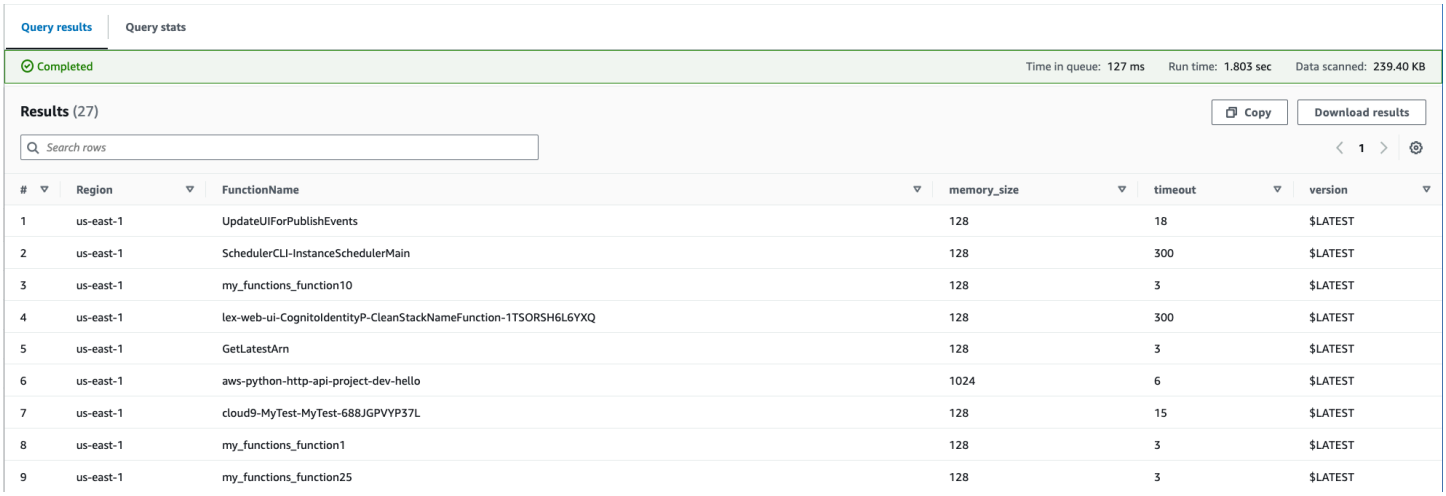
AWS Config에서는 구성 데이터를 수집하고 Amazon Simple Storage Service(Amazon S3) 버킷에 저장합니다. Amazon Athena를 사용하여 S3 버킷에서 직접 이 데이터를 쿼리할 수 있습니다. Athena를 사용하면 조직 수준에서 이 데이터를 집계하여 모든 계정의 리소스 구성을 전체적으로 파악할 수 있습니다. 리소스 구성 데이터의 집계를 설정하려면 AWS 클라우드 운영 및 관리 블로그에서 Visualizing AWS Config data using Athena and Amazon QuickSight를 참조하세요.

다음은 특정 계층 ARN을 사용하여 모든 Lambda 함수를 식별하기 위한 샘플 Athena 쿼리입니다.

```
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%'
```

쿼리 결과는 다음과 같습니다.



Query results | Query stats

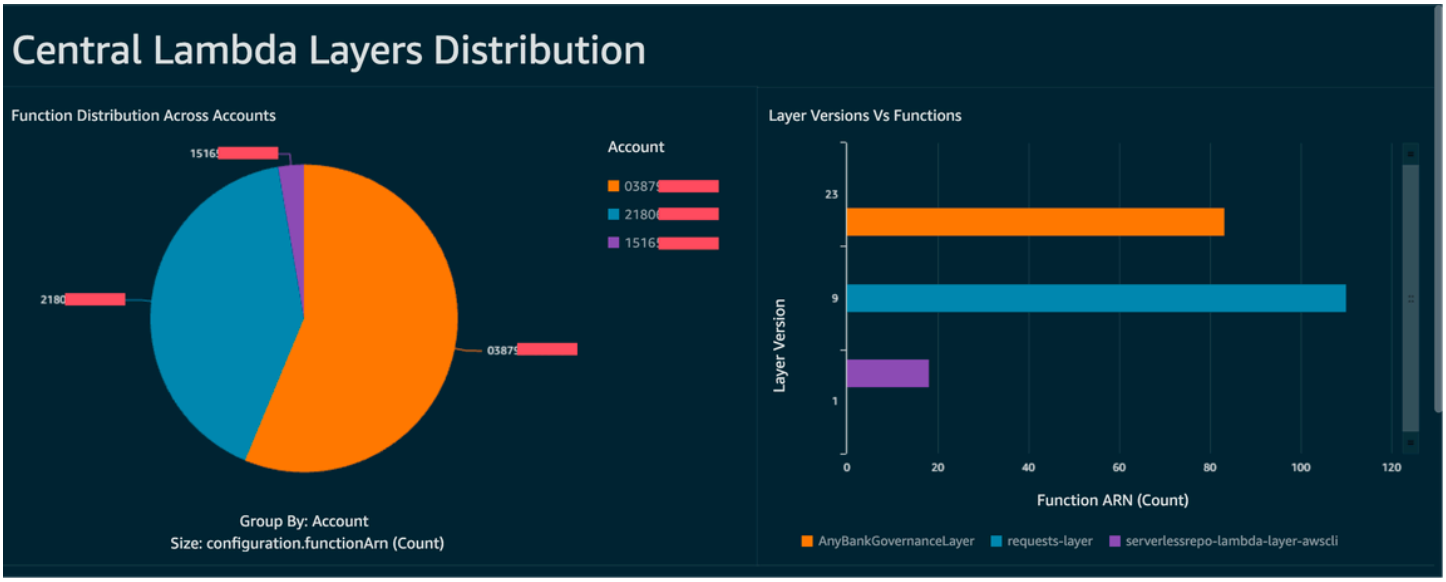
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

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

조직 전체에서 집계된 AWS Config 데이터를 사용하여 Amazon QuickSight를 통해 대시보드를 생성할 수 있습니다. Athena 결과를 Amazon QuickSight로 가져오면 Lambda 함수가 계층 버전 규칙을 얼마나 잘 준수하는지 시각화할 수 있습니다. 이 대시보드는 규정 준수 및 규정 미준수 리소스를 강조 표시하므로, 다음 섹션에 설명된 대로 적용 정책을 결정하는 데 도움이 됩니다. 다음 이미지는 조직 내 기능에 적용된 계층 버전의 분포를 보고하는 대시보드 예제입니다.



3단계: 구현 및 적용

이제 선택적으로 AWS SDK for Python (Boto3)에서 쓴 Python 스크립트로 작성한 Systems Manager 자동화 문서를 통해 1단계에서 생성한 계층 버전 규칙을 수정 조치와 페어링할 수 있습니다. 스크립트는 각 Lambda 함수에 대해 UpdateFunctionConfiguration API 작업을 직접 호출하여 새 계층 ARN으로 함수 구성을 업데이트합니다. 또는 스크립트가 코드 리포지토리에 풀 요청을 제출하여 계층 ARN을 업데이트하도록 할 수도 있습니다. 그러면 향후 코드 배포 시에도 올바른 계층 ARN으로 업데이트됩니다.

AWS Signer를 사용하여 Lambda 코드 서명

AWS Signer는 완전 관리형 코드 서명 서비스로, 디지털 서명을 대상으로 코드를 검증하여 코드가 변경되지 않고 신뢰할 수 있는 게시자가 제공한 코드인지 확인할 수 있습니다. AWS Signer는 AWS Lambda와 함께 사용하여 AWS 환경에 배포하기 전에 함수와 계층이 변경되지 않았는지 확인할 수 있습니다. 이를 통해 새 함수를 생성하거나 기존 함수를 업데이트하기 위해 보안 인증 정보를 획득했을 수 있는 악의적인 행위자로부터 조직을 보호할 수 있습니다.

Lambda 함수에 대한 코드 서명을 설정하려면 먼저 버전 관리가 활성화된 S3 버킷을 생성합니다. 그런 다음, AWS Signer에서 서명 프로파일을 생성하고 Lambda를 플랫폼으로 지정한 후 서명 프로파일의 유효 기간(일)을 지정합니다. 예제

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

그런 다음, 서명 프로파일을 사용하고 Lambda로 서명 구성을 생성합니다. 서명 구성에서 예상한 디지털 서명과 일치하지 않는 아티팩트를 찾은 경우 수행할 작업(경고(배포 허용) 또는 적용(배포 차단))을 지정해야 합니다. 아래 예제는 배포를 적용하고 차단하도록 구성되어 있습니다.

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

이제 신뢰할 수 없는 배포를 차단하도록 Lambda에서 AWS Signer를 구성했습니다. 기능 요청 코딩을 마쳤고 이제 함수를 배포할 준비가 되었다고 가정합니다. 첫 번째 단계는 적절한 종속 항목으로 코드를 압축한 다음, 생성한 서명 프로파일을 사용하여 아티팩트에 서명하는 것입니다. zip 아티팩트를 S3 버킷에 업로드한 다음, 서명 작업을 시작하면 이 작업을 수행할 수 있습니다.

```
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
```

결과는 다음과 같습니다. 여기서 jobId는 대상 버킷에서 생성된 객체 및 접두사이며 jobOwner는 작업이 실행된 12자리 AWS 계정 ID입니다.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

이제 서명된 S3 객체 및 생성한 코드 서명 구성을 사용하여 함수를 배포할 수 있습니다.

```
  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
```

서명되지 않은 원래 소스 zip 아티팩트를 사용하여 함수 배포를 테스트할 수도 있습니다. 배포에 실패하고 다음 메시지가 표시됩니다.

```
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)을 사용하여 함수를 빌드하고 배포하는 경우 패키지 명령은 S3로의 zip 아티팩트 업로드를 처리하고 서명 작업을 시작한 후 서명된 아티팩트를 가져옵니다. 이를 위해 다음 명령과 파라미터를 사용합니다.

```
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 Signer는 계정에 배포된 zip 아티팩트가 배포 시 신뢰할 수 있는지 확인하는 데 도움이 됩니다. 위의 프로세스를 CI/CD 파이프라인에 포함하고 이전 주제에서 설명한 방법을 사용하여 모든 함수에 코드 서명 구성을 연결하도록 요구할 수 있습니다. Lambda 함수 배포에서 코드 서명을 사용하면 함수를 생성하거나 업데이트하기 위해 보안 인증 정보를 획득했을 수 있는 악의적인 행위자가 함수에 악성 코드를 삽입하는 것을 방지할 수 있습니다.

Amazon Inspector를 사용하여 Lambda에 대한 보안 평가 자동화

Amazon Inspector는 알려진 소프트웨어 취약성 및 의도하지 않은 네트워크 노출이 있는 워크로드를 지속적으로 스캔하는 취약성 관리 서비스입니다. Amazon Inspector는 취약성을 설명하고, 영향을 받는 리소스를 식별하며, 취약성의 심각도를 평가하고, 해결 지침을 제공하는 조사 결과를 생성합니다.

Amazon Inspector 지원은 Lambda 함수 및 계층에 대한 지속적이고 자동화된 보안 취약성 평가를 제공합니다. Amazon Inspector는 Lambda에 대해 두 가지 유형의 스캔을 제공합니다.

- Lambda 표준 스캔(기본값): Lambda 함수 및 해당 계층 내의 애플리케이션 종속 항목을 스캔하여 패키지 취약성을 찾습니다.
- Lambda 코드 스캔: 함수 및 계층에서 사용자 지정 애플리케이션 코드를 스캔하여 코드 취약성을 찾습니다. Lambda 표준 스캔을 활성화하거나, Lambda 코드 스캔과 함께 Lambda 표준 스캔을 활성화할 수 있습니다.

Amazon Inspector를 활성화하려면 Amazon Inspector 콘솔로 이동하고 설정 섹션을 확장한 다음, 계정 관리를 선택합니다. 계정 탭에서 활성화를 선택한 다음, 스캔 옵션 중 하나를 선택합니다.

Amazon Inspector를 설정하는 동안 여러 계정에 대해 Amazon Inspector를 활성화하고 조직의 Amazon Inspector를 관리할 권한을 특정 계정에 위임할 수 있습니다. 활성화하는 동안 역할(`AWSServiceRoleForAmazonInspector2`)을 생성하여 Amazon Inspector에 권한을 부여해야 합니다. Amazon Inspector 콘솔에서는 원클릭 옵션을 사용하여 이 역할을 생성할 수 있습니다.

Lambda 표준 스캔의 경우 Amazon Inspector는 다음과 같은 경우에 Lambda 함수의 취약성 스캔을 시작합니다.

- Amazon Inspector에서 기존 Lambda 함수를 발견하는 즉시
- 새 Lambda 함수를 배포하는 경우
- 기존 Lambda 함수 또는 해당 계층의 애플리케이션 코드 또는 종속성에 대한 업데이트를 배포하는 경우
- Amazon Inspector가 새로운 일반적인 취약성 및 노출(CVE) 항목을 데이터베이스에 추가하고, 해당 CVE가 함수와 관련이 있는 경우

Lambda 코드 스캔의 경우 Amazon Inspector는 애플리케이션 코드의 전반적인 보안 규정 준수를 분석하는 자동 추론 및 기계 학습을 사용하여 Lambda 함수 애플리케이션 코드를 평가합니다. Amazon Inspector에서 Lambda 함수 애플리케이션 코드의 취약성을 탐지한 경우 Amazon Inspector는 상세한

코드 취약성 결과를 생성합니다. 가능한 탐지 목록은 Amazon CodeGuru Detector Library를 참조하세요.

결과를 보려면 Amazon Inspector 콘솔로 이동합니다. 결과 메뉴에서 Lambda 함수 기준을 선택하여 Lambda 함수에서 수행한 보안 스캔 결과를 표시합니다.

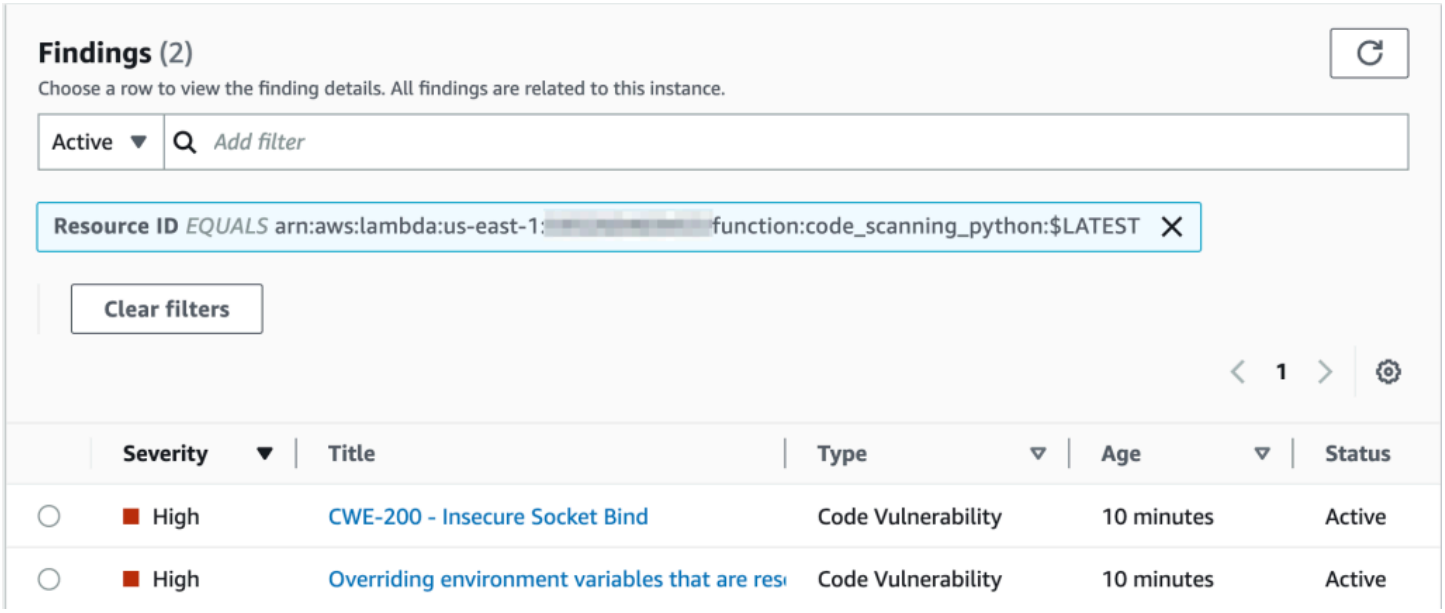
Lambda 함수를 표준 스캔에서 제외하려면 다음 키 값 페어를 사용하여 함수에 태그를 지정합니다.

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

Lambda 함수를 코드 스캔에서 제외하려면 다음 키 값 페어를 사용하여 함수에 태그를 지정합니다.

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

예를 들어, 다음 이미지에 표시된 대로, Amazon Inspector는 자동으로 취약성을 감지하고 결과를 코드 취약성 유형으로 분류합니다. 코드 종속 라이브러리가 아니라 함수의 코드에 취약성이 있음을 나타냅니다. 특정 함수 또는 여러 함수에 대한 이러한 세부 정보를 한 번에 확인할 수 있습니다.



이러한 각 결과를 자세히 살펴보고 문제를 해결하는 방법을 알아볼 수 있습니다.

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 함수를 작업할 때는 Lambda 함수의 이름 지정 규칙을 준수해야 합니다. 자세한 내용은 Lambda 환경 변수를 사용하여 코드의 값 구성 단원을 참조하십시오.

제안된 해결 방법을 수락할 경우 이에 따른 책임은 귀하에게 있습니다. 그러므로 제안된 해결 방법을 수락하기 전에 항상 검토하세요. 코드가 의도한 대로 작동하도록 제안된 해결 방법을 수정해야 할 수도 있습니다.

Lambda 보안 및 규정 준수를 위한 관찰성 구현

AWS Config는 규정 미준수 AWS 서버리스 리소스를 찾아 수정하는 데 유용한 도구입니다. 서버리스 리소스에서 변경한 내용은 모두 AWS Config에 기록됩니다. 또한 AWS Config에서는 구성 스냅샷 데이터를 S3에 저장할 수 있습니다. Amazon Athena와 QuickSight Amazon을 사용하여 대시보드를 만들고 데이터를 볼 수 있습니다. AWS Config [AWS Config를 사용하여 규정 미준수 Lambda 배포 및 구성 감지](#)에서는 Lambda 계층과 같은 특정 구성을 시각화하는 방법을 설명했습니다. 이 주제에서는 이러한 개념을 확장합니다.

Lambda 구성에 대한 가시성

쿼리를 사용하여 AWS 계정 ID, 리전, AWS X-Ray 트레이싱 구성, VPC 구성, 메모리 크기, 런타임, 태그와 같은 중요한 구성을 가져올 수 있습니다. 다음은 Athena에서 이 정보를 가져오는 데 사용할 수 있는 샘플 쿼리입니다.

```
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
```

쿼리를 사용하여 Amazon QuickSight 대시보드를 구축하고 데이터를 시각화할 수 있습니다. AWS리소스 구성 데이터를 집계하고, Athena에서 테이블을 생성하고, Athena의 데이터를 기반으로 QuickSight Amazon 대시보드를 구축하려면 클라우드 운영 및 관리 블로그에서 Athena와 QuickSight Amazon을 AWS Config 사용한 데이터 시각화를 참조하십시오. AWS 특히 이 쿼리는 함수에 대한 태그 정보도 검색합니다. 이를 통해 특히 사용자 지정 태그를 사용하는 경우 워크로드와 환경에 대한 심층적인 인사이트를 얻을 수 있습니다.

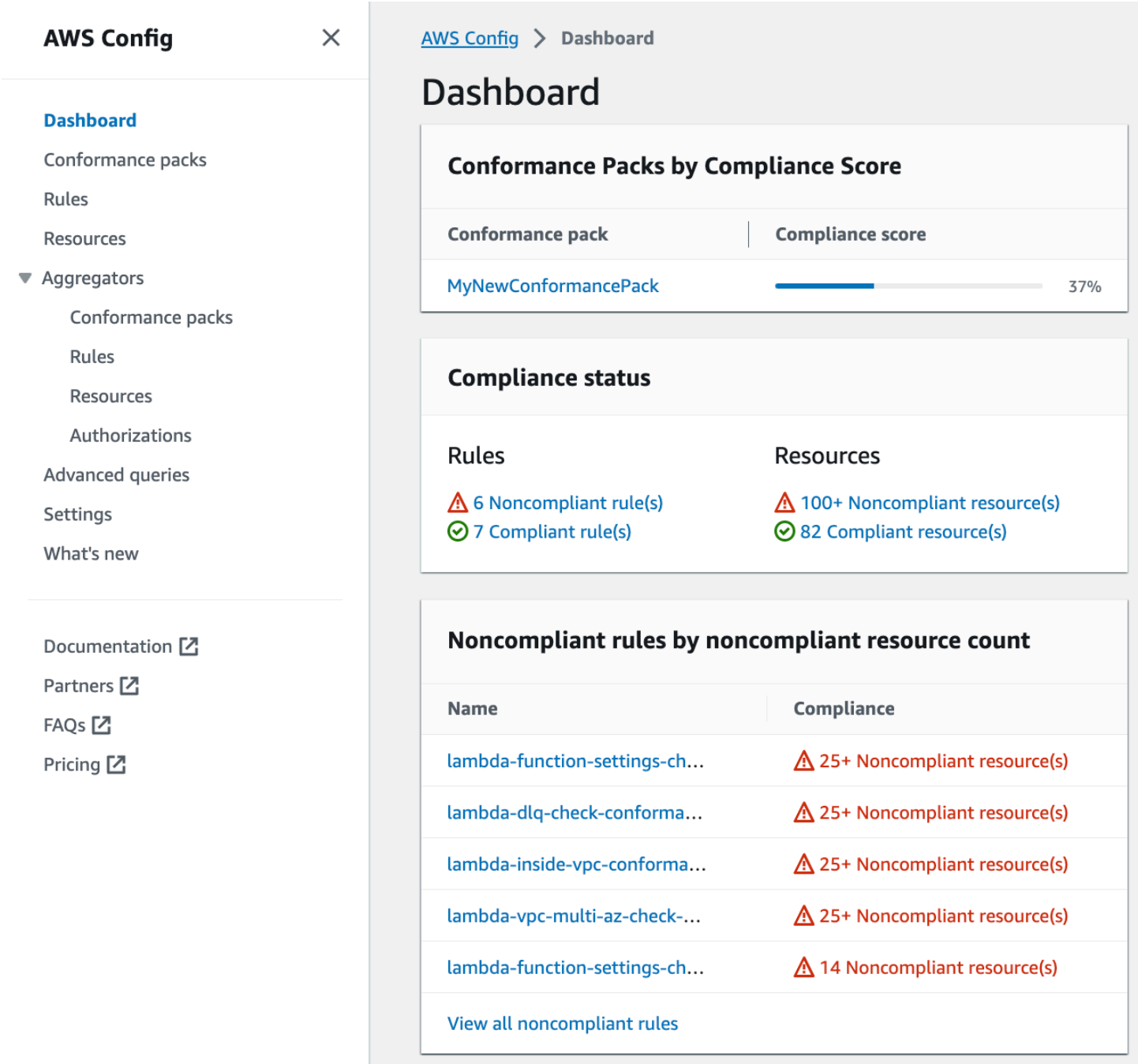
수행할 수 있는 작업에 대한 자세한 내용은 이 주제의 후반부에 나오는 관찰성 결과 처리 섹션을 참조하세요.

Lambda 규정 준수에 대한 가시성

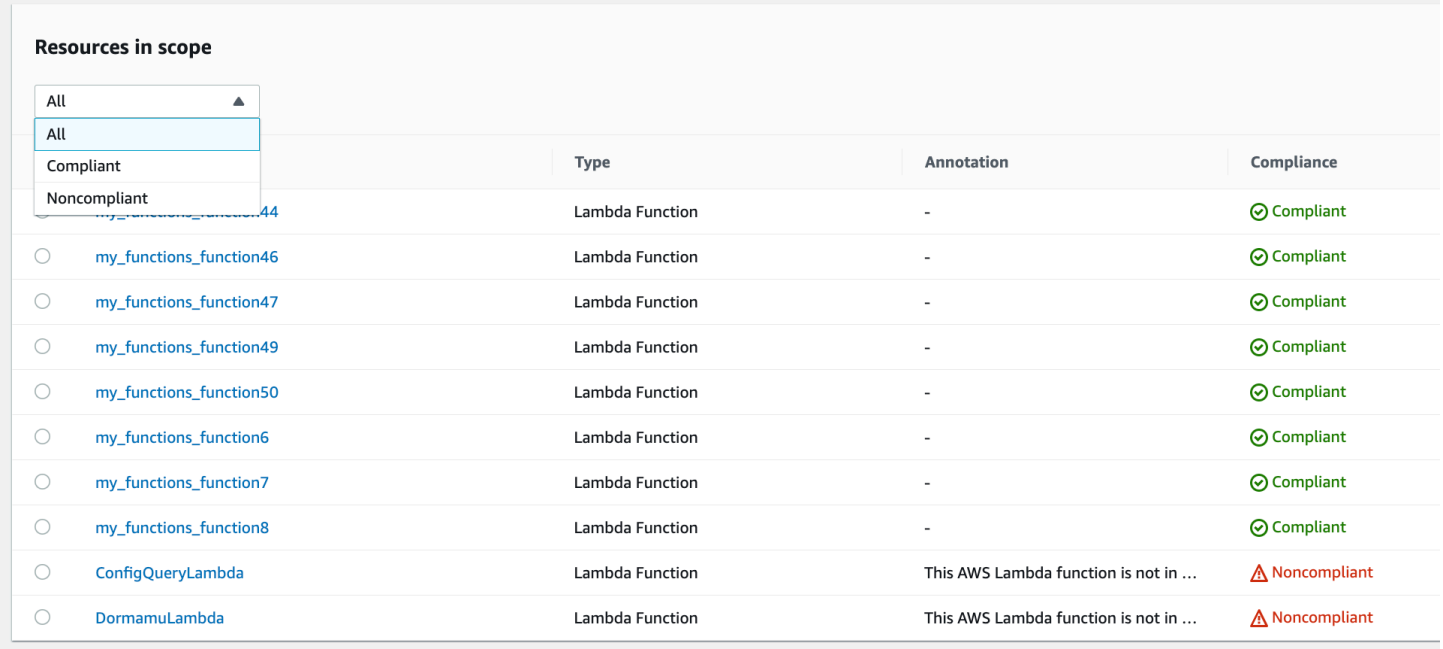
AWS Config에서 생성한 데이터를 사용하여 조직 수준의 대시보드를 만들어 규정 준수를 모니터링할 수 있습니다. 이를 통해 다음을 일관되게 추적하고 모니터링할 수 있습니다.

- 규정 준수 점수별 규정 준수 팩
- 규정 미준수 리소스별 규칙

- 규정 준수 상태

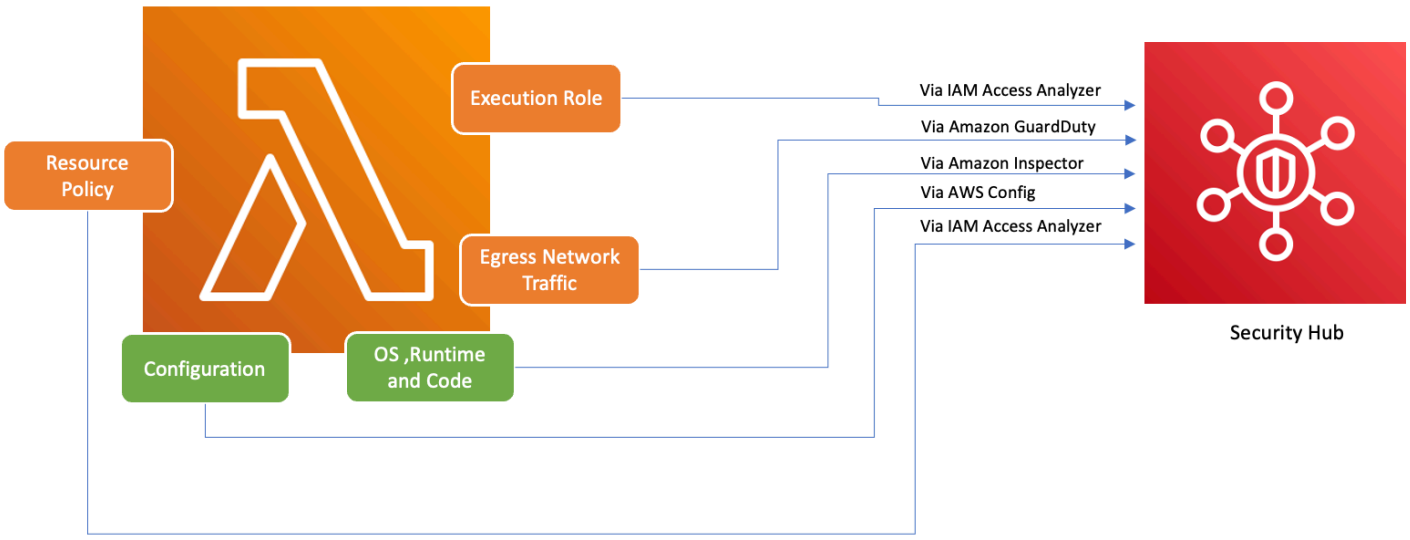


각 규칙을 확인하여 해당 규칙에 대한 규정 미준수 리소스를 식별합니다. 예를 들어, 조직에서 모든 Lambda 함수를 VPC와 연결하도록 규정하고 규정 준수를 식별하는 AWS Config 규칙을 배포한 경우 위 목록에서 `lambda-inside-vpc` 규칙을 선택할 수 있습니다.



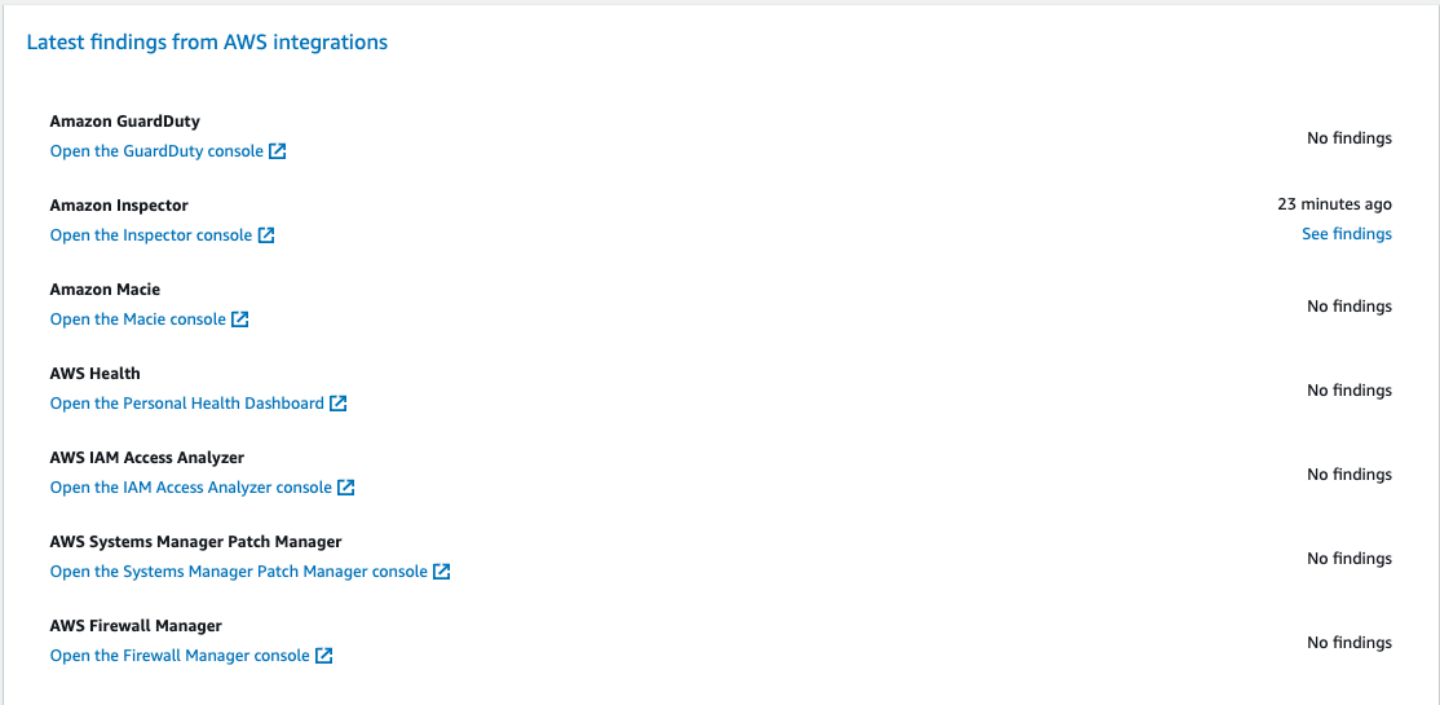
수행할 수 있는 작업에 대한 자세한 내용은 아래 관찰성 결과 처리 섹션을 참조하세요.

Security Hub를 사용하여 Lambda 함수 경계에 대한 가시성 확보

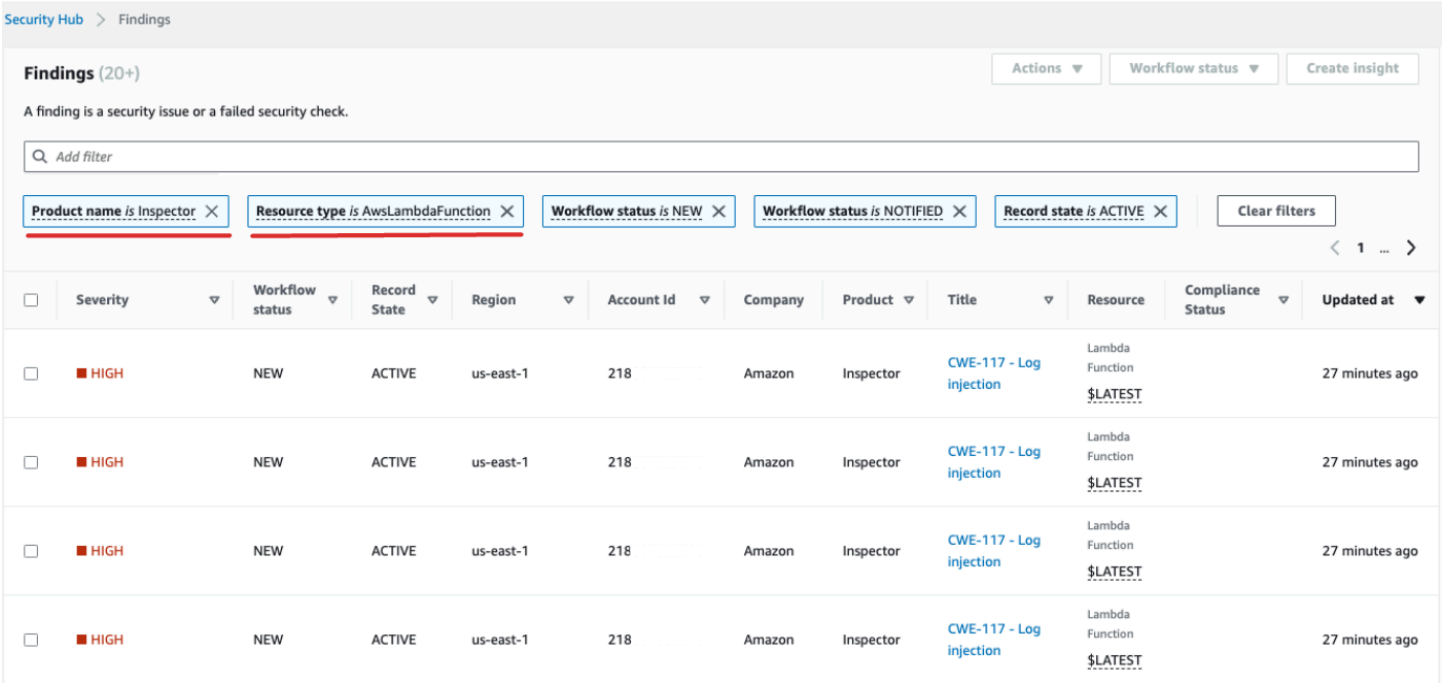


Lambda를 비롯한 AWS 서비스를 안전하게 사용할 수 있도록 AWS에서는 Foundational Security Best Practices v1.0.0을 도입했습니다. 이 모범 사례 세트는 강력한 보안 태세 유지의 중요성을 강조하면서 AWS 환경의 리소스 및 데이터 보안을 위한 명확한 지침을 제공합니다. AWS Security Hub는 통합 보안 및 규정 준수 센터를 제공하여 이를 보완합니다. AWSAmazon Inspector 및 Amazon과 같은 여러 서비스의 보안 결과를 집계하고 구성하며 우선 순위를 지정합니다. AWS Identity and Access Management Access Analyzer GuardDuty

Security Hub, Amazon Inspector, IAM 액세스 분석기가 있고 조직 내에 GuardDuty 활성화되어 있는 경우 AWS Security Hub는 이러한 서비스의 결과를 자동으로 집계합니다. 예를 들어 Amazon Inspector를 고려합니다. Security Hub를 사용하면 Lambda 함수의 코드 및 패키지 취약성을 효율적으로 식별할 수 있습니다. Security Hub 콘솔에서 하단의 AWS 통합의 최근 분석 결과 섹션으로 이동합니다. 여기에서 통합된 다양한 AWS 서비스에서 가져온 결과를 보고 분석할 수 있습니다.



세부 정보를 보려면 두 번째 열의 결과 보기 링크를 선택합니다. 여기에는 Amazon Inspector와 같이 제품별로 필터링된 결과 목록이 표시됩니다. 검색을 Lambda 함수로 제한하려면 `ResourceType`을 `AwsLambdaFunction`으로 설정합니다. 여기에는 Lambda 함수와 관련된 Amazon Inspector의 결과가 표시됩니다.



를 위해 GuardDuty 의심스러운 네트워크 트래픽 패턴을 식별할 수 있습니다. 이러한 이상 현상은 Lambda 함수 내에 잠재적인 악성 코드가 존재함을 암시할 수 있습니다.

IAM Access Analyzer를 사용하면 정책, 특히 외부 엔터티에 대한 함수 액세스 권한을 부여하는 조건문이 있는 정책을 확인할 수 있습니다. 또한 IAM 액세스 분석기는 Lambda API에서 AddPermission작업을 사용할 때 설정된 권한을 평가합니다. `EventSourceToken`

관찰성 결과 처리

Lambda 함수에 사용할 수 있는 광범위한 구성과 고유한 요구 사항을 고려할 때 문제 해결을 위한 표준화된 자동화 솔루션이 모든 상황에 적합하지 않을 수도 있습니다. 또한 변경 사항은 다양한 환경에서 다르게 구현됩니다. 호환되지 않는 구성이 있으면 다음 지침을 고려합니다.

1. 태그 지정 전략

   포괄적인 태그 지정 전략을 구현하는 것이 좋습니다. 각 Lambda 함수에는 다음과 같은 주요 정보로 태그를 지정해야 합니다.

   - 소유자: 함수를 담당하는 사람 또는 팀.
   - 환경: 프로덕션, 스테이징, 개발 또는 샌드박스.
   - 애플리케이션: 해당하는 경우 이 함수가 속한 더 넓은 컨텍스트.

2. 소유자 아웃리치

주요 변경 사항(예: VPC 구성 조정)을 자동화하는 대신, 규정 미준수 함수의 소유자(소유자 태그로 식별됨)에게 사전에 연락하여 다음 중 하나를 수행할 수 있는 충분한 시간을 확보합니다.

- Lambda 함수에서 규정 미준수 구성을 조정합니다.
- 설명을 제공하고 예외를 요청하거나 규정 준수 표준을 세분화합니다.

3. 구성 관리 데이터베이스(CMDB) 유지 관리

   태그는 즉각적인 컨텍스트를 제공하는 반면, 중앙 집중식 CMDB를 유지 관리하면 더 심층적인 인사이트를 제공할 수 있습니다. 각 Lambda 함수, 해당 종속 항목 및 기타 중요한 메타데이터에 대한 보다 세분화된 정보를 보관할 수 있습니다. CMDB는 감사, 규정 준수 검사 및 함수 소유자 식별을 위한 매우 중요한 리소스입니다.

서버리스 인프라 환경이 지속적으로 발전함에 따라 모니터링에 대한 사전 예방적 접근 태세를 취해야 합니다. AWS Config, Security Hub 및 Amazon Inspector와 같은 도구를 사용하면 잠재적 이상 또는 규정 미준수 구성을 신속하게 식별할 수 있습니다. 하지만 도구만으로는 완전한 규정 준수나 최적의 구성을 보장할 수 없습니다. 이러한 도구를 잘 문서화된 프로세스 및 모범 사례와 함께 사용하는 것이 중요합니다.

- 피드백 루프: 문제 해결 단계를 수행한 후 피드백 루프가 있는지 확인합니다. 즉, 규정 미준수 리소스를 주기적으로 다시 확인하여 해당 리소스가 업데이트되었는지 또는 동일한 문제가 계속 발생하는지 확인해야 합니다.
- 문서화: 항상 관찰 내용, 수행한 작업, 허용된 예외 사항을 문서화합니다. 적절한 문서화는 감사 중에 도움이 될 뿐만 아니라 향후 규정 준수 및 보안 강화를 위한 프로세스를 개선하는 데도 도움이 됩니다.
- 교육 및 인식: 모든 이해관계자, 특히 Lambda 함수 소유자가 모범 사례, 조직 정책 및 규정 준수 규정을 주기적으로 교육받고 인지해야 합니다. 정기 워크숍, 웨비나 또는 교육 세션은 보안 및 규정 준수와 관련하여 모든 사람이 동일한 이해 기반을 갖추는 데 큰 도움이 될 수 있습니다.

결론적으로, 도구와 기술은 잠재적 문제를 감지하고 플래그를 지정하는 강력한 기능을 제공하지만, 이해, 소통, 교육 및 문서화와 같은 인적 요소도 여전히 중요합니다. 이 둘을 함께 활용하면 Lambda 기능 및 광범위한 인프라가 규정을 준수하고 안전하며 비즈니스 요구 사항에 맞게 최적화되도록 보장하는 강력한 이점을 제공합니다.

AWS Lambda의 규정 준수 확인

타사 감사자는 여러 AWS Lambda 규정 준수 프로그램의 일환으로 AWS의 보안 및 규정 준수를 평가합니다. 여기에는 SOC, PCI, FedRAMP, HIPAA 등이 포함됩니다.

특정 규정 준수 프로그램 범위에 속하는 AWS 서비스의 목록은 규정 준수 프로그램 제공 AWS 범위 내 서비스를 참조하세요. 일반 정보는 AWS 규정 준수 프로그램을 참조하세요.

AWS Artifact를 사용하여 타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 AWS 아티팩트의 보고서 다운로드를 참조하세요.

Lambda 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률 및 규정에 따라 결정됩니다. 회사의 Lambda 함수가 규정 준수 요구 사항을 충족하도록 거버넌스 제어를 구현할 수 있습니다. 자세한 내용은 Lambda 함수 및 계층에 대한 거버넌스 전략 생성을(를) 참조하세요.

AWS Lambda의 복원성

AWS 글로벌 인프라는 AWS리전 및 가용 영역을 중심으로 구축됩니다. AWS 지역은 물리적으로 분리되고 격리된 다수의 가용 영역을 제공하며 이러한 가용 영역은 짧은 지연 시간, 높은 처리량 및 높은 중복성을 갖춘 네트워크에 연결되어 있습니다. 가용 영역을 사용하면 중단 없이 가용 영역 간에 자동으로 장애 조치가 이루어지는 애플리케이션 및 데이터베이스를 설계하고 운영할 수 있습니다. 가용 영역은 기존의 단일 또는 다중 데이터 센터 인프라보다 가용성, 내결함성, 확장성이 뛰어납니다.

AWS 리전 및 가용 영역에 대한 자세한 내용은 AWS 글로벌 인프라를 참조하세요.

AWS 글로벌 인프라뿐만 아니라 Lambda도 데이터 복원력과 백업 요구 사항을 지원하는 다양한 기능을 제공합니다.

- 버전 관리 – Lambda에서 버전 관리를 사용하여 함수 개발 시 그 코드와 구성을 저장할 수 있습니다. 별칭과 함께 버전 관리를 사용하여 블루/그린 및 롤링 배포를 수행할 수 있습니다. 자세한 내용은 Lambda 함수 버전 단원을 참조하세요.
- 확장 – 함수가 이전 요청을 처리하는 동안 요청을 받으면 Lambda는 함수의 다른 인스턴스를 실행하여 늘어난 로드를 처리합니다. Lambda는 리전당 1,000개의 동시 실행을 처리할 수 있도록 자동으로 확장되며, 필요한 경우 할당량을 증가시킬 수 있습니다. 자세한 내용은 Lambda 함수 규모 조정 이행 단원을 참조하세요.
- 고가용성 – Lambda는 함수를 여러 가용 영역에서 실행하여 단일 영역 내 서비스 중단 발생 시 이벤트를 처리할 수 있도록 합니다. 계정의 VPC(Virtual Private Cloud)로 연결되도록 함수를 구성하는 경우에는 여러 가용 영역의 서브넷을 지정해 고가용성을 보장합니다. 자세한 내용은 Lambda 함수에 Amazon VPC의 리소스에 대한 액세스 권한 부여 단원을 참조하세요.

- 예약 동시성 – 함수가 추가 요청을 처리할 수 있도록 항상 확장할 수 있게 해당 함수에 대한 동시성을 예약할 수 있습니다. 함수에 대해 예약 동시성을 설정하면 특정 수의 동시 호출까지 확장할 수 있습니다(이를 초과하지는 않음). 그러면 다른 함수가 가용 동시성을 모두 사용하여 요청을 놓치는 경우를 피할 수 있습니다. 자세한 내용은 함수에 대해 예약된 동시성 구성 단원을 참조하세요.
- 재시도 – 다른 서비스에 의해 트리거되는 호출의 서브셋과 비동기식 호출의 경우, Lambda는 재시도 간의 지연 시간을 두고 오류 시 재시도를 자동으로 수행합니다. 함수를 동기식으로 호출하는 다른 클라이언트와 AWS 서비스가 재시도 수행 기능을 담당합니다. 자세한 내용은 Lambda의 재시도 동작 이해 단원을 참조하세요.
- 배달 못한 편지 대기열 – 비동기식 호출의 경우 재시도가 모두 실패 시 배달 못한 편지 대기열로 요청을 전송하도록 Lambda를 구성할 수 있습니다. 배달 못한 편지 대기열은 문제 해결이나 재처리에 대한 이벤트를 수신하는 Amazon SNS 주제 또는 Amazon SQS 대기열입니다. 자세한 내용은 배달 못한 편지 대기열 단원을 참조하세요.

AWS Lambda에서 인프라 보안

관리형 서비스인 AWS Lambda은(는) AWS 글로벌 네트워크 보안으로 보호됩니다. AWS 보안 서비스와 AWS의 인프라 보호 방법에 관한 자세한 내용은 AWS 클라우드 보안을 참조하세요. 인프라 보안에 대한 모범 사례를 사용하여 AWS 환경을 설계하려면 보안 원칙 AWS Well-Architected Framework의 인프라 보호를 참조하십시오.

AWS에서 게시한 API 호출을 사용하여 네트워크를 통해 Lambda에 액세스합니다. 고객은 다음을 지원해야 합니다.

- 전송 계층 보안(TLS) TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군. Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

또한 요청은 액세스 키 ID 및 IAM 주체와 관련된 비밀 액세스 키를 사용하여 서명해야 합니다. 또는 AWS Security Token Service(AWS STS)를 사용하여 임시 보안 인증을 생성하여 요청에 서명할 수 있습니다.

Lambda 함수 모니터링 및 문제 해결

AWS Lambda는 다른 AWS 서비스와 통합하여 Lambda 함수를 모니터링하고 문제를 해결하는 데 도움을 줍니다. Lambda는 운영자를 대신해 자동으로 Lambda 함수를 모니터링하고 Amazon CloudWatch를 통해 측정치를 보고합니다. 실행 시 코드를 모니터링할 수 있도록, Lambda에서는 요청 수, 요청당 호출 기간 및 오류를 유발하는 요청 수를 자동으로 추적합니다.

다른 AWS 서비스를 사용하여 Lambda 함수 문제를 해결할 수 있습니다. 이 섹션에서는 이러한 AWS 서비스를 사용하여 Lambda 함수 및 애플리케이션을 모니터링, 추적, 디버깅 및 문제 해결하는 방법을 설명합니다. 각 런타임의 함수 로깅 및 오류에 대한 자세한 내용은 개별 런타임 섹션을 참조하세요.

Lambda 애플리케이션 모니터링에 대한 자세한 내용은 Serverless Land의 Monitoring and observability를 참조하세요.

Sections

Lambda 콘솔에서 함수 모니터링

Lambda 서비스는 사용자를 대신하여 기능을 모니터링하고 Amazon에 지표를 전송합니다. CloudWatch Lambda 콘솔은 이러한 지표의 그래프를 생성하고 각 Lambda 함수에 대한 모니터링 페이지에 이를 표시합니다.

Lambda 콘솔은 지표, 로그, 추적이 표시된 단일 창 보기를 제공합니다. 콘솔은 모든 창에 보편적으로 적용되는 시간 범위, 시간대 및 새로 고침 옵션에 대한 필터를 제공합니다. 지표, 로그 및 추적을 쉽게 상호 연관시켜서 Lambda 함수의 오류를 해결할 때 평균 복구 시간(MTTR) 을 줄일 수 있습니다.

요금

CloudWatch 영구 프리 티어가 제공됩니다. 프리 티어 한도를 초과하면 지표, 대시보드, 경보, 로그 및 인사이트에 대한 CloudWatch 요금이 부과됩니다. 자세한 내용은 Amazon CloudWatch 요금을 참조하십시오.

Lambda 콘솔 사용

Lambda 콘솔에서 Lambda 함수 및 애플리케이션을 모니터링할 수 있습니다.

함수를 모니터링하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 모니터링 탭을 선택합니다.

모니터링 그래프의 유형

다음 섹션에서는 Lambda 콘솔의 모니터링 그래프를 설명합니다.

Lambda 모니터링 그래프

- Invocations – 5분 기간 동안 함수가 호출된 횟수입니다.
- Duration(기간) – 함수 코드가 이벤트를 처리하는 데 소요하는 평균, 최소 및 최대 시간입니다.
- Error count and success rate (%)(오류 수 및 성공률)(%) – 오류 수 및 오류 없이 완료된 호출의 백분율입니다.
- Throttles(제한) – 동시성 한도로 인해 호출에 실패한 횟수입니다.

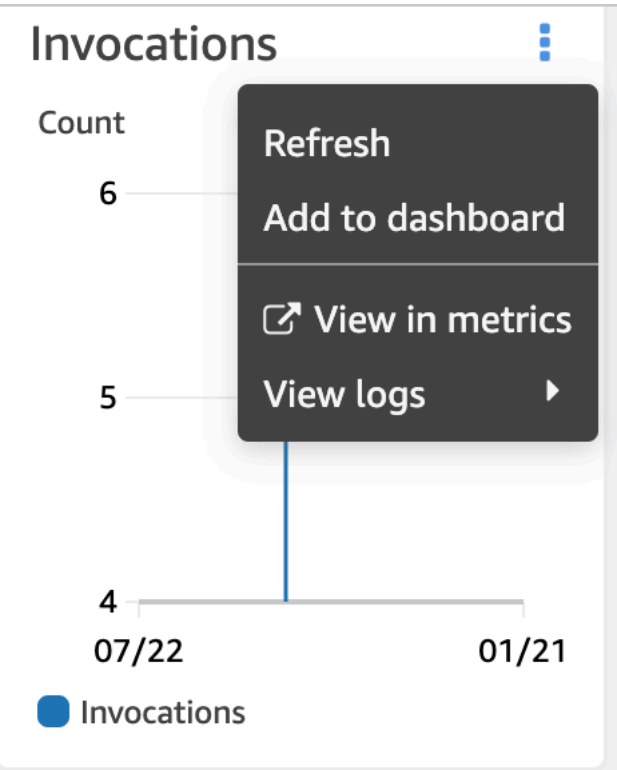
- IteratorAge— 스트림 이벤트 소스의 경우, Lambda가 항목을 수신하고 함수를 호출한 시점의 배치 마지막 항목의 수명.
- Async delivery failures(비동기 전송 실패) – Lambda가 대상 또는 배달 못한 편지 대기열에 쓰려고 할 때 발생한 오류의 개수입니다.
- Concurrent executions(동시 실행) – 이벤트를 처리 중인 함수 인스턴스의 개수입니다.

Lambda 콘솔에서 그래프 보기

다음 섹션에서는 Lambda 콘솔에서 CloudWatch 모니터링 그래프를 보고 지표 대시보드를 여는 CloudWatch 방법을 설명합니다.

함수에 대한 모니터링 그래프를 보는 방법

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 모니터링 탭을 선택합니다.
4. 미리 정의된 시간 범위 중에서 선택하거나 사용자 지정 시간 범위를 선택합니다.
5. 에서 CloudWatch 그래프의 정의를 보려면 세 개의 수직 점 (위젯 작업) 을 선택한 다음 지표에서 보기를 선택하여 콘솔에서 Metrics 대시보드를 엽니다. CloudWatch

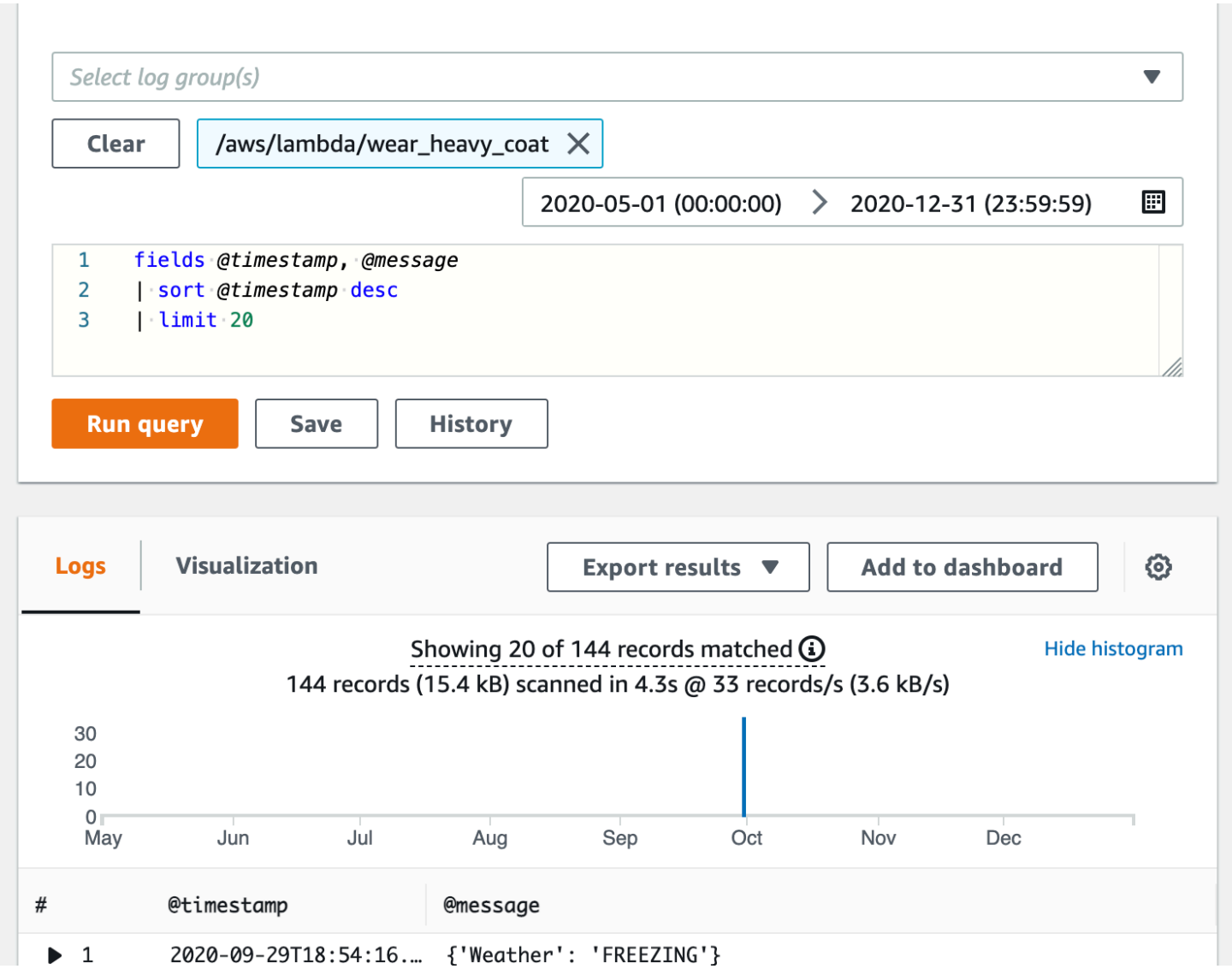


CloudWatch 로그 콘솔에서 쿼리 보기

다음 섹션에서는 CloudWatch Logs Insights의 보고서를 보고 CloudWatch Logs 콘솔의 사용자 지정 대시보드에 추가하는 방법을 설명합니다.

함수에 대한 보고서를 보는 방법

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 모니터링 탭을 선택합니다.
4. 로그인 보기를 선택합니다 CloudWatch.
5. [Logs Insights에서 보기(View in Logs Insights)]를 선택합니다.
6. 미리 정의된 시간 범위 중에서 선택하거나 사용자 지정 시간 범위를 선택합니다.
7. 쿼리 실행을 선택합니다.
8. (선택 사항) [저장(Save)]을 선택합니다.



다음 단계

- Lambda가 기록하고 전송하는 지표에 대해 알아보십시오. CloudWatch Lambda 함수 지표 작업
- CloudWatch Lambda Insights를 사용하여 Lambda 함수 런타임 성능 메트릭을 수집하고 집계하고 로그인하는 방법을 알아보십시오. Amazon CloudWatch Lambda Insights로 함수 성능 모니터링

Lambda 함수 지표 작업

AWS Lambda 함수가 이벤트 처리를 마치면 Lambda는 호출에 대한 지표를 Amazon CloudWatch로 전송합니다. 이러한 지표에 대해서는 요금이 부과되지 않습니다.

CloudWatch 콘솔에서 이러한 지표를 사용하여 그래프와 대시보드를 빌드할 수 있습니다. 경보를 설정하여 사용률, 성능 또는 오류율의 변화에 대응할 수 있습니다. Lambda는 1분 간격으로 지표 데이터를 CloudWatch로 보냅니다. Lambda 함수에 대한 보다 즉각적인 인사이트를 위해 Serverless Land의 설명에 따라 고분해능 사용자 지정 지표를 생성할 수 있습니다. 사용자 지정 지표 및 CloudWatch 경보에는 요금이 부과됩니다. 자세한 내용은 Amazon CloudWatch 요금을 참조하세요.

이 페이지에서는 CloudWatch 콘솔에서 사용할 수 있는 Lambda 함수 호출, 성능 및 동시성 지표를 설명합니다.

Sections

CloudWatch 콘솔에서 지표 보기

또한, CloudWatch 콘솔을 사용하여 함수 이름, 별칭 또는 버전별로 함수 지표를 필터링하고 정렬할 수 있습니다.

CloudWatch 콘솔에서 지표 보기

1. CloudWatch 콘솔의 지표 페이지(`AWS/Lambda` 네임스페이스)를 엽니다.
2. 찾아보기 탭의 지표에서 다음 차원 중 하나를 선택합니다.
   - 함수 이름별(`FunctionName`) – 함수의 모든 버전 및 별칭에 대한 집계 지표를 봅니다.
   - 리소스별(`Resource`) – 함수의 버전 또는 별칭에 대한 지표를 봅니다.
   - 실행된 버전별(`ExecutedVersion`) – 별칭과 버전의 조합에 대한 지표를 봅니다. `ExecutedVersion` 차원을 사용하여 가중치 기반 별칭의 두 대상인 함수의 두 버전에 대한 오류율을 비교합니다.
   - Across All Functions(전체 함수)(없음) – 현재 AWS 리전의 모든 함수에 대한 집계 지표를 봅니다.
3. 지표를 선택한 다음 A그래프에 추가 또는 다른 그래프 옵션을 선택합니다.

기본적으로 그래프는 모든 지표에 대해 Sum 통계를 사용합니다. 다른 통계를 선택하고 그래프를 사용자 지정하려면 그래프로 표시된 지표 탭의 옵션을 사용합니다.

> **Note**
>
> 지표의 타임스탬프는 함수가 호출된 시기를 반영합니다. 호출 기간에 따라 지표를 내보내기까지 몇 분이 걸릴 수 있습니다. 예를 들어, 함수에 10분의 제한 시간이 있는 경우 정확한 지표를 위해 이전의 10분 이상을 살펴봅니다.

CloudWatch에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서를 참조하세요.

지표의 유형

다음 섹션에서는 CloudWatch 콘솔에서 사용할 수 있는 Lambda 지표의 유형을 설명합니다.

호출 지표

호출 지표는 Lambda 함수 호출 결과의 이진 표시기입니다. 예를 들어, 함수가 오류를 반환하면 Lambda에서 값이 1인 `Errors` 지표를 보냅니다. 1분마다 발생한 함수 오류의 수를 확인하려면 1분의 기간에 대한 Sum 지표의 `Errors`를 확인합니다.

> **Note**
>
> Sum 통계로 다음 호출 지표를 확인합니다.

- `Invocations` – 성공적인 호출 및 함수 오류를 유발하는 호출을 포함하여 함수 코드가 호출된 횟수입니다. 호출 요청이 제한되거나 다른 방식으로 호출 오류가 발생하는 경우 호출이 기록되지 않습니다. `Invocations`의 값은 요금이 청구된 요청 수와 동일합니다.
- `Errors` – 함수 오류가 발생하는 호출 수. 함수 오류에는 코드에서 발생하는 예외와 Lambda 런타임에서 발생하는 예외가 포함됩니다. 런타임은 시간 초과 및 구성 오류와 같은 문제에 대한 오류를 반환합니다. 오류율을 계산하려면 `Errors`의 값을 `Invocations`의 값으로 나눕니다. 오류 지표의 타임스탬프는 오류가 발생한 시기가 아니라 함수가 호출된 시기를 반영합니다.
- `DeadLetterErrors` – 비동기 호출의 경우 Lambda에서 배달 못한 편지 대기열(DLQ)에 이벤트를 보내려고 시도하지만 실패하는 횟수입니다. 배달 못한 편지 오류는 잘못 구성된 리소스 또는 크기 제한으로 인해 발생할 수 있습니다.

- `DestinationDeliveryFailures` – 비동기 호출과 지원되는 이벤트 소스 매핑의 경우 Lambda에서 대상에 이벤트를 보내려고 시도하지만 실패하는 횟수입니다. 이벤트 소스 매핑의 경우 Lambda는 스트림 소스(DynamoDB 및 Kinesis)에 대한 대상을 지원합니다. 전송 오류는 권한 오류, 잘못 구성된 리소스 또는 크기 제한으로 인해 발생할 수 있습니다. 오류는 또한 구성한 대상이 Amazon SQS FIFO 대기열 또는 Amazon SNS FIFO 주제와 같이 지원되지 않는 유형인 경우 발생할 수 있습니다.
- `Throttles` – 제한된 호출 요청 수. 모든 함수 인스턴스가 요청을 처리 중이고 확장할 수 있는 동시성이 없는 경우 Lambda는 `TooManyRequestsException` 오류를 통해 추가 요청을 거부합니다. 제한된 요청 및 기타 호출 오류는 `Invocations` 또는 `Errors`로 간주되지 않습니다.
- `OversizedRecordCount` - Amazon DocumentDB 이벤트 소스의 경우 함수가 변경 스트림으로부터 수신하는 이벤트 중 크기가 6MB를 초과하는 이벤트의 수입니다. Lambda는 메시지를 삭제하고 이 지표를 내보냅니다.
- `ProvisionedConcurrencyInvocations` – 프로비저닝된 동시성을 사용하여 함수 코드가 호출되는 횟수입니다.
- `ProvisionedConcurrencySpilloverInvocations` – 모든 프로비저닝된 동시성을 사용 중인 경우, 표준 동시성을 사용하여 함수 코드가 호출되는 횟수입니다.
- `RecursiveInvocationsDropped` – 함수가 무한 재귀 루프의 일부인 것으로 감지되어 Lambda가 함수 간접 호출을 중단한 횟수입니다. Lambda 재귀 루프 감지를 사용하여 무한 루프 방지는 지원되는 AWS SDK에서 추가한 메타데이터를 추적하여 요청 체인의 일부로 함수가 간접적으로 호출되는 횟수를 모니터링합니다. 함수가 요청 체인의 일부로 16회 넘게 간접적으로 호출되면 Lambda는 다음 간접 호출을 삭제합니다.

성능 지표

성능 지표는 단일 함수 호출에 대한 성능 세부 정보를 제공합니다. 예를 들어, `Duration` 지표는 함수가 이벤트를 처리하는 데 소요되는 시간(밀리초)을 나타냅니다. 함수가 이벤트를 처리하는 속도를 확인하려면 `Average` 또는 `Max` 통계와 함께 이러한 지표를 살펴봅니다.

- `Duration` – 함수 코드가 이벤트를 처리하는 데 소요되는 시간. 요금이 청구되는 간접 호출 소요 시간은 가장 가까운 밀리초로 반올림된 `Duration` 값입니다. `Duration`에는 콜드 시작 시간이 포함되지 않습니다.
- `PostRuntimeExtensionsDuration` – 함수 코드가 완료된 후 런타임이 확장을 위해 코드를 실행하는 데 소비하는 누적 시간입니다.
- `IteratorAge` - DynamoDB, Kinesis 및 Amazon DocumentDB 이벤트 소스의 경우 이벤트의 마지막 레코드의 수명입니다. 이 지표는 경과 시간은 스트림이 레코드를 수신하는 시점과 이벤트 소스 매핑이 이벤트를 함수에 보내는 시점 사이의 시간을 측정합니다.

- `OffsetLag` - 자체 관리형 Apache Kafka 및 Amazon Managed Streaming for Apache Kafka(Amazon MSK) 이벤트 소스의 경우 주제에 작성된 마지막 레코드와 함수의 소비자 그룹이 처리한 마지막 레코드 간의 오프셋 차이입니다. Kafka 주제에는 여러 개의 파티션이 있을 수 있지만, 이 지표는 주제 수준에서 오프셋 지연을 측정합니다.

`Duration`는 백분위수 통계(p)도 지원합니다. `Average` 및 `Maximum` 통계를 왜곡하는 특이값을 제외하려면 백분위수를 사용합니다. 예를 들어, p95 통계는 가장 느린 5%를 제외하고 호출의 최대 지속 시간 95%를 보여줍니다. 자세한 내용은 Amazon CloudWatch 사용 설명서의 백분위수를 참조하세요.

동시성 지표

Lambda는 동시성 지표를 함수, 버전, 별칭 또는 AWS 리전에서 이벤트를 처리하는 인스턴스 수의 집계로 보고합니다. 동시성 제한에 얼마나 근접했는지 확인하려면 `Max` 통계와 함께 이러한 지표를 살펴봅니다.

- `ConcurrentExecutions` – 이벤트를 처리 중인 함수 인스턴스의 개수. 이 수가 리전의 동시 실행 할당량 또는 함수에 대해 구성한 예약된 동시성 제한에 도달하면 Lambda가 추가 호출 요청을 제한합니다.
- `ProvisionedConcurrentExecutions` – 프로비저닝된 동시성을 사용하여 이벤트를 처리 중인 함수 인스턴스의 개수. 프로비저닝된 동시성이 있는 별칭 또는 버전을 호출할 때마다 Lambda는 현재 개수를 내보냅니다.
- `ProvisionedConcurrencyUtilization` - 버전 또는 별칭의 경우 `ProvisionedConcurrentExecutions`의 값을 구성된 총 프로비저닝된 동시성의 양으로 나눈 값입니다. 예를 들어 함수에 대해 프로비저닝된 동시성을 10으로 구성하고 `ProvisionedConcurrentExecutions`가 7인 경우 `ProvisionedConcurrencyUtilization`은 0.7이 됩니다.
- `UnreservedConcurrentExecutions` – 리전의 경우 예약된 동시성이 없는 함수에서 처리 중인 이벤트의 수입니다.
- `ClaimedAccountConcurrency` - 리전에서 온디맨드 간접 호출에 사용할 수 없는 동시 실행 수입니다. `ClaimedAccountConcurrency`는 `UnreservedConcurrentExecutions`에 할당된 동시성 크기를 더한 값과 같습니다(즉, 예약된 총 동시성과 프로비저닝된 총 동시성 합계). 자세한 내용은 `ClaimedAccountConcurrency` 지표 작업 단원을 참조하십시오.

비동기 호출 지표

비동기 호출 지표는 이벤트 소스의 비동기 호출 및 직접 호출에 대한 세부 정보를 제공합니다. 임계값 및 경보를 설정하여 특정 변경 사항을 알릴 수 있습니다. 예를 들어, 처리를 위해 대기열에 있는 이벤트 수가 원치 않게 증가한 경우가 있습니다(AsyncEventsReceived). 또는 이벤트가 처리를 위해 장시간 동안 대기하고 있는 경우가 있습니다(AsyncEventAge).

- AsyncEventsReceived - Lambda가 처리를 위해 성공적으로 대기열에 추가한 이벤트 수. 이 지표는 Lambda 함수가 수신하는 이벤트 수에 대한 통찰력을 제공합니다. 이 지표를 모니터링하고 임계값에 대한 경보를 설정하여 문제를 확인합니다. 예를 들어, Lambda로 전송된 원치 않는 수의 이벤트를 감지하고, 잘못된 트리거 또는 함수 구성으로 인한 문제를 신속하게 진단합니다. AsyncEventsReceived 및 Invocations 사이의 불일치는 처리 불일치, 이벤트 삭제 또는 잠재적인 대기열 백로그를 의미할 수 있습니다.
- AsyncEventAge - Lambda가 이벤트를 성공적으로 대기열에 추가한 시점과 함수가 호출되는 시점 사이의 시간. 호출 실패 또는 병목 현상으로 인해 이벤트가 재시도될 때 이 지표의 값이 증가합니다. 이 지표를 모니터링하고 대기열 축적이 발생할 때 다양한 통계에 대한 임계값 관련 경보를 설정합니다. 이 지표의 증가 문제를 해결하려면 Errors 지표를 살펴보고 함수 오류를 식별한 다음 Throttles 지표를 살펴보고 동시성 문제를 식별합니다.
- AsyncEventsDropped - 함수를 성공적으로 실행하지 않고 삭제된 이벤트 수. DLQ(Dead Letter Queue) 또는 OnFailure 대상을 구성하는 경우 이벤트가 삭제되기 전에 해당 위치로 전송됩니다. 이벤트는 다양한 원인으로 삭제됩니다. 예를 들어, 이벤트가 최대 이벤트 기간을 초과하거나 최대 재시도 횟수를 소진하거나 예약된 동시성이 0으로 설정될 수 있습니다. 이벤트가 삭제되는 이유를 해결하려면 Errors 지표를 살펴보고 함수 오류를 식별한 다음 Throttles 지표를 살펴보고 동시성 문제를 식별합니다.

AWS Lambda과 함께 Amazon CloudWatch Logs 사용

AWS Lambda은 자동으로 Lambda 함수를 모니터링하여 함수의 장애 문제를 해결하는 데 도움이 됩니다. 함수의 실행 역할에 필요한 권한이 있는 한, Lambda는 함수가 처리하는 모든 요청에 대한 로그를 캡처하여 Amazon CloudWatch Logs로 전송합니다.

코드에 로깅 문을 삽입하여 코드가 예상대로 작동하는지 확인할 수 있습니다. Lambda는 CloudWatch Logs와 자동으로 통합되며 코드의 모든 로그를 Lambda 함수와 연결된 CloudWatch 로그 그룹에 전송합니다.

기본적으로 Lambda는 `/aws/lambda/<function name>`라는 로그 그룹에 로그를 전송합니다. 함수가 로그를 다른 그룹으로 전송하도록 하려면 Lambda 콘솔, AWS Command Line Interface(AWS CLI) 또는 Lambda API를 사용하여 이를 구성할 수 있습니다. 자세한 내용은 the section called “CloudWatch 로그 그룹 구성” 섹션을 참조하세요.

Lambda 함수의 로그는 Lambda 콘솔, CloudWatch 콘솔, AWS Command Line Interface(AWS CLI) 또는 CloudWatch API.를 사용해 확인할 수 있습니다.

> **Note**
>
> 함수 호출 후 로그가 표시되는 데 5~10분 정도 걸릴 수 있습니다.

섹션

필수 조건

실행 역할에는 CloudWatch Logs에 로그를 업로드할 수 있는 권한이 필요합니다. Lambda에서 제공하는 `AWSLambdaBasicExecutionRole` AWS 관리형 정책을 사용하여 CloudWatch Logs 권한을 추가할 수 있습니다. 이 정책을 역할에 추가하려면 다음 명령을 실행합니다.

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

자세한 내용은 the section called “AWS 관리형 정책” 단원을 참조하십시오.

요금

Lambda 로그 사용에 대한 추가 비용은 없지만 표준 CloudWatch Logs 비용이 적용됩니다. 자세한 내용은 CloudWatch 요금을 참조하세요.

Lambda 함수에 대한 고급 로깅 제어 구성

함수의 로그를 캡처, 처리 및 사용하는 방법을 더 잘 제어할 수 있도록 Lambda는 다음과 같은 로깅 구성 옵션을 제공합니다.

- 로그 형식 - 함수 로그의 경우 일반 텍스트와 구조화된 JSON 형식 중에서 선택
- 로그 수준 - JSON 형식의 로그의 경우, Lambda가 CloudWatch로 전송하는 로그의 세부 수준(ERROR, DEBUG 또는 INFO 등)을 선택
- 로그 그룹 - 함수가 로그를 보내는 CloudWatch 로그 그룹을 선택

JSON 및 일반 텍스트 로그 형식 구성

로그 출력을 JSON 키 값 쌍으로 캡처하면 함수를 디버깅할 때 더 쉽게 검색하고 필터링할 수 있습니다. JSON 형식의 로그를 사용하면 로그에 태그와 컨텍스트 정보를 추가할 수도 있습니다. 이를 통해 대량의 로그 데이터를 자동으로 분석할 수 있습니다. 개발 워크플로가 Lambda 로그를 일반 텍스트로 사용하는 기존 도구를 사용하지 않는 한, 로그 형식으로 JSON을 선택하는 것이 좋습니다.

모든 Lambda 관리형 런타임의 경우, 함수의 시스템 로그를 구조화되지 않은 일반 텍스트와 JSON 형식 중 어떻게 CloudWatch Logs에 보낼지 선택할 수 있습니다. 시스템 로그는 Lambda가 생성하는 로그이며 플랫폼 이벤트 로그라고도 합니다.

지원되는 런타임의 경우, 지원되는 내장 로깅 방법 중 하나를 사용하면 Lambda는 함수의 애플리케이션 로그(함수 코드가 생성하는 로그)를 구조화된 JSON 형식으로 출력할 수도 있습니다. 이러한 런타임에 대해 함수의 로그 형식을 구성할 때 선택한 구성이 시스템 및 애플리케이션 로그 모두에 적용됩니다.

지원되는 런타임의 경우 함수가 지원되는 로깅 라이브러리 또는 메서드를 사용하는 경우 구조화된 JSON으로 로그를 캡처하기 위해 Lambda의 기존 코드를 변경할 필요가 없습니다.

> **Note**
>
> JSON 로그 형식을 사용하면 메타데이터가 추가되고 로그 메시지가 일련의 키 값 쌍을 포함하는 JSON 객체로 인코딩됩니다. 이로 인해 함수의 로그 메시지 크기가 커질 수 있습니다.

지원되는 런타임 및 로깅 메서드

Lambda는 현재 다음 런타임에 대해 JSON 구조화된 애플리케이션 로그를 출력하는 옵션을 지원합니다.

런타임	지원되는 버전
Java	Amazon Linux 1에서 Java 8을 제외한 모든 Java 런타임
Node.js	Node.js 16 이상
Python	Python 3.7 이상

Lambda가 함수의 애플리케이션 로그를 구조화된 JSON 형식으로 CloudWatch에 보내려면 함수가 다음과 같은 내장 로깅 도구를 사용하여 로그를 출력해야 합니다.

- Java - `LambdaLogger` 로거 또는 Log4j2
- Node.js - 콘솔 메서드`console.trace`, `console.debug`, `console.log`, `console.info`, `console.error` 및 `console.warn`
- Python - 표준 Python `logging` 라이브러리

지원되는 런타임으로 고급 로깅 컨트롤을 사용하는 방법에 대한 자세한 내용은 the section called “로깅”, the section called “로깅” 및 the section called “로깅”을 참조하십시오.

다른 관리형 Lambda 런타임의 경우, Lambda는 현재 구조화된 JSON 형식의 시스템 로그 캡처만 기본적으로 지원합니다. 하지만 JSON 형식의 로그 출력 출력에 대해 AWS Lambda용 Powertools와 같은 로깅 도구를 사용하면 모든 런타임에서 구조화된 JSON 형식으로 애플리케이션 로그를 캡처할 수 있습니다.

기본 로그 형식

현재 모든 Lambda 런타임의 기본 로그 형식은 일반 텍스트입니다.

이미 AWS Lambda용 Powertools과 같은 로깅 라이브러리를 사용하여 함수 로그를 JSON 구조 형식으로 생성하고 있다면 JSON 로그 형식 지정을 선택하면 코드를 변경할 필요가 없습니다. Lambda는 이미 JSON으로 인코딩된 로그를 이중 인코딩하지 않으므로 함수의 애플리케이션 로그는 이전과 같이 계속 캡처됩니다.

시스템 로그의 JSON 형식

함수의 로그 형식을 JSON으로 구성하면 각 시스템 로그 항목(플랫폼 이벤트)이 다음 키가 있는 키 값 쌍을 포함하는 JSON 객체로 캡처됩니다.

- `"time"` - 로그 메시지가 생성된 시간
- `"type"` - 로그되는 이벤트 유형
- `"record"` - 로그 출력 내용

`"record"` 값 형식은 로그되는 이벤트 유형에 따라 달라집니다. 자세한 정보는 the section called “텔레메트리 API Event 객체 유형” 섹션을 참조하세요. 시스템 로그 이벤트에 할당된 로그 수준에 대한 자세한 내용은 the section called “시스템 로그 수준 이벤트 매핑”을 참조하세요.

비교를 위해 다음 두 예제는 일반 텍스트 형식과 구조화된 JSON 형식 모두에서 동일한 로그 출력을 보여줍니다. 대부분의 경우 시스템 로그 이벤트는 JSON 형식으로 출력할 때 일반 텍스트로 출력할 때보다 더 많은 정보를 포함합니다.

Example 일반 텍스트

```
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 구조화된 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 “텔레메트리 API”은 항상 JSON 형식으로 START 및 REPORT 등의 플랫폼 이벤트를 방출합니다. Lambda가 CloudWatch로 보내는 시스템 로그의 형식을 구성해도 Lambda 텔레메트리 API 동작에는 영향을 주지 않습니다.

애플리케이션 로그의 JSON 형식

함수의 로그 형식을 JSON으로 구성하면 지원되는 로깅 라이브러리 및 메서드를 사용하여 작성된 애플리케이션 로그 출력이 다음 키와 함께 키 값 페어를 포함하는 JSON 객체로 캡처됩니다.

- `"timestamp"` - 로그 메시지가 생성된 시간
- `"level"` - 메시지에 할당된 로그 수준
- `"message"` - 로그 메시지의 내용
- `"requestId"`(Python 및 Node.js) 또는 `"AWSrequestId"`(Java) - 함수 간접 호출을 위한 고유한 요청 ID

함수에서 사용하는 런타임 및 로깅 방법에 따라 이 JSON 객체에 추가 키 페어가 포함될 수도 있습니다. 예를 들어 Node.js에서 함수가 `console` 메서드를 통해 여러 인수를 사용하여 오류 객체를 기록하는 경우 JSON 객체에는 키 `errorMessage`, `errorType`, `stackTrace`와 함께 추가 키 값 페어가 포함됩니다. 다양한 Lambda 런타임에서 JSON 형식 로그에 대한 자세한 내용은 the section called “로깅”, the section called “로깅”, the section called “로깅” 섹션을 참조하세요.

> **Note**
>
> Lambda가 타임스탬프 값에 사용하는 키는 시스템 로그와 애플리케이션 로그마다 다릅니다. 시스템 로그의 경우 Lambda는 `"time"` 키를 사용하여 텔레메트리 API와의 일관성을 유지합니다. 애플리케이션 로그의 경우 Lambda는 지원되는 런타임 의 규칙을 따르며 `"timestamp"`를 사용합니다.

비교를 위해 다음 두 예제는 일반 텍스트 형식과 구조화된 JSON 형식 모두에서 동일한 로그 출력을 보여줍니다.

Example 일반 텍스트

```
2023-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example 구조화된 JSON

```
{
    "timestamp":"2023-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

함수의 로그 형식 설정

함수의 로그 형식을 구성하려면 Lambda 콘솔 또는 AWS Command Line Interface(AWS CLI)을 사용할 수 있습니다. CreateFunction 및 UpdateFunctionConfiguration Lambda API 명령, AWS Serverless Application Model(AWS SAM) AWS::Serverless::Function 리소스, AWS CloudFormation AWS::Lambda::Function 리소스를 사용하여 함수의 로그 형식을 구성할 수도 있습니다.

함수의 로그 형식을 변경해도 CloudWatch Logs에 저장된 기존 로그에는 영향을 주지 않습니다. 새 로그만 업데이트된 형식을 사용합니다.

함수의 로그 형식을 JSON으로 변경하고 로그 수준을 설정하지 않으면 Lambda는 함수의 애플리케이션 로그 수준과 시스템 로그 수준을 정보로 자동 설정합니다. 즉, Lambda는 정보 수준 이하의 로그 출력만 CloudWatch Logs로 전송합니다. 애플리케이션 및 시스템 로그 수준 필터링에 대한 자세한 내용은 the section called “로그 수준 필터링” 섹션을 참조하세요.

> **Note**
>
> Python 런타임의 경우 함수의 로그 형식이 일반 텍스트로 설정된 경우 기본 로그 수준 설정은 경고입니다. 즉, Lambda는 경고 수준 이하의 로그 출력만 CloudWatch Logs로 전송합니다. 함수의 로그 형식을 JSON으로 변경하면 이 기본 동작이 변경됩니다. Python에서 로깅에 대해 자세한 내용은 the section called “로깅” 섹션을 참조하세요.

임베디드 지표 형식 (EMF) 로그를 내보내는 Node.js 함수의 경우 함수의 로그 형식을 JSON으로 변경하면 CloudWatch에서 지표를 인식하지 못할 수 있습니다.

> **Important**
>
> 함수가 AWS Lambda용 Powertools(TypeScript) 또는 오픈 소스 EMF 클라이언트 라이브러리를 사용하여 EMF 로그를 방출하는 경우 CloudWatch가 계속해서 로그를 올바르게 구문분석할 수 있도록 Powertools 및 EMF 라이브러리를 최신 버전으로 업데이트합니다. JSON 로그 형식으로 전환하는 경우 함수에 내장된 지표와의 호환성을 확인하기 위한 테스트도 수행하는 것이 좋습니다. EMF 로그를 내보내는 node.js 함수에 대한 자세한 내용은 the section called “구조화된 JSON 로그가 포함된 임베디드 메트릭 형식(EMF) 클라이언트 라이브러리 사용”을 참조하십시오.

함수의 로그 형식 구성하기(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수 선택
3. 함수 구성 페이지에서 모니터링 및 작업 도구를 선택합니다.
4. 로깅 구성 창에서 편집을 선택합니다.
5. 로그 콘텐츠에서 로그 형식에 대해 텍스트 또는 JSON을 선택합니다.
6. Save(저장)를 선택합니다.

기존 함수의 로그 형식 변경하기(AWS CLI)

- 기존 함수의 로그 형식을 변경하려면 `update-function-configuration` 명령을 사용합니다. `LoggingConfig`에서 `LogFormat` 옵션을 `JSON` 또는 `Text`로 설정합니다.

```
aws lambda update-function-configuration \
--function-name myFunction --logging-config LogFormat=JSON
```

함수를 생성할 때 로그 형식 설정하기(AWS CLI)

- 새 함수를 만들 때 로그 형식을 구성하려면 `--logging-config` 옵션을 `create-function` 명령에서 사용합니다. `LogFormat`을 `JSON` 또는 `Text`로 설정합니다. 다음 예제 명령은 Node.js 18 런타임을 사용하여 로그를 구조화된 JSON으로 출력하는 함수를 만듭니다.

함수를 생성할 때 로그 형식을 지정하지 않으면 Lambda는 선택한 런타임 버전의 기본 로그 형식을 사용합니다. 기본 로깅 형식에 대한 자세한 내용은 the section called “기본 로그 형식”을 참조하세요.

```
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
```

로그 수준 필터링

Lambda는 특정 세부 수준 이하의 로그만 CloudWatch Logs로 전송되도록 함수의 로그를 필터링할 수 있습니다. 함수의 시스템 로그(Lambda가 생성하는 로그)와 애플리케이션 로그(함수 코드가 생성하는 로그)에 대해 개별적으로 로그 수준 필터링을 구성할 수 있습니다.

the section called “지원되는 런타임 및 로깅 메서드”의 경우 함수의 애플리케이션 로그를 필터링하기 위해 Lambda의 함수 코드를 변경할 필요가 없습니다.

다른 모든 런타임 및 로깅 메서드의 경우 함수 코드는 키 `"level"`과 키 값 쌍을 포함하는 JSON 형식의 객체로 로그 이벤트를 `stdout` 또는 `stderr`에 출력해야 합니다. 예를 들어, Lambda는 다음 `stdout`로의 출력을 DEBUG 수준 로그로 해석합니다.

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2023-11-02T16:51:31.587199Z"}')
```

`"level"` 값 필드가 유효하지 않거나 누락된 경우 Lambda는 로그 출력에 수준 INFO를 할당합니다. Lambda가 타임스탬프 필드를 사용하려면 유효한 RFC 3339 타임스탬프 형식으로 시간을 지정해야 합니다. 유효한 타임스탬프를 제공하지 않으면 Lambda는 로그에 레벨 INFO를 할당하고 타임스탬프를 추가합니다.

타임스탬프 키의 이름을 지정할 때는 사용 중인 런타임의 규칙을 따르십시오. Lambda는 관리형 런타임에서 사용되는 대부분의 일반적인 명명 규칙을 지원합니다. 예를 들어, .NET 런타임을 사용하는 함수에서 Lambda는 `"Timestamp"` 키를 인식합니다.

> **Note**
>
> 로그 수준 필터링을 사용하려면 함수가 JSON 로그 형식을 사용하도록 구성해야 합니다. 현재 모든 Lambda 관리형 런타임의 기본 로그 형식은 일반 텍스트입니다. 함수의 로그 형식을

JSON으로 구성하는 방법을 알아보려면 the section called “함수의 로그 형식 설정”을 참조하세요.

애플리케이션 로그(함수 코드로 생성된 로그)의 경우 다음 로그 수준 중에서 선택할 수 있습니다.

로그 수준	표준 사용량
TRACE(최대 세부 정보)	코드 실행 경로를 추적하는 데 사용되는 가장 세밀한 정보
DEBUG	시스템 디버깅에 대한 세부 정보
INFO	함수의 정상 작동을 기록하는 메시지
WARN	해결되지 않을 경우 예상치 못한 동작으로 이어질 수 있는 잠재적 오류에 대한 메시지
ERROR	코드가 예상대로 작동하지 못하게 하는 문제에 대한 메시지
FATAL(최소 세부 정보)	응용 프로그램 작동을 중지시키는 심각한 오류에 대한 메시지

로그 수준을 선택하면 Lambda는 해당 수준 이하의 로그를 CloudWatch Logs로 보냅니다. 예를 들어 함수의 애플리케이션 로그 수준을 WARN으로 설정하면 Lambda는 INFO 및 DEBUG 수준에서 로그 출력을 전송하지 않습니다. 로그 필터링의 기본 애플리케이션 로그 수준은 INFO입니다.

Lambda가 함수의 애플리케이션 로그를 필터링할 때 레벨이 없는 로그 메시지에는 로그 수준 INFO가 할당됩니다.

시스템 로그(Lambda 서비스에서 생성된 로그)의 경우 다음 로그 수준 중에서 선택할 수 있습니다.

로그 수준	사용량
DEBUG(최대 세부 정보)	시스템 디버깅에 대한 세부 정보
INFO	함수의 정상 작동을 기록하는 메시지

로그 수준	사용량
WARN(최소 세부 정보)	해결되지 않을 경우 예상치 못한 동작으로 이어질 수 있는 잠재적 오류에 대한 메시지

로그 수준을 선택하면 Lambda는 해당 수준 이하의 로그를 전송합니다. 예를 들어 함수의 시스템 로그 수준을 INFO로 설정하면 Lambda는 DEBUG 수준에서 로그 출력을 전송하지 않습니다.

기본적으로 Lambda는 시스템 로그 레벨을 INFO로 설정합니다. 이 설정을 사용하면 Lambda는 자동으로 `"start"` 및 `"report"` 로그 메시지를 CloudWatch에 전송합니다. 더 많거나 덜 상세한 시스템 로그를 수신하려면 로그 수준을 DEBUG 또는 WARN으로 변경합니다. Lambda가 다양한 시스템 로그 이벤트를 매핑하는 로그 수준 목록을 보려면 the section called “시스템 로그 수준 이벤트 매핑”을 참조하세요.

로그 수준 필터링 구성

함수에 대한 애플리케이션 및 시스템 로그 수준 필터링을 구성하려면 AWS Command Line Interface(AWS CLI)에서 Lambda 콘솔을 사용할 수 있습니다. CreateFunction 및 UpdateFunctionConfiguration Lambda API 명령, AWS Serverless Application Model(AWS SAM) AWS::Serverless::Function 리소스, AWS CloudFormation AWS::Lambda::Function 리소스를 사용하여 함수의 로그 수준을 구성할 수도 있습니다.

코드에서 함수의 로그 수준을 설정하는 경우 이 설정은 구성된 다른 로그 수준 설정보다 우선합니다. 예를 들어 Python `logging` `setLevel()` 메서드를 사용하여 함수의 로깅 수준을 정보로 설정하는 경우 이 설정은 Lambda 콘솔을 사용하여 구성한 경고 설정보다 우선합니다.

기존 함수의 애플리케이션 또는 시스템 로그 수준 구성하기(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 구성 페이지에서 모니터링 및 작업 도구를 선택합니다.
4. 로깅 구성 창에서 편집을 선택합니다.
5. 로그 콘텐츠에서 로그 형식에 대해 JSON이 선택되어 있는지 확인합니다.
6. 라디오 버튼을 사용하여 함수에 대해 원하는 애플리케이션 로그 수준 및 시스템 로그 수준을 선택합니다.
7. Save(저장)를 선택합니다.

기존 함수의 애플리케이션 또는 시스템 로그 수준 구성하기(AWS CLI)

- 기존 함수의 애플리케이션 또는 시스템 로그 수준을 변경하려면 `update-function-configuration` 명령을 사용합니다. `--system-log-level`을 `DEBUG`, `INFO` 또는 `WARN`중 하나로 설정합니다. `--application-log-level`을 `DEBUG`, `INFO` `WARN`, `ERROR` 또는 `FATAL` 중 하나로 설정합니다.

```
aws lambda update-function-configuration \
--function-name myFunction --system-log-level WARN \
--application-log-level ERROR
```

함수를 생성할 때 로그 수준 필터링 구성하기

- 새 함수를 만들 때 로그 수준 필터링을 구성하려면 create-function 명령의 `--system-log-level` 및 `--application-log-level` 옵션을 사용합니다. `--system-log-level`을 `DEBUG`, `INFO` 또는 `WARN`중 하나로 설정합니다. `--application-log-level`을 `DEBUG`, `INFO` `WARN`, `WARN` 또는 `FATAL` 중 하나로 설정합니다.

```
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
```

시스템 로그 수준 이벤트 매핑

Lambda에서 생성된 시스템 수준 로그 이벤트의 경우 다음 표는 각 이벤트에 할당된 로그 수준을 정의합니다. 표에 나열된 이벤트에 대한 자세한 내용은 the section called “Event 스키마 참조”를 참조하세요.

이벤트 이름	Condition	지정된 로그 수준
initStart	runtimeVersion is set	INFO
initStart	runtimeVersion is not set	DEBUG
initRuntimeDone	status=success	DEBUG
initRuntimeDone	status!=success	WARN

이벤트 이름	Condition	지정된 로그 수준
initReport	initializationType=snapstart	INFO
initReport	initializationType!=snapstart	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion is set	INFO
restoreStart	runtimeVersion is not set	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
시작	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
report	status=success	INFO
report	status!=success	WARN
extension	state=success	INFO
extension	state!=success	WARN
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> the section called “텔레메트리 API”은 항상 전체 플랫폼 이벤트 세트를 방출합니다. Lambda가 CloudWatch로 보내는 시스템 로그의 수준을 구성해도 Lambda 텔레메트리 API 동작에는 영향을 미치지 않습니다.

사용자 지정 런타임을 사용한 애플리케이션 로그 수준 필터링

함수에 대한 애플리케이션 로그 레벨 필터링을 구성하면 Lambda는 `AWS_LAMBDA_LOG_LEVEL` 환경 변수를 사용하여 백그라운드에서 애플리케이션 로그 레벨을 설정합니다. 또한 Lambda는 `AWS_LAMBDA_LOG_FORMAT` 환경 변수를 사용하여 함수의 로그 형식을 설정합니다. 이러한 변수를 사용하여 Lambda 고급 로깅 제어를 사용자 지정 런타임에 통합할 수 있습니다.

Lambda 콘솔, AWS CLI 및 Lambda API로 사용자 지정 런타임을 사용하여 함수에 대한 로깅 설정을 구성하려면 이러한 환경 변수의 값을 확인하도록 사용자 지정 런타임을 구성합니다. 그런 다음 선택한 로그 형식 및 로그 수준에 따라 런타임의 로거를 구성할 수 있습니다.

CloudWatch 로그 그룹 구성

기본적으로 CloudWatch는 함수를 처음 간접 호출할 때 함수의 `/aws/lambda/<function name>`이라는 이름이 지정된 로그 그룹을 자동으로 생성합니다. 기존 로그 그룹에 로그를 전송하도록 함수를 구성하거나 함수에 대한 새 로그 그룹을 생성하려면 Lambda 콘솔 또는 AWS CLI를 사용할 수 있습니다. CreateFunction, UpdateFunctionConfiguration Lambda API 명령 및 AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function 리소스를 사용하여 사용자 지정 로그 그룹을 구성할 수도 있습니다.

로그를 동일한 CloudWatch 로그 그룹에 전송하도록 여러 Lambda 함수를 구성할 수 있습니다. 예를 들어 단일 로그 그룹을 사용하여 특정 애플리케이션을 구성하는 모든 Lambda 함수에 대한 로그를 저장할 수 있습니다. Lambda 함수에 대한 사용자 지정 로그 그룹을 사용하는 경우 Lambda가 생성하는 로그 스트림에는 함수 이름과 함수 버전이 포함됩니다. 이렇게 하면 여러 함수에 동일한 로그 그룹을 사용하더라도 로그 메시지와 함수 간의 매핑이 보존됩니다.

사용자 지정 로그 그룹의 로그 스트림 이름 지정 형식은 다음 규칙을 따릅니다.

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

사용자 지정 로그 그룹을 구성할 때 로그 그룹에 대해 선택한 이름은 CloudWatch 로그 이름 지정 규칙을 따라야 한다는 점에 유의합니다. 또한 사용자 지정 로그 그룹 이름은 `aws/` 문자열로 시작해서는

안 됩니다. aws/로 시작하는 사용자 지정 로그 그룹을 생성하는 경우 Lambda는 로그 그룹을 생성할 수 없습니다. 따라서 함수의 로그가 CloudWatch로 전송되지 않습니다.

함수의 로그 그룹 변경하기(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 함수 구성 페이지에서 모니터링 및 작업 도구를 선택합니다.
4. 로깅 구성 창에서 편집을 선택합니다.
5. 로깅 그룹 창의 CloudWatch 로그 그룹에서 사용자 지정을 선택합니다.
6. 사용자 지정 로그 그룹에서 함수가 로그를 전송하기를 원하는 CloudWatch 로그 그룹의 이름을 입력합니다. 기존 로그 그룹의 이름을 입력하면 함수가 해당 그룹을 사용합니다. 입력한 이름을 가진 로그 그룹이 없는 경우 Lambda는 해당 이름으로 함수에 대한 새 로그 그룹을 생성합니다.

함수의 로그 그룹 변경하기(AWS CLI)

- 기존 함수의 로그 그룹을 변경하려면 update-function-configuration 명령을 사용합니다. 기존 로그 그룹의 이름을 지정하면 함수에서 해당 그룹을 사용합니다. 지정한 이름을 가진 로그 그룹이 없는 경우 Lambda는 해당 이름으로 함수에 대한 새 로그 그룹을 생성합니다.

```
aws lambda update-function-configuration \
--function-name myFunction --log-group myLogGroup
```

함수를 생성할 때 사용자 지정 로그 그룹 지정하기(AWS CLI)

- AWS CLI를 사용하여 새 Lambda 함수를 생성할 때 사용자 지정 로그 그룹을 지정하려면 --log-group 옵션을 사용하십시오. 기존 로그 그룹의 이름을 지정하면 함수에서 해당 그룹을 사용합니다. 지정한 이름을 가진 로그 그룹이 없는 경우 Lambda는 해당 이름으로 함수에 대한 새 로그 그룹을 생성합니다.

  다음 예제 명령은 myLogGroup이라는 로그 그룹에 로그를 보내는 Node.js Lambda 함수를 생성합니다.

```
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
```

실행 역할 권한

함수가 로그를 CloudWatch Logs로 전송하려면 해당 함수에 `logs:PutLogEvents` 권한이 있어야 합니다. Lambda 콘솔을 사용하여 함수의 로그 그룹을 구성할 때 함수에 이 권한이 없는 경우 Lambda는 기본적으로 이를 함수의 실행 역할에 추가합니다. Lambda가 이 권한을 추가하면 모든 CloudWatch Logs 로그 그룹에 로그를 전송할 수 있는 권한을 함수에 부여합니다.

Lambda가 함수의 실행 역할을 자동으로 업데이트하지 않도록 하고 대신 수동으로 편집하려면 권한을 확장하고 필요한 권한 추가를 선택 취소합니다.

AWS CLI를 사용하여 함수의 로그 그룹을 구성하면 Lambda는 `logs:PutLogEvents` 권한을 자동으로 추가하지 않습니다. 함수의 실행 역할에 권한을 추가합니다(아직 권한이 없는 경우). 이러한 권한은 AWSLambdaBasicExecutionRole 관리형 정책에 포함됩니다.

Lambda 콘솔을 사용하여 로그에 액세스

Lambda 콘솔을 사용하여 로그를 보려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 모니터(Monitor)를 선택합니다.
4. CloudWatch에서 로그 보기를 선택합니다.

AWS CLI를 사용하여 로그에 액세스

AWS CLI은(는) 명령줄 셸의 명령을 사용하여 AWS 서비스와 상호 작용할 수 있는 오픈 소스 도구입니다. 이 섹션의 단계를 완료하려면 다음이 필요합니다.

- AWS Command Line Interface(AWS CLI) 버전 2
- AWS CLI - `aws configure`을 통한 빠른 구성

AWS CLI를 사용하면 `--log-type` 명령 옵션을 통해 호출에 대한 로그를 검색할 수 있습니다. 호출에서 base64로 인코딩된 로그를 최대 4KB까지 포함하는 `LogResult` 필드가 응답에 포함됩니다.

Example 로그 ID 검색

다음 예제에서는 `LogResult`이라는 함수의 `my-function` 필드에서 로그 ID를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

다음 결과가 표시됩니다:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

동일한 명령 프롬프트에서 base64 유틸리티를 사용하여 로그를 디코딩합니다. 다음 예제에서는 my-function에 대한 base64로 인코딩된 로그를 검색하는 방법을 보여줍니다.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

다음 결과가 표시됩니다.

```
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 유틸리티는 Linux, macOS 및 Ubuntu on Windows에서 사용할 수 있습니다. macOS 사용자는 base64 -D를 사용해야 할 수도 있습니다.

Example get-logs.sh 스크립트

동일한 명령 프롬프트에서 다음 스크립트를 사용하여 마지막 5개 로그 이벤트를 다운로드합니다. 이 스크립트는 sed를 사용하여 출력 파일에서 따옴표를 제거하고, 로그를 사용할 수 있는 시간을 허용하기 위해 15초 동안 대기합니다. 출력에는 Lambda의 응답과 get-log-events 명령의 출력이 포함됩니다.

다음 코드 샘플의 내용을 복사하고 Lambda 프로젝트 디렉터리에 get-logs.sh로 저장합니다.

cli-binary-format 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 이 설정을 기본 설정으로 지정하려면 aws configure set cli-binary-format raw-in-base64-out을(를) 실행하세요. 자세한 내용은 AWS CLI 지원되는 글로벌 명령줄 옵션을 AWS Command Line Interface 사용 설명서 버전 2에서 참조하세요.

```
#!/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 및 Linux(전용)

동일한 명령 프롬프트에서 macOS 및 Linux 사용자는 스크립트가 실행 가능한지 확인하기 위해 다음 명령을 실행해야 할 수 있습니다.

```
chmod -R 755 get-logs.sh
```

Example 마지막 5개 로그 이벤트 검색

동일한 명령 프롬프트에서 다음 스크립트를 실행하여 마지막 5개 로그 이벤트를 가져옵니다.

```
./get-logs.sh
```

다음 결과가 표시됩니다.

```
{
    "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"
}
```


런타임 함수 로깅

코드를 디버깅하고 예상대로 작동하는지 확인하기 위해 프로그래밍 언어에 대한 표준 로깅 기능을 사용하여 로그를 출력할 수 있습니다. Lambda 런타임은 함수의 로그 출력을 CloudWatch Logs에 업로드합니다. 언어별 지침은 다음 주제를 참조하세요.

- AWS Lambda 함수 로깅(Node.js)
- AWS Lambda 함수 로깅(Python)
- AWS Lambda 함수 로깅(Ruby)
- AWS Lambda 함수 로깅(Java)
- AWS Lambda 함수 로깅(Go)
- Lambda 함수 로깅(C#)
- AWS Lambda 함수 로깅(PowerShell)

다음 단계

- 로그 그룹 및 CloudWatch 콘솔을 통해 해당 그룹에 액세스하는 방법에 대한 자세한 내용은 Amazon CloudWatch 사용 설명서의 시스템, 애플리케이션 및 사용자 지정 로그 파일 모니터링을 참조하세요.

AWS CloudTrail을 사용하여 AWS Lambda API 호출 로깅

AWS Lambda는 사용자, 역할 또는 AWS 서비스가 수행한 작업의 레코드를 제공하는 서비스인 AWS CloudTrail과 통합됩니다. CloudTrail은 Lambda에 대한 API 호출을 이벤트로 캡처합니다. 캡처되는 호출에는 Lambda 콘솔에서의 호출과 Lambda API 작업에 대한 코드 호출이 포함됩니다. CloudTrail에서 수집한 정보를 사용하여 Lambda에 수행된 요청, 요청이 수행된 IP 주소, 요청이 수행된 시간, 추가 세부 정보를 확인할 수 있습니다.

모든 이벤트 및 로그 항목에는 요청을 생성한 사용자에 대한 정보가 들어 있습니다. 신원 정보를 이용하면 다음을 쉽게 알아볼 수 있습니다.

- 요청을 루트 사용자로 했는지 사용자 보안 인증으로 했는지 여부.
- IAM Identity Center 사용자를 대신하여 요청이 이루어졌는지 여부입니다.
- 역할 또는 페더레이션 사용자에 대한 임시 보안 자격 증명을 사용하여 요청이 생성되었는지 여부.
- 다른 AWS 서비스에서 요청했는지 여부.

계정을 생성할 때 AWS 계정에서 CloudTrail이 활성 상태이며, CloudTrail 이벤트 기록에 자동으로 액세스할 수 있습니다. CloudTrail 이벤트 기록은 지난 90일 간 AWS 리전의 관리 이벤트에 대해 보기, 검색 및 다운로드가 가능하고, 수정이 불가능한 레코드를 제공합니다. 자세한 설명은 AWS CloudTrail 사용 설명서의 CloudTrail 이벤트 기록 작업을 참조하세요. Event history(이벤트 기록) 보기는 CloudTrail 요금이 부과되지 않습니다.

지난 90일 동안 AWS 계정에서 진행 중인 이벤트 기록을 보려면 추적 또는 CloudTrail Lake 이벤트 데이터 스토어를 생성합니다.

CloudTrail 추적

CloudTrail은 추적을 사용하여 Amazon S3 버킷으로 로그 파일을 전송할 수 있습니다. AWS Management Console을 사용하여 만든 추적은 모두 다중 리전입니다. AWS CLI를 사용하여 단일 리전 또는 다중 리전 추적을 생성할 수 있습니다. 계정의 모든 AWS 리전에서 활동을 캡처하므로, 다중 리전 추적 생성이 권장됩니다. 단일 리전 추적을 생성하는 경우 추적의 AWS 리전에 로깅된 이벤트만 볼 수 있습니다. 추적에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 Creating a trail for your AWS 계정 및 Creating a trail for an organization을 참조하세요.

CloudTrail에서 추적을 생성하여 진행 중인 관리 이벤트의 사본 하나를 Amazon S3 버킷으로 무료로 전송할 수는 있지만, Amazon S3 스토리지 요금이 부과됩니다. CloudTrail 요금에 대한 자세한 내용은 AWS CloudTrail 요금을 참조하세요. Amazon S3 요금에 대한 자세한 내용은 Amazon S3 요금을 참조하세요.

CloudTrail Lake 이벤트 데이터 스토어

CloudTrail Lake를 사용하면 이벤트에 대해 SQL 기반 쿼리를 실행할 수 있습니다. CloudTrail Lake는 행 기반 JSON 형식의 기존 이벤트를 Apache ORC 형식으로 변환합니다. ORC는 빠른 데이터 검색에 최적화된 열 기반 스토리지 형식입니다. 이벤트는 이벤트 데이터 스토어로 집계되며, 이벤트 데이터 스토어는 고급 이벤트 선택기를 적용하여 선택한 기준을 기반으로 하는 변경 불가능한 이벤트 컬렉션입니다. 이벤트 데이터 스토어에 적용하는 선택기는 어떤 이벤트가 지속되고 쿼리할 수 있는지 제어합니다. CloudTrail Lake에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 Working with AWS CloudTrail Lake를 참조하세요.

CloudTrail Lake 이벤트 데이터 스토어 및 쿼리에는 비용이 발생합니다. 이벤트 데이터 스토어를 생성할 때 이벤트 데이터 스토어에 사용할 요금 옵션을 선택합니다. 요금 옵션에 따라 이벤트 모으기 및 저장 비용과 이벤트 데이터 스토어의 기본 및 최대 보존 기간이 결정됩니다. CloudTrail 요금에 대한 자세한 내용은 AWS CloudTrail 요금을 참조하세요.

CloudTrail의 Lambda 데이터 이벤트

데이터 이벤트는 리소스 기반 또는 리소스에서 수행된 리소스 작업에 대한 정보를 제공합니다(예: Amazon S3 객체 읽기 또는 쓰기). 이를 데이터 영역 작업이라고도 합니다. 데이터 이벤트가 대량 활동인 경우도 있습니다. 기본적으로 CloudTrail은 대부분의 데이터 이벤트를 기록하지 않으며 CloudTrail 이벤트 기록에는 이러한 이벤트가 기록되지 않습니다.

지원되는 서비스에 대해 기본적으로 기록되는 CloudTrail 데이터 이벤트는 `LambdaESMDisabled`입니다. Lambda 이벤트 소스 매핑 관련 문제를 해결하기 위해 이 이벤트를 사용하는 방법을 자세히 알아보려면 the section called “CloudTrail을 사용하여 비활성화된 Lambda 이벤트 소스 문제 해결” 섹션을 참조하세요.

데이터 이벤트에는 추가 요금이 적용됩니다. CloudTrail 요금에 대한 자세한 내용은 AWS CloudTrail 요금을 참조하세요.

CloudTrail 콘솔, AWS CLI 또는 CloudTrail API 작업을 사용하여 `AWS::Lambda::Function` 리소스 유형에 대한 데이터 이벤트를 로깅할 수 있습니다. 데이터 이벤트를 로깅하는 방법에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 Logging data events with the AWS Management Console 및 Logging data events with the AWS Command Line Interface를 참조하세요.

다음 표에는 데이터 이벤트를 로깅할 수 있는 Lambda 리소스 유형이 나열되어 있습니다. 데이터 이벤트 유형(콘솔) 열에는 CloudTrail 콘솔의 데이터 이벤트 유형 목록에서 선택할 값이 표시됩니다. resources.type 값 열에는 AWS CLI 또는 CloudTrail API를 사용하여 고급 이벤트 선택기를 구성할 때

지정하는 `resources.type` 값이 표시됩니다. CloudTrail에 로깅되는 데이터 API 열에는 리소스 유형에 대해 CloudTrail에 로깅된 API 호출이 표시됩니다.

데이터 이벤트 유형(콘솔)	resources.type 값	CloudTrail에 로깅되는 데이터 API
Lambda	`AWS::Lambda::Function`	Invoke

`eventName`, `readOnly` 및 `resources.ARN` 필드를 필터링하여 중요한 이벤트만 로깅하도록 고급 이벤트 선택기를 구성할 수 있습니다. 다음 예제는 특정 함수에 대한 이벤트만 로깅하는 데이터 이벤트 구성의 JSON 뷰입니다. 이러한 필드에 대한 자세한 내용은 AWS CloudTrail API 참조의 AdvancedFieldSelector 섹션을 참조하세요.

```
[
  {
    "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"
        ]
      }
    ]
  }
]
```

CloudTrail의 Lambda 관리 이벤트

관리 이벤트는 AWS 계정의 리소스에 대해 수행되는 관리 작업에 대한 정보를 제공합니다. 이를 제어 영역 작업이라고도 합니다. 기본적으로 CloudTrail은 관리 이벤트를 로깅합니다.

Lambda는 CloudTrail 로그 파일에 다음 작업을 관리 이벤트로 로깅합니다.

> **Note**
>
> CloudTrail 로그 파일에서 `eventName`에 날짜 및 버전 정보가 포함될 수 있지만, 여전히 동일한 퍼블릭 API 작업을 참조합니다. 예를 들어 `GetFunction` 작업은 `GetFunction20150331v2`로 표시됩니다. 다음 목록은 이벤트 이름이 API 작업 이름과 다른 경우를 지정합니다.

- AddLayerVersionPermission
- AddPermission(이벤트 이름: `AddPermission20150331v2`)
- CreateAlias(이벤트 이름: `CreateAlias20150331`)
- CreateEventSourceMapping(이벤트 이름: `CreateEventSourceMapping20150331`)
- CreateFunction(이벤트 이름: `CreateFunction20150331`)

  (`Environment` 및 `ZipFile` 파라미터가 `CreateFunction`의 CloudTrail 로그에서 생략됨)
- CreateFunctionUrlConfig
- DeleteAlias(이벤트 이름: `DeleteAlias20150331`)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping(이벤트 이름: `DeleteEventSourceMapping20150331`)
- DeleteFunction(이벤트 이름: `DeleteFunction20150331`)
- DeleteFunctionConcurrency(이벤트 이름: `DeleteFunctionConcurrency20171031`)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias(이벤트 이름: `GetAlias20150331`)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig

- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion(이벤트 이름: PublishLayerVersion20181031)

  (ZipFile 파라미터가 PublishLayerVersion의 CloudTrail 로그에서 생략됨)
- PublishVersion(이벤트 이름: PublishVersion20150331)
- PutFunctionConcurrency(이벤트 이름: PutFunctionConcurrency20171031)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission(이벤트 이름: RemovePermission20150331v2)
- TagResource(이벤트 이름: TagResource20170331v2)
- UntagResource(이벤트 이름: UntagResource20170331v2)
- UpdateAlias(이벤트 이름: UpdateAlias20150331)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping(이벤트 이름: UpdateEventSourceMapping20150331)
- UpdateFunctionCode(이벤트 이름: UpdateFunctionCode20150331v2)

  (ZipFile 파라미터가 UpdateFunctionCode의 CloudTrail 로그에서 생략됨)
- UpdateFunctionConfiguration(이벤트 이름: UpdateFunctionConfiguration20150331v2)

  (Environment 파라미터가 UpdateFunctionConfiguration의 CloudTrail 로그에서 생략됨)
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

CloudTrail을 사용하여 비활성화된 Lambda 이벤트 소스 문제 해결

UpdateEventSourceMapping API 작업을 사용하여 이벤트 소스 매핑의 상태를 변경하는 경우 API 호출이 CloudTrail에 관리 이벤트로 기록됩니다. 오류 때문에 이벤트 소스 매핑이 `Disabled` 상태로 직접 전환될 수도 있습니다.

다음 서비스의 경우 Lambda는 이벤트 소스가 비활성화된 상태로 전환될 때 CloudTrail에 `LambdaESMDisabled` 데이터 이벤트를 게시합니다.

- Amazon Simple Queue Service(Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda는 다른 이벤트 소스 매핑 유형에 대해서는 이 이벤트를 지원하지 않습니다.

지원되는 서비스의 이벤트 소스 매핑이 `Disabled` 상태로 전환될 때 알림을 받으려면 `LambdaESMDisabled` CloudTrail 이벤트를 사용하여 Amazon CloudWatch에서 경보를 설정하세요. CloudWatch 경보 설정에 대한 자세한 내용은 CloudTrail 이벤트에 대한 CloudWatch 경보 생성: 예를 참조하세요.

`LambdaESMDisabled` 이벤트 메시지의 `serviceEventDetails` 엔티티에는 다음 오류 코드 중 하나가 포함되어 있습니다.

`RESOURCE_NOT_FOUND`

요청에 지정된 리소스가 없습니다.

`FUNCTION_NOT_FOUND`

이벤트 소스에 연결된 함수가 없습니다.

`REGION_NAME_NOT_VALID`

이벤트 소스 또는 함수에 제공된 리전 이름이 잘못되었습니다.

`AUTHORIZATION_ERROR`

권한이 설정되지 않았거나 잘못 구성되었습니다.

`FUNCTION_IN_FAILED_STATE`

함수 코드가 컴파일되지 않거나 복구할 수 없는 예외가 발생했거나 잘못된 배포가 발생했습니다.

Lambda 이벤트 예제

이벤트는 모든 소스로부터의 단일 요청을 나타내며 요청된 API 작업, 작업 날짜와 시간, 요청 파라미터 등에 대한 정보가 들어 있습니다. CloudTrail 로그 파일은 퍼블릭 API 직접 호출의 주문 스택 추적이 아니므로 이벤트가 특정 순서로 표시되지 않습니다.

다음은 `GetFunction` 및 `DeleteFunction` 작업의 CloudTrail 로그 항목을 보여주는 예제입니다.

> **Note**
>
> `eventName`은 `"GetFunction20150331"`과 같은 날짜 및 버전 정보를 포함할 수 있지만, 여전히 동일한 퍼블릭 API를 참조합니다.


```
{
  "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 레코드 콘텐츠에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 CloudTrail record contents를 참조하세요.

AWS X-Ray로 Lambda 함수 간접 호출 시각화

AWS X-Ray를 사용하여 애플리케이션의 구성 요소를 시각화하고, 성능 병목 현상을 식별하고, 오류가 발생한 요청을 문제 해결할 수 있습니다. Lambda 함수는 추적 데이터를 X-Ray로 보내고, X-Ray는 데이터를 처리하여 서비스 맵과 검색 가능한 추적 요약을 생성합니다.

함수를 호출하는 서비스에서 X-Ray 추적을 활성화한 경우 Lambda는 자동으로 추적을 X-Ray로 보냅니다. Amazon API Gateway와 같은 업스트림 서비스 또는 X-Ray SDK로 계측되는 Amazon EC2에서 호스팅되는 애플리케이션은 들어오는 요청을 샘플링하고 Lambda에게 추적 전송 여부를 지시하는 추적 헤더를 추가합니다. Amazon SQS와 같은 업스트림 메시지 생산자의 트레이스가 다운스트림 Lambda 함수의 트레이스에 자동으로 연결되므로 전체 애플리케이션을 종합적으로 파악할 수 있습니다. 자세한 내용은 AWS X-Ray 개발자 가이드에서 이벤트 중심 애플리케이션 추적을 참조하세요.

> **Note**
>
> X-Ray 트레이스는 현재 Amazon Managed Streaming for Apache Kafka(Amazon MSK), 자체 관리형 Apache Kafka, ActiveMQ 및 RabbitMQ를 사용하는 Amazon MQ 또는 Amazon DocumentDB 이벤트 소스 매핑을 사용하는 Lambda 함수에 대해 지원되지 않습니다.

콘솔을 사용하여 Lambda 함수에 대한 활성 추적을 전환하려면 다음 단계를 따르십시오.

활성 추적 켜기

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성(Configuration)을 선택한 다음 모니터링 및 운영 도구(Monitoring and operations tools)를 선택합니다.
4. 편집을 선택합니다.
5. X-Ray에서 활성 추적을 켭니다.
6. Save(저장)를 선택합니다.

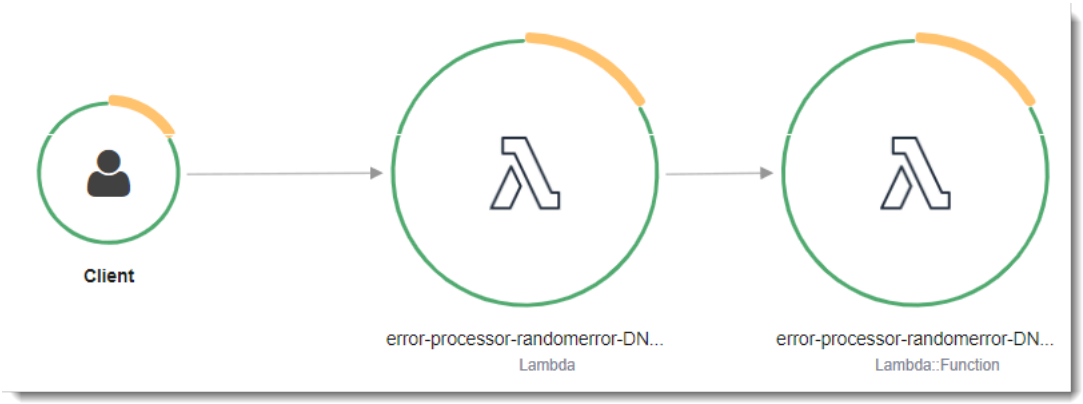
> **요금**
>
> X-Ray 추적을 AWS 프리 티어의 일부로서 특정 한도까지 매월 무료로 사용할 수 있습니다. 해당 한도를 초과하면 추적 저장 및 검색에 대한 X-Ray 요금이 부과됩니다. 자세한 내용은 AWS X-Ray 요금을 참조하십시오.

함수에 추적 데이터를 X-Ray로 업로드할 권한이 있어야 합니다. Lambda 콘솔에서 추적을 활성화하면 Lambda가 필요한 권한을 함수의 실행 역할에 추가합니다. 그렇지 않으면 실행 역할에 AWSXRayDaemonWriteAccess 정책을 추가합니다.

X-Ray는 애플리케이션에 대한 모든 요청을 추적하지 않습니다. X-Ray는 모든 요청의 대표 샘플을 여전히 제공하면서 추적이 효율적으로 수행되도록 샘플링 알고리즘을 적용합니다. 샘플링 요율은 초당 요청이 1개이며 추가 요청의 5퍼센트입니다.

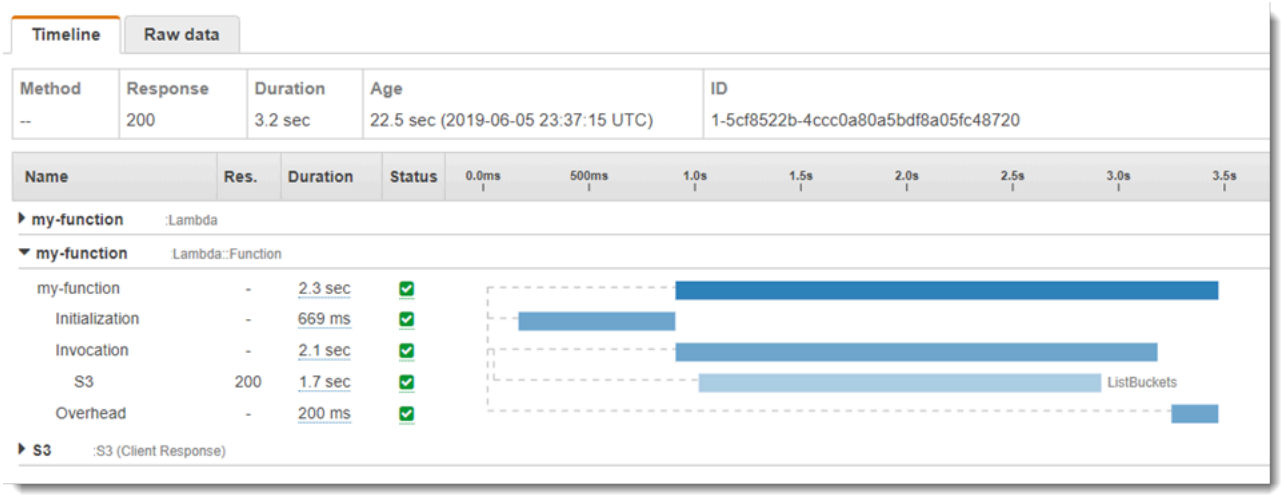
> **Note**
>
> 함수에 대해 X-Ray 샘플링 요율을 구성할 수 없습니다.

X-Ray에서 추적은 하나 이상의 서비스에서 처리되는 요청에 대한 정보를 기록합니다. Lambda는 각 추적에 대해 2개의 세그먼트를 기록하고, 이에 따라 서비스 그래프에 2개의 노드가 생성됩니다. 다음 이미지에서는 이 두 노드를 강조 표시합니다.



왼쪽의 첫 번째 노드는 호출 요청을 수신하는 Lambda 서비스를 나타냅니다. 두 번째 노드는 특정 Lambda 함수를 나타냅니다. 다음 예에서는 이러한 2개의 세그먼트가 있는 추적을 보여줍니다. 둘 다 이름이 my-function 이지만 하나는 오리진이 `AWS::Lambda`이고 다른 하나는 오리진이 `AWS::Lambda::Function`입니다. `AWS::Lambda` 세그먼트에 오류가 표시되면 Lambda 서비스에

문제가 있는 것입니다. AWS::Lambda::Function 세그먼트에 오류가 표시되면 함수에 문제가 있는 것입니다.



함수 세그먼트(AWS::Lambda::Function)는 Initialization, Invocation, Restore(Lambda SnapStart 전용) 및 Overhead의 하위 세그먼트와 함께 제공됩니다. 자세한 내용은 Lambda 실행 환경 수명 주기를 참조하세요.

> **Note**
>
> X-Ray는 Lambda 함수에서 처리되지 않은 예외를 Error 상태로 취급합니다. X-Ray는 Lambda에 내부 서버 오류가 발생한 경우에만 Fault 상태를 기록합니다. 자세한 내용은 X-Ray 개발자 가이드에서 오류, 결함 및 예외를 참조하세요.

Initialization 하위 세그먼트는 Lambda 실행 환경 수명 주기의 초기화 단계를 나타냅니다. 이 단계 중에 Lambda는 사용자가 구성한 리소스로 실행 환경을 만들거나 고정 해제하고, 함수 코드와 모든 계층을 다운로드하고, 익스텐션을 초기화하고, 런타임을 초기화하고, 함수의 초기화 코드를 실행합니다.

Invocation 하위 세그먼트는 Lambda가 함수 핸들러를 호출하는 호출 단계를 나타냅니다. 이것은 런타임 및 익스텐션 등록으로 시작되며 런타임에서 응답을 보낼 준비가 되면 끝납니다.

(Lambda SnapStart만 해당) Restore 하위 세그먼트는 Lambda가 스냅샷을 복원하고, 런타임(JVM)을 로드하고, afterRestore 런타임 후크를 실행하는 데 걸리는 시간을 보여줍니다. 스냅샷 복원 프로세스에는 microVM 외부 작업에 소요되는 시간이 포함될 수 있습니다. 이 시간은 Restore 하위 세그먼트에서 보고됩니다. 스냅샷을 복원하기 위해 microVM 외부에서 소요된 시간에 대한 요금은 부과되지 않습니다.

`Overhead` 하위 세그먼트는 런타임이 응답을 보내는 시간과 다음 호출에 대한 신호 사이에 발생하는 위상을 나타냅니다. 이 시간 동안 런타임은 호출과 관련된 모든 태스크를 완료하고 샌드박스를 고정할 준비를 합니다.

> **Note**
>
> 경우에 따라 X-Ray 트레이스에서 함수 초기화 단계와 호출 단계 사이의 간격이 크게 나타날 수 있습니다. 프로비저닝된 동시성을 사용하는 함수의 경우 이는 Lambda가 호출 전에 함수 인스턴스를 초기화하기 때문입니다. 예약되지 않은(온디맨드) 동시성을 사용하는 함수의 경우 호출이 없더라도 Lambda가 함수 인스턴스를 사전에 초기화할 수 있습니다. 시각적으로 이 두 경우 모두 초기화 단계와 호출 단계 사이의 시간 차이로 표시됩니다.

> **Important**
>
> Lambda에서는 X-Ray SDK를 사용하여 `Invocation` 하위 세그먼트를 다운스트림 호출, 주석 및 메타데이터에 대한 추가 하위 세그먼트로 확장할 수 있습니다. 함수 세그먼트에 직접 액세스하거나 핸들러 호출 범위 외부에서 수행된 작업을 기록할 수 없습니다.

Lambda에서의 추적에 대한 언어별 소개는 다음 주제를 참조하세요.

- AWS Lambda에서 Node.js 코드 계측
- AWS Lambda에서 Python 코드 계측
- AWS Lambda에서 Ruby 코드 계측
- AWS Lambda에서 Java 코드 계측
- AWS Lambda에서 Go 코드 계측
- AWS Lambda에서 C# 코드 계측

활성 계측을 지원하는 서비스의 전체 목록은 AWS X-Ray 개발자 안내서에서 지원되는 AWS 서비스를 참조하세요.

단원

- 실행 역할 권한
- AWS X-Ray 데몬
- Lambda API로 활성 추적 활성화

- AWS CloudFormation으로 활성 추적 활성화

실행 역할 권한

X-Ray에 추적 데이터를 보내려면 Lambda에 다음 권한이 필요합니다. 이러한 권한을 함수의 실행 역할에 추가합니다.

- xray:PutTraceSegments
- xray:PutTelemetryRecords

이러한 권한은 AWSXRayDaemonWriteAccess 관리형 정책에 포함됩니다.

AWS X-Ray 데몬

X-Ray API로 추적 데이터를 직접 보내는 대신 X-Ray SDK는 데몬 프로세스를 사용합니다. AWS X-Ray 데몬은 Lambda 환경에서 실행되고 세그먼트 및 하위 세그먼트를 포함하는 UDP 트래픽을 수신하는 애플리케이션입니다. 수신 데이터를 버퍼링하고 배치로 X-Ray에 작성하여 호출을 추적하는 데 필요한 처리 및 메모리 오버헤드를 줄입니다.

Lambda 런타임을 사용하면 데몬이 함수에 구성된 메모리의 최대 3% 또는 16MB 중 더 큰 메모리를 사용할 수 있습니다. 호출 중에 함수의 메모리가 부족하면 런타임은 먼저 데몬 프로세스를 종료하여 메모리를 확보합니다.

데몬 프로세스는 Lambda에 의해 완전히 관리되며 사용자가 구성할 수 없습니다. 함수 호출에 의해 생성된 모든 세그먼트는 Lambda 함수와 동일한 계정에 기록됩니다. 데몬은 다른 계정으로 리디렉션하도록 구성할 수 없습니다.

자세한 내용은 X-Ray 개발자 안내서의 X-Ray 데몬을 참조하세요.

Lambda API로 활성 추적 활성화

AWS CLI 또는 AWS SDK를 사용하여 추적 구성을 관리하려면 다음 API 작업을 사용합니다.

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

다음 예제 AWS CLI 명령은 my-function이라는 함수에 대한 활성 추적을 사용 설정합니다.

```
aws lambda update-function-configuration \
--function-name my-function \
--tracing-config Mode=Active
```

추적 모드는 함수 버전을 게시할 때 버전별 구성의 일부입니다. 게시된 버전에 대한 추적 모드는 변경할 수 없습니다.

AWS CloudFormation으로 활성 추적 활성화

AWS CloudFormation 템플릿에서 `AWS::Lambda::Function` 리소스에 대한 추적을 활성화하려면 `TracingConfig` 속성을 사용합니다.

Example function-inline.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

AWS Serverless Application Model(AWS SAM) `AWS::Serverless::Function` 리소스의 경우 `Tracing` 속성을 사용합니다.

Example template.yml – 추적 구성

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Amazon CloudWatch Lambda Insights로 함수 성능 모니터링

Amazon CloudWatch Lambda Insights는 서버리스 애플리케이션에 대한 Lambda 함수 런타임 성능 지표 및 로그를 수집하고 집계합니다. 이 페이지에서는 Lambda Insights를 활성화하고 이를 사용하여 Lambda 함수 문제를 진단하는 방법을 설명합니다.

단원

Lambda Insights에서 서버리스 애플리케이션을 모니터링하는 방식

CloudWatch Lambda Insights는 AWS Lambda에서 실행되는 서버리스 애플리케이션에 대한 모니터링 및 문제 해결 솔루션입니다. 이 솔루션은 CPU 시간, 메모리, 디스크 및 네트워크 사용량을 포함한 시스템 수준 지표를 수집, 집계 및 요약합니다. 또한 콜드 스타트 및 Lambda 작업자 종료와 같은 진단 정보를 수집, 집계 및 요약하여 Lambda 함수 관련 문제를 격리하고 신속하게 해결할 수 있도록 지원합니다.

Lambda Insights는 Lambda 계츠으로 제공되는 새로운 CloudWatch Lambda Insights 익스텐션을 사용합니다. Lambda 함수에서 지원되는 런타임에 대해 이 익스텐션을 활성화하면 시스템 수준 지표를 수집하고 Lambda 함수를 호출할 때마다 단일 성능 로그 이벤트를 내보냅니다. CloudWatch는 포함된 지표 서식을 사용하여 로그 이벤트에서 지표를 추출합니다. 자세한 내용은 AWS Lambda 익스텐션 사용을 참조하세요.

Lambda Insights 계층은 `/aws/lambda-insights/` 로그 그룹에 대해 `CreateLogStream` 및 `PutLogEvents`를 확장합니다.

요금

Lambda 함수에 대해 Lambda Insights를 활성화하면 Lambda Insights는 함수당 8개의 지표를 보고하고 모든 함수 호출은 약 1KB 정도의 로그 데이터를 CloudWatch로 전송합니다. Lambda Insights가 함수에 대해 보고한 지표 및 로그에 대해서만 비용을 지불하면 됩니다. 최소 요금이나 필수 서비스 사용 정책은 없습니다. 함수가 호출되지 않은 경우 Lambda Insights에 대한 비용을 지불하지 않습니다. 요금의 예시는 Amazon CloudWatch 요금을 참조하세요.

지원되는 런타임

Lambda 익스텐션을 지원하는 모든 런타임에 Lambda Insights를 사용할 수 있습니다.

Lambda 콘솔에서 Lambda Insights 사용

신규 및 기존 Lambda 함수에 대한 Lambda Insights 확장 모니터링을 활성화할 수 있습니다. 지원되는 런타임에 대해 Lambda 콘솔의 함수에서 Lambda Insights를 활성화하면 Lambda는 Lambda Insights 익스텐션을 함수에 계층으로 추가하고 해당 `CloudWatchLambdaInsightsExecutionRolePolicy` 정책을 함수의 실행 역할에 연결합니다.

Lambda 콘솔에서 Lambda Insights를 사용하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택합니다.
4. 왼쪽 메뉴에서 모니터링 및 작업 도구를 선택합니다.
5. 추가 모니터링 도구 창에서 편집을 선택합니다.
6. CloudWatch Lambda Insights에서 확장 모니터링을 켭니다.
7. Save(저장)를 선택합니다.

프로그래밍 방식으로 Lambda Insights 활성화

AWS Command Line Interface(AWS CLI), AWS Serverless Application Model(SAM) CLI, AWS CloudFormation 또는 AWS Cloud Development Kit (AWS CDK)를 사용해 Lambda Insights를 활성화할 수도 있습니다. 지원되는 런타임에 대한 함수에서 프로그래밍 방식으로 Lambda Insights를 활성화하면 CloudWatch는 `CloudWatchLambdaInsightsExecutionRolePolicy` 정책을 함수의 실행 역할에 연결합니다.

자세한 내용은 Amazon CloudWatch 사용 설명서의 Lambda Insights 시작하기를 참조하세요.

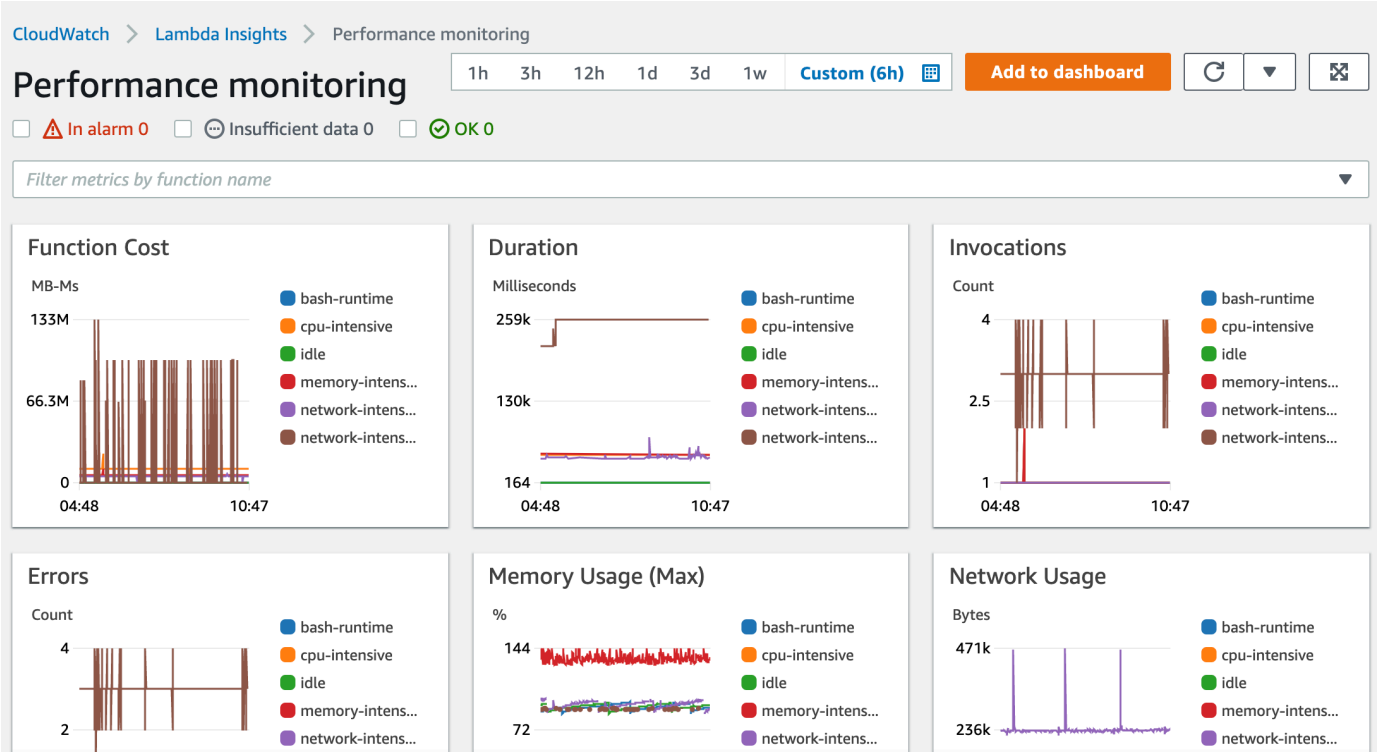
Lambda 인사이트 대시보드 사용

CloudWatch 콘솔의 Lambda Insights 대시보드에는 다중 함수 개요와 단일 함수 보기의 두 가지 보기가 있습니다. 다중 함수 개요에는 현재 AWS 계정 및 리전의 Lambda 함수에 대한 런타임 지표가 집계됩니다. 단일 함수 보기에는 단일 Lambda 함수에 대해 사용 가능한 런타임 지표가 표시됩니다.

CloudWatch 콘솔에서 Lambda Insights 대시보드 다중 함수 개요를 사용하여 사용률이 높거나 낮은 Lambda 함수를 식별할 수 있습니다. CloudWatch 콘솔에서 Lambda Insights 대시보드 단일 함수 보기를 사용하여 개별 요청 문제를 해결할 수 있습니다.

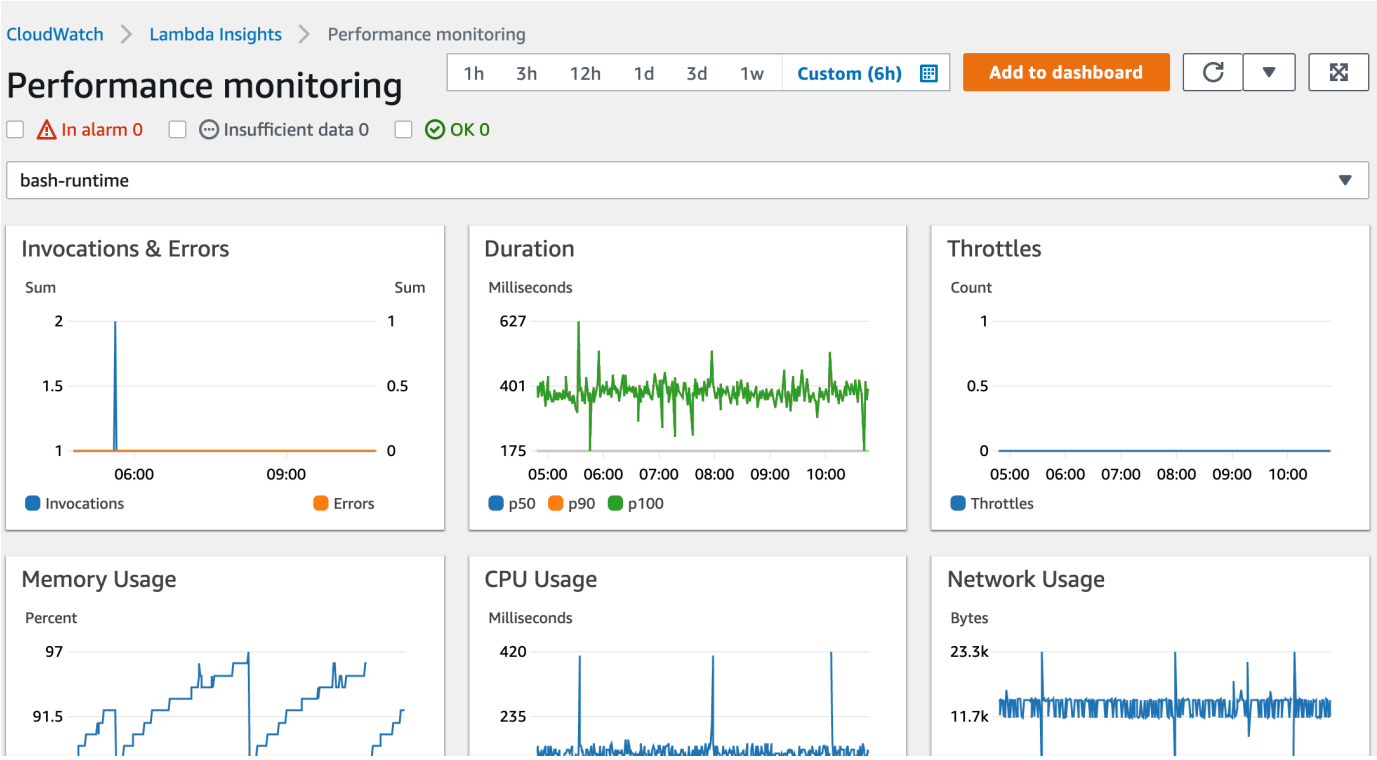
모든 함수에 대한 런타임 지표를 보려면

1. CloudWatch 콘솔에서 다중 함수 페이지를 엽니다.
2. 미리 정의된 시간 범위 중에서 선택하거나 사용자 지정 시간 범위를 선택합니다.
3. (선택 사항) 대시보드에 추가를 선택하여 CloudWatch 대시보드에 위젯을 추가합니다.



단일 함수의 런타임 지표를 보려면

1. CloudWatch 콘솔에서 단일 함수 페이지를 엽니다.
2. 미리 정의된 시간 범위 중에서 선택하거나 사용자 지정 시간 범위를 선택합니다.
3. (선택 사항) 대시보드에 추가를 선택하여 CloudWatch 대시보드에 위젯을 추가합니다.



자세한 내용은 CloudWatch 대시보드에서 위젯 생성 및 사용을 참조하세요.

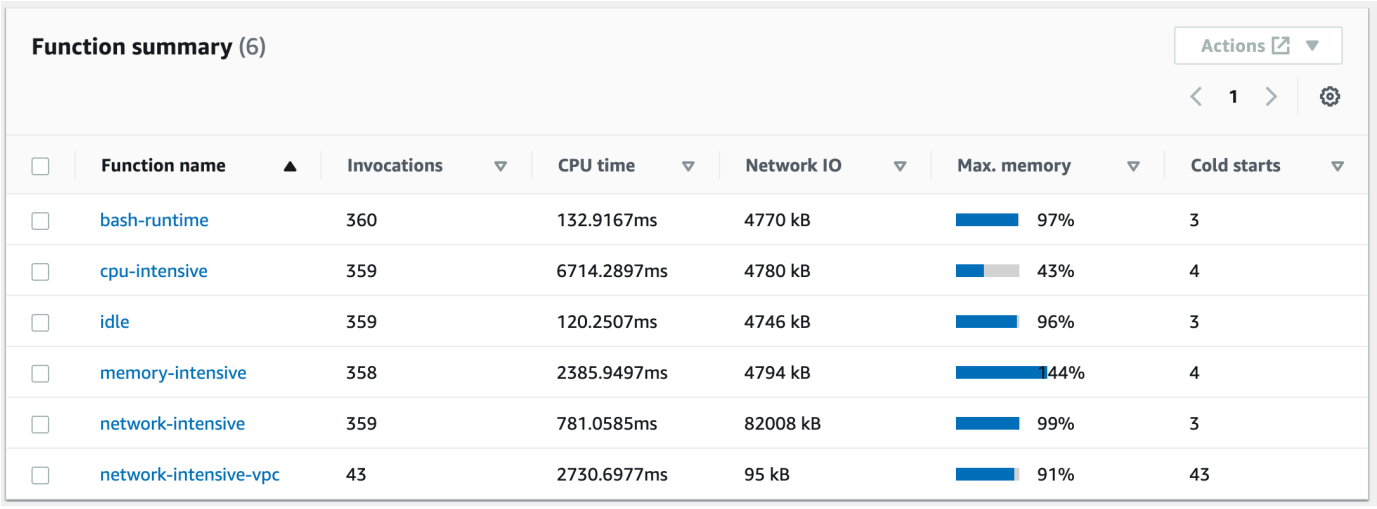
함수 이상을 탐지하는 예제 워크플로

Lambda Insights 대시보드의 다중 함수 개요를 사용하여 함수에 대한 컴퓨팅 메모리 이상을 식별하고 탐지할 수 있습니다. 예를 들어 다중 함수 개요에서 함수가 많은 양의 메모리를 사용하고 있는 것으로 나타날 경우 메모리 사용량(Memory Usage) 창에서 자세한 메모리 사용률 지표를 볼 수 있습니다. 그런 다음 지표 대시보드로 이동하여 이상 탐지를 활성화하거나 경보를 생성할 수 있습니다.

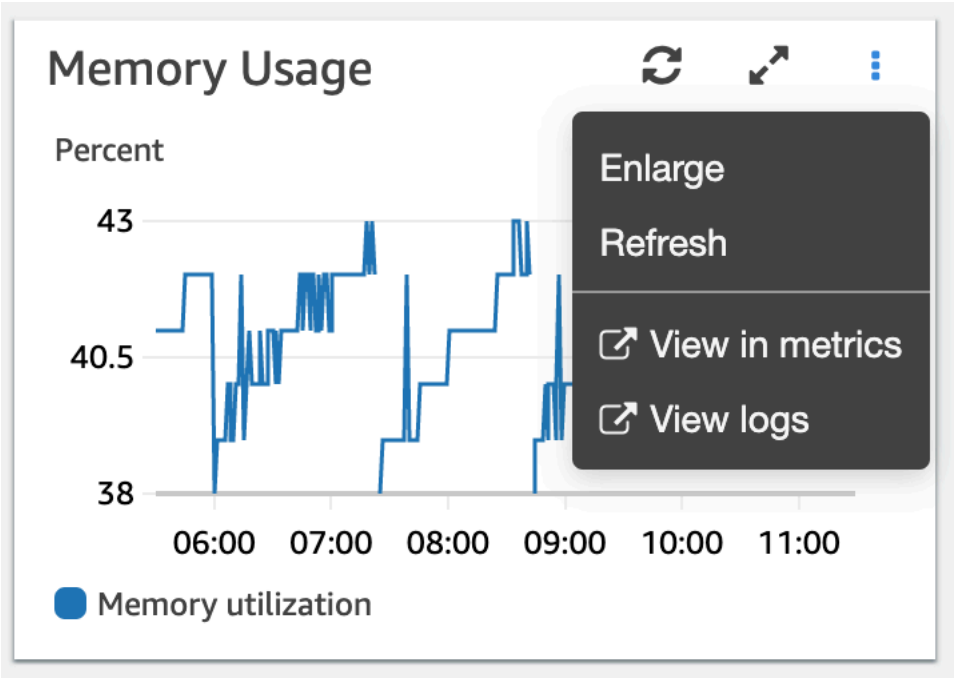
함수에 대한 이상 탐지를 활성화하려면

1. CloudWatch 콘솔에서 다중 함수 페이지를 엽니다.
2. 함수 요약(Function summary)에서 함수 이름을 선택합니다.

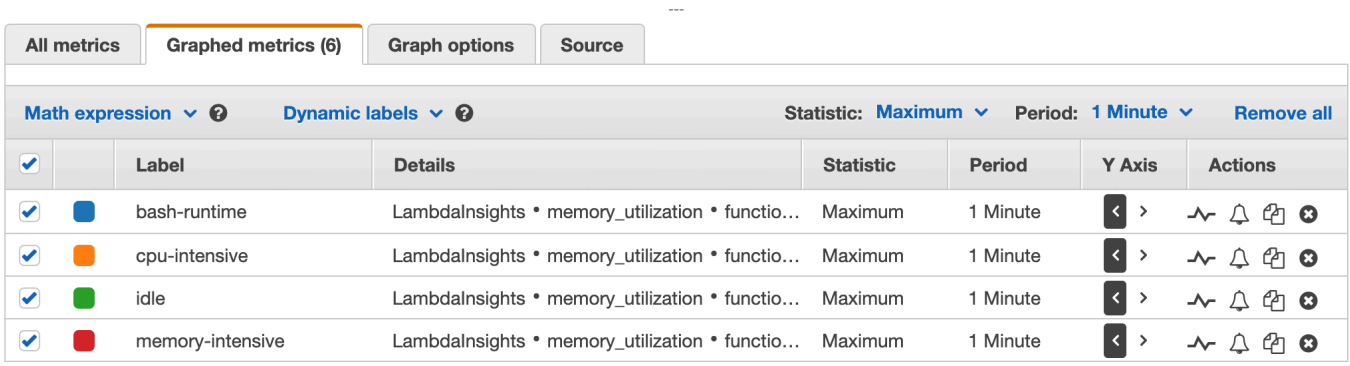
   함수 런타임 지표가 표시된 단일 함수 보기가 열립니다.



3. 메모리 사용량(Memory Usage) 창에서 세로 점 3개 모양의 아이콘을 선택한 다음 지표에서 보기(View in metrics)를 선택하여 지표(Metrics) 대시보드를 엽니다.



4. 그래프로 표시된 지표(Graphed metrics) 탭의 작업(Actions) 열에서 첫 번째 아이콘을 선택하여 함수에 대한 이상 탐지를 활성화합니다.



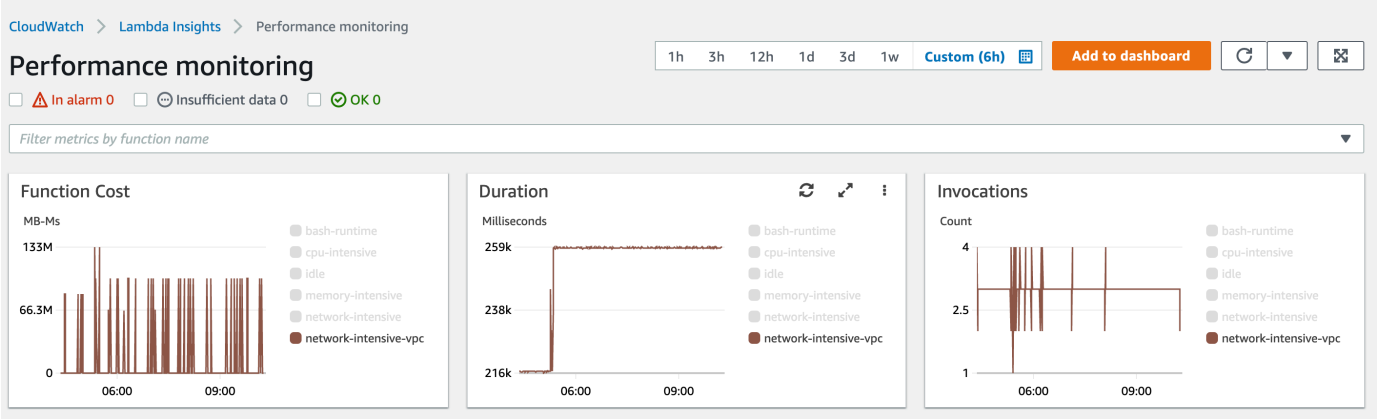
자세한 내용은 CloudWatch 이상 탐지 사용을 참조하세요.

쿼리를 사용하여 함수 문제를 해결하는 예제 워크플로

Lambda Insights 대시보드의 단일 함수 보기를 사용하여 함수 기간 스파이크의 근본 원인을 식별할 수 있습니다. 예를 들어, 다중 함수 개요에 함수 기간이 크게 증가한 것으로 나타날 경우 기간(Duration) 창에서 각 함수를 일시 중지하거나 선택하여 이러한 증가를 일으킨 함수를 확인할 수 있습니다. 그런 다음 단일 함수 보기로 이동하여 애플리케이션 로그(Application logs)를 검토해 근본 원인을 파악할 수 있습니다.

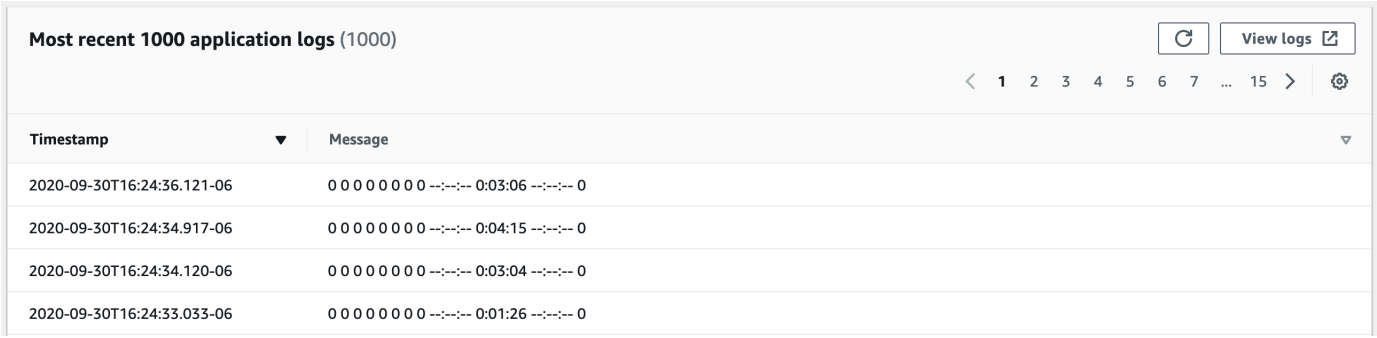
함수에 대한 쿼리를 실행하려면

1. CloudWatch 콘솔에서 다중 함수 페이지를 엽니다.
2. 기간(Duration) 창에서 기간 지표를 필터링할 함수를 선택합니다.

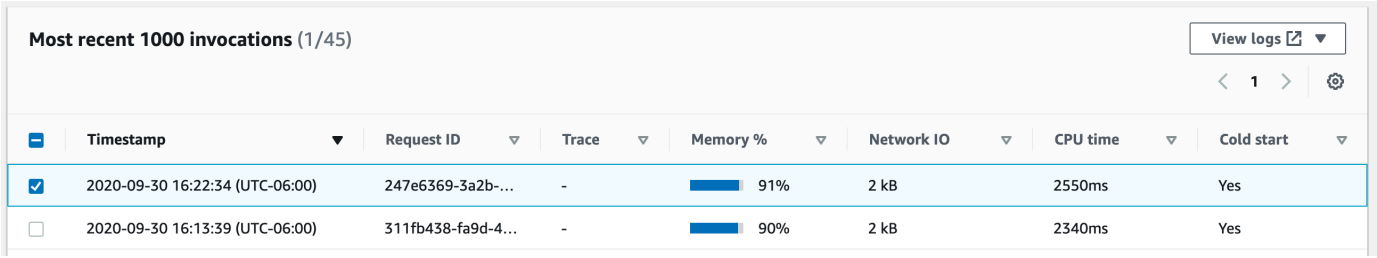


3. 단일 함수 페이지를 엽니다.
4. 함수 이름으로 지표 필터링(Filter metrics by function name) 드롭다운 목록을 선택한 다음 함수를 선택합니다.

5. 최근 1000개의 애플리케이션 로그(Most recent 1000 application logs)를 보려면 애플리케이션 로그(Application logs) 탭을 선택합니다.
6. 타임스탬프(Timestamp) 및 메시지(Message)를 검토하여 문제를 해결할 호출 요청을 식별합니다.

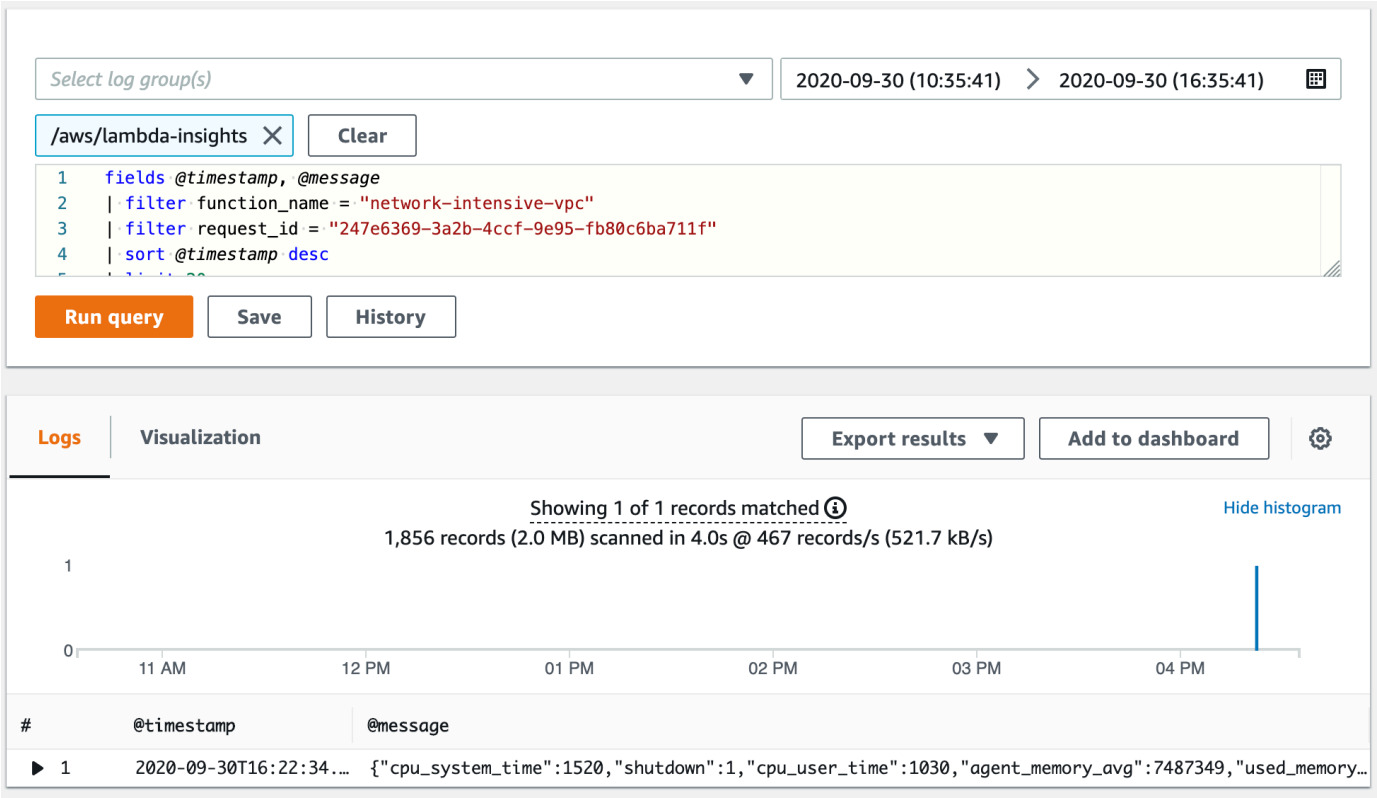


7. 최근 1000개의 호출(Most recent 1000 invocations)을 표시하려면 호출(Invocations) 탭을 선택합니다.
8. 문제를 해결하려는 호출 요청에 대한 타임스탬프(Timestamp) 또는 메시지(Message)를 선택합니다.



9. 로그 보기(View logs) 드롭다운 목록을 선택한 다음 성능 로그 보기(View performance logs)를 선택합니다.

   함수에 대한 자동 생성된 쿼리가 Logs Insights 대시보드에서 열립니다.
10. 쿼리 실행(Run query)을 선택하여 호출 요청에 대한 로그(Logs) 메시지를 생성합니다.



다음 단계

- Amazon CloudWatch 사용 설명서의 대시보드 만들기에서 CloudWatch Logs 대시보드를 만드는 방법을 알아보세요.
- Amazon CloudWatch 사용 설명서의 대시보드에 쿼리 추가 또는 쿼리 결과 내보내기에서 CloudWatch Logs 대시보드에 쿼리를 추가하는 방법을 알아보세요.

Lambda 함수와 함께 CodeGuru 프로파일러 사용

Amazon CodeGuru 프로파일러를 사용하여 Lambda 함수의 런타임 성능에 대한 통찰력을 얻을 수 있습니다. 이 페이지는 Lambda 콘솔에서 CodeGuru 프로파일러를 활성화하는 방법을 설명합니다.

Sections

지원되는 런타임

함수 런타임이 Python3.8, Python3.9, 자바 8 (아마존 리눅스 2 포함), 자바 11 또는 자바 17인 경우 Lambda 콘솔에서 CodeGuru 프로파일러를 활성화할 수 있습니다. 추가 런타임 버전의 CodeGuru 경우 프로파일러를 수동으로 활성화할 수 있습니다.

- Java 런타임에 대해서는 AWS Lambda에서 실행되는 Java 애플리케이션 프로파일링을 참조하세요.
- Python 런타임에 대해서는 AWS Lambda에서 실행되는 Python 애플리케이션 프로파일링을 참조하세요.

> **Note**
>
> CodeGuru 프로파일러는 현재 x86_64 아키텍처를 사용하는 함수만 지원합니다.

Lambda 콘솔에서 CodeGuru 프로파일러 활성화

이 섹션에서는 Lambda 콘솔에서 CodeGuru 프로파일러를 활성화하는 방법을 설명합니다.

Lambda 콘솔에서 CodeGuru 프로파일러를 활성화하려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택합니다.

4. 모니터링 및 작업 도구 창에서 편집을 선택합니다.
5. Amazon CodeGuru 프로파일러에서 코드 프로파일링을 활성화합니다.
6. 저장을 선택합니다.

활성화 후 이름이 있는 프로파일러 그룹을 CodeGuru 자동으로 생성합니다. `aws-lambda-<your-function-name>` 콘솔에서 이름을 변경할 수 있습니다. CodeGuru

Lambda 콘솔에서 CodeGuru 프로파일러를 활성화하면 어떻게 됩니까?

콘솔에서 CodeGuru 프로파일러를 활성화하면 Lambda는 사용자를 대신하여 다음 작업을 자동으로 수행합니다.

- Lambda는 프로파일러 CodeGuru 계층을 함수에 추가합니다. 자세한 내용은 Amazon CodeGuru 프로파일러 사용 설명서의 AWS Lambda 레이어 사용을 참조하십시오.
- Lambda는 함수에 환경 변수도 추가합니다. 정확한 값은 런타임에 따라 다릅니다.

  환경 변수

런타임	키	값
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`

- Lambda는 함수의 실행 역할에 `AmazonCodeGuruProfilerAgentAccess` 정책을 추가합니다.

> **Note**
>
> 콘솔에서 CodeGuru 프로파일러를 비활성화하면 Lambda는 프로파일러 레이어를 함수에서 자동으로 제거합니다. CodeGuru 그러나 Lambda는 실행 역할에서 환경 변수나 `AmazonCodeGuruProfilerAgentAccess` 정책을 제거하지 않습니다.

다음 단계

- Amazon CodeGuru Profiler 사용 설명서의 시각화 작업에서 프로파일러 그룹이 수집한 데이터에 대해 자세히 알아보십시오.

다른 AWS 서비스를 사용하는 워크플로 예제

AWS Lambda는 다른 AWS 서비스와 통합하여 Lambda 함수를 모니터링, 추적, 디버그 및 문제 해결하는 데 도움을 줍니다. 이 페이지에서는 AWS X-Ray 및 AWS Trusted Advisor에서 Lambda 함수를 추적하고 문제 해결하는 데 사용할 수 있는 워크플로를 보여줍니다.

Sections

사전 조건

다음 섹션에서는 AWS X-Ray 및 Trusted Advisor를 사용하여 Lambda 함수의 문제를 해결하는 단계를 설명합니다.

사용 AWS X-Ray

이 페이지에서 AWS X-Ray 워크플로를 완료하려면 Lambda 콘솔에서 AWS X-Ray를 활성화해야 합니다. 실행 역할에 필요한 권한이 없는 경우 Lambda 콘솔에서 실행 역할에 해당 권한을 추가하려고 시도합니다.

Lambda 콘솔에서 AWS X-Ray를 활성화하는 방법

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 구성 탭을 선택합니다.
4. 모니터링 도구(Monitoring tools) 창에서 편집(Edit)을 선택합니다.
5. AWS X-Ray 아래에서 [활성 추적(Active tracing)]을 켭니다.
6. 저장을 선택합니다.

사용 AWS Trusted Advisor

AWS Trusted Advisor는 고객의 AWS 환경을 검사하고 비용 절감, 시스템 가용성과 성능 개선 및 보안 격차 방법에 대한 권장 사항을 제공합니다. Trusted Advisor 검사를 사용하여 AWS 계정의 Lambda 함수와 애플리케이션을 평가할 수 있습니다. 이 검사는 취할 수 있는 권장 단계와 자세한 정보를 위한 리소스를 제공합니다.

- AWS 검사를 위한 Trusted Advisor 지원 계획의 자세한 내용은 지원 계획을 참조하세요.
- Lambda 검사에 대한 자세한 내용은 AWS Trusted Advisor 모범 사례 체크리스트를 참조하세요.
- Trusted Advisor 콘솔 사용 방법에 대한 자세한 내용은 AWS Trusted Advisor 시작하기를 참조하세요.
- Trusted Advisor에 대한 콘솔 액세스 허용 및 거부 방법에 대한 지침은 IAM 정책 예를 참조하세요.

요금

- AWS X-Ray에서는 기록, 검색 및 스캔된 트레이스 수를 기준으로 사용한 만큼만 비용을 지불합니다. 자세한 내용은 AWS X-Ray 요금을 참조하세요.
- Trusted Advisor 비용 최적화 검사는 AWS Business 및 Enterprise 지원 구독에 포함되어 있습니다. 자세한 내용은 AWS Trusted Advisor 요금을 참조하세요.

트레이스 맵을 보기 위한 AWS X-Ray 워크플로 예제

AWS X-Ray활성화한 경우 CloudWatch 콘솔에서 추적 맵을 볼 수 있습니다. 추적 맵은 서비스 엔드포인트와 리소스를 노드로 표시하고 각 노드 및 해당 연결에 대한 트래픽, 지연 시간 및 오류를 강조 표시합니다.

노드를 선택하여 서비스의 해당 부분과 연관된 상관관계가 있는 지표, 로그 및 추적에 대한 자세한 정보를 볼 수 있습니다. 이를 통해 문제와 해당 문제가 애플리케이션에 미치는 영향을 조사할 수 있습니다.

CloudWatch 콘솔을 사용하여 트레이스 맵과 트레이스를 보려면

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 모니터링을 선택합니다.
4. X-Ray 트레이스 보기를 선택합니다.

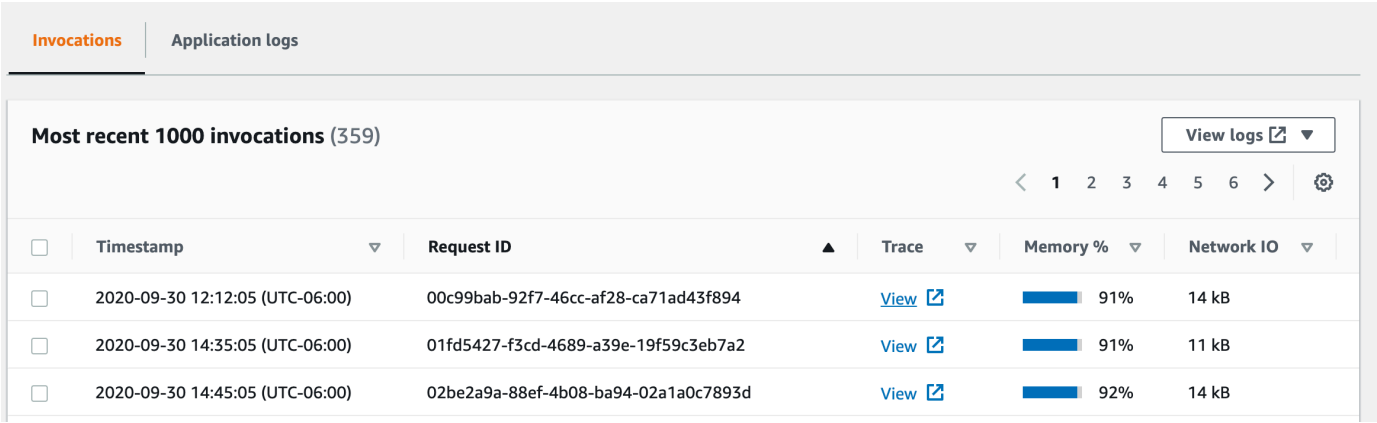
5. 왼쪽 탐색 창의 X-Ray 트레이스에서 트레이스 맵을 선택합니다.
6. 미리 정의된 시간 범위 중에서 선택하거나 사용자 지정 시간 범위를 선택합니다.
7. 요청 문제를 해결하려면 필터를 선택합니다.

트레이스 세부 정보를 보는 AWS X-Ray 워크플로 예제

AWS X-Ray활성화한 경우 CloudWatch Lambda Insights 대시보드의 단일 함수 보기를 사용하여 함수 호출 오류의 분산 추적 데이터를 표시할 수 있습니다. 예를 들어, 애플리케이션 로그 메시지에 오류가 표시되면 트레이스 맵을 열어 분산 트레이스 데이터 및 트랜잭션을 처리하는 다른 서비스를 볼 수 있습니다.

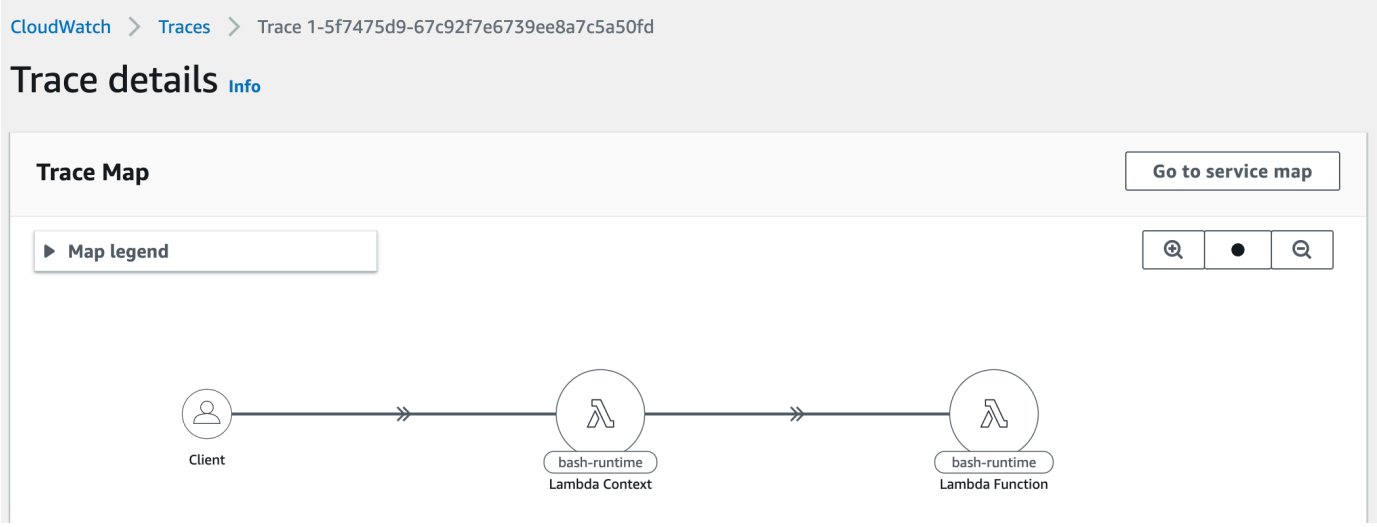
함수의 트레이스 세부 정보를 보는 방법

1. 콘솔에서 단일 함수 보기를 엽니다. CloudWatch
2. [애플리케이션 로그(Application logs)] 탭을 선택합니다.
3. [타임스탬프(Timestamp)] 및 [메시지(Message)]를 검토하여 문제를 해결할 호출 요청을 식별합니다.
4. 최근 1000개의 호출(Most recent 1000 invocations)을 표시하려면 호출(Invocations) 탭을 선택합니다.



5. [요청 ID(Request ID)] 열을 선택하여 항목을 사전의 오름차순으로 정렬합니다.
6. [추적(Trace)] 열에서 [보기(View)]를 선택합니다.

   트레이스 세부 정보 페이지가 트레이스 맵 보기에서 열립니다.



권장 사항을 보기 위한 AWS Trusted Advisor 워크플로 예제

Trusted Advisor는 모든 AWS 리전의 Lambda 함수를 검사하여 잠재적인 비용 절감 효과가 가장 높은 함수를 식별하고 실행 가능한 최적화 권장 사항을 제공합니다. 함수 실행 시간, 청구 기간, 사용된 메모리, 구성된 메모리, 시간 초과 구성, 오류 등의 Lambda 사용 데이터를 분석합니다.

예를 들어 오류율이 높은 Lambda 함수 검사는 Lambda 함수를 AWS X-Ray CloudWatch 사용하거나 Lambda 함수의 오류를 탐지하도록 권장합니다.

오류율이 높은 함수를 검사하려면

1. Trusted Advisor 콘솔을 엽니다.
2. 비용 최적화 범주를 선택합니다.
3. 오류율이 높은 AWS Lambda 함수까지 아래로 스크롤합니다. 섹션을 확장하여 결과와 권장 조치를 봅니다.

다음 단계

- X-Ray 트레이스 맵 사용에서 트레이스, 지표, 로그 및 경보를 통합하는 방법을 자세히 알아봅니다.
- 웹 서비스로 Trusted Advisor 사용에서 Trusted Advisor 검사 목록을 가져오는 방법에 대해 자세히 알아보세요.

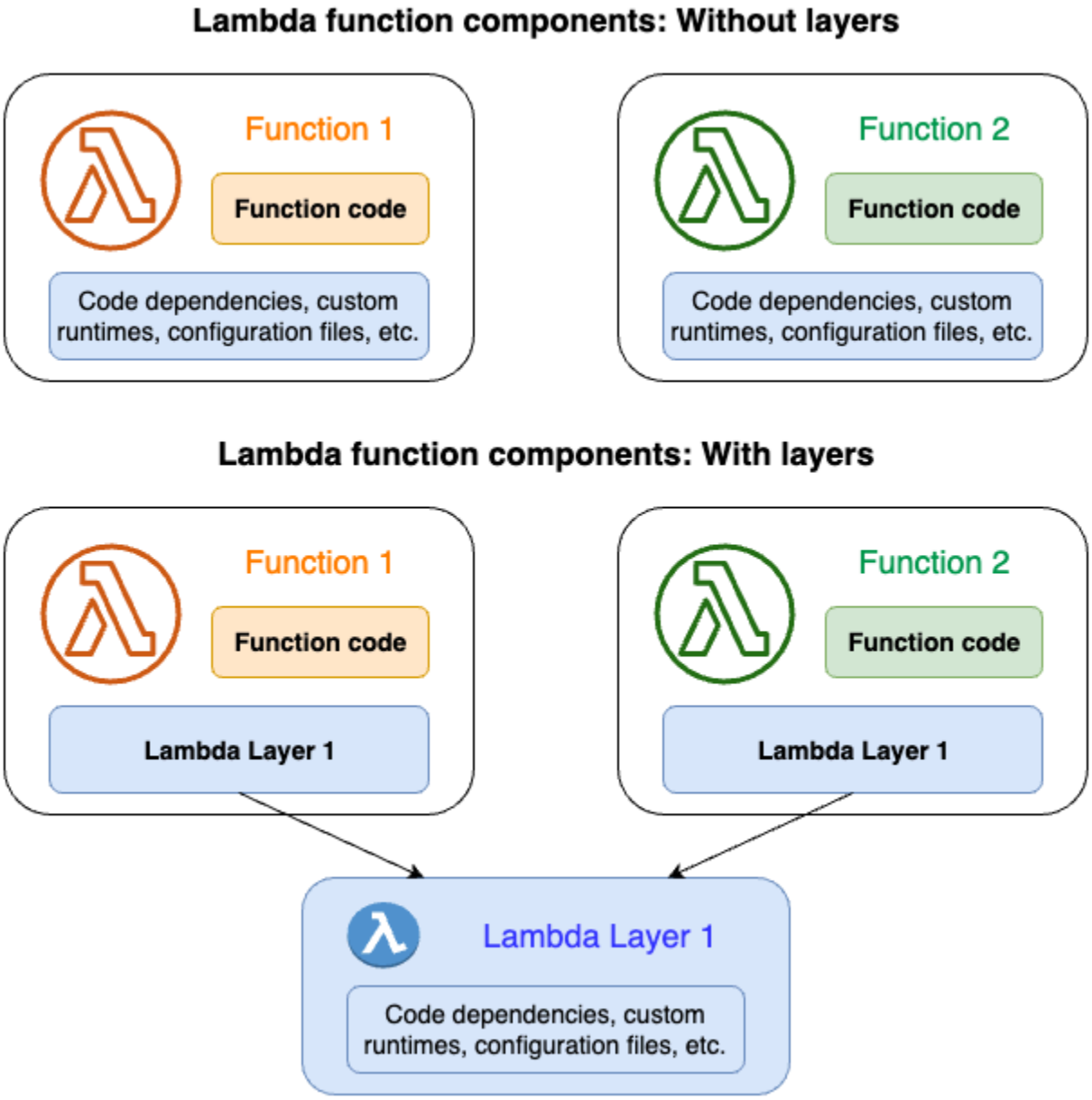
계층으로 Lambda 종속성 관리

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다.

계층 사용을 고려하는 데에는 여러 가지 이유가 있습니다.

- 배포 패키지의 크기를 줄이기 위해. 모든 함수 종속 항목을 함수 코드와 함께 배포 패키지에 포함하는 대신 계층에 배치합니다. 이렇게 하면 배포 패키지가 작고 체계적으로 유지됩니다.
- 핵심 함수 로직을 종속 항목과 분리하기 위해. 계층을 사용하면 함수 코드와 독립적으로 함수 종속 항목을 업데이트할 수 있으며 그 반대의 경우도 마찬가지입니다. 이렇게 하면 관심사를 분리하고 함수 로직에 집중할 수 있습니다.
- 여러 함수에서 종속 항목을 공유하기 위해. 계층을 생성한 후 계정의 여러 함수에 적용할 수 있습니다. 계층이 없으면 각 개별 배포 패키지에 동일한 종속 항목을 포함해야 합니다.
- Lambda 콘솔 코드 편집기를 사용하기 위해. 코드 편집기는 함수 코드의 부분 업데이트를 빠르게 테스트하는 데 유용한 도구입니다. 그러나 배포 패키지 크기가 너무 큰 경우 편집기를 사용할 수 없습니다. 계층을 사용하면 패키지 크기가 줄어들고 코드 편집기를 사용할 수 있습니다.

다음 다이어그램에서는 종속 항목을 공유하는 두 함수 간의 중요 아키텍처 차이를 보여줍니다. 하나는 Lambda 계층을 사용하고 다른 하나는 사용하지 않습니다.



함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 함수 실행 환경의 `/opt` 디렉터리로 추출합니다. 기본적으로 지원되는 모든 Lambda 런타임에는 `/opt` 디렉터리 내의 특정 디렉터리에 대한 경로가 포함되어 있습니다. 이를 통해 함수가 계층 콘텐츠에 액세스할 수 있습니다. 이러한 특정 경로와 계층을 올바르게 패키징하는 방법에 대한 자세한 내용은 the section called “계층 패키징” 섹션을 참조하세요.

함수당 최대 5개의 계층을 포함할 수 있습니다. 또한 .zip 파일 아카이브로 배포된 Lambda 함수에서만 계층을 사용할 수 있습니다. 컨테이너 이미지로 정의된 함수의 경우 컨테이너 이미지를 생성할 때 기본 런타임 및 모든 코드 종속 항목을 패키징합니다. 자세한 내용은 AWS 컴퓨팅 블로그에서 컨테이너 이미지의 Lambda 계층 및 익스텐션 작업을 참조하세요.

주제

계층 사용 방법

계층을 생성하려면 일반 배포 패키지를 생성하는 방법과 유사하게 종속 항목을 .zip 파일로 패키징합니다. 보다 구체적으로, 계층을 생성하고 사용하는 일반적인 프로세스에는 다음 세 단계가 포함됩니다.

- 먼저 계층 콘텐츠를 패키징합니다. 즉, .zip 파일 아카이브를 생성합니다. 자세한 내용은 the section called “계층 패키징” 단원을 참조하십시오.
- 다음으로 Lambda에서 계층을 생성합니다. 자세한 내용은 the section called “계층 생성 및 삭제” 단원을 참조하십시오.
- 함수에 계층을 추가합니다. 자세한 내용은 the section called “계층 추가” 단원을 참조하십시오.

계층 및 계층 버전

계층 버전은 특정 계층 버전의 변경 불가능한 스냅샷입니다. 새 계층을 생성하면 Lambda는 버전 번호가 1인 새 계층 버전을 생성합니다. 계층에 업데이트를 게시할 때마다 Lambda는 버전 번호를 늘리고 새 계층 버전을 생성합니다.

모든 계층 버전은 고유한 Amazon 리소스 이름(ARN)으로 식별됩니다. 함수에 계층을 추가할 때 사용하려는 정확한 계층 버전을 지정해야 합니다.

계층 콘텐츠 패키징

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다.

이 섹션에서는 계층 콘텐츠를 올바르게 패키징하는 방법을 설명합니다. 계층에 대한 개념 정보와 계층 사용을 고려하는 이유에 대한 자세한 내용은 Lambda 계층 섹션을 참조하세요.

계층을 생성하는 첫 번째 단계는 모든 계층 콘텐츠를.zip 파일 아카이브로 번들링하는 것입니다. Lambda 함수는 Amazon Linux에서 실행되기 때문에 계층 콘텐츠는 Linux 환경에서 컴파일하고 빌드할 수 있어야 합니다.

Linux 환경에서 계층 콘텐츠가 제대로 작동하도록 하려면 Docker 또는 AWS Cloud9과 같은 도구를 사용하여 계층 콘텐츠를 만드는 것이 좋습니다. AWS Cloud9은 코드를 실행하고 테스트하기 위해 Linux 서버에 대한 기본 제공 액세스를 제공하는 클라우드 기반 통합 개발 환경(IDE)입니다. 자세한 내용은 AWS Compute Blog의 Using Lambda layers to simplify your development process를 참조하세요.

주제

- 각 Lambda 런타임에 대한 계층 경로

각 Lambda 런타임에 대한 계층 경로

함수에 계층을 추가하면 Lambda는 계층 콘텐츠를 해당 실행 환경의 `/opt` 디렉터리로 추출합니다. 각 Lambda 런타임에 대해 `PATH` 변수에는 `/opt` 디렉터리 내의 특정 폴더 경로가 이미 포함되어 있습니다. `PATH` 변수가 계층 콘텐츠를 가져오도록 하려면 계층 .zip 파일의 종속성이 다음 폴더 경로에 있어야 합니다.

각 Lambda 런타임에 대한 계층 경로

런타임	경로
Node.js	`nodejs/node_modules`
	`nodejs/node14/node_modules` (`NODE_PATH`)
	`nodejs/node16/node_modules` (`NODE_PATH`)
	`nodejs/node18/node_modules` (`NODE_PATH`)
Python	`python`

런타임	경로
	python/lib/ *python3.x* /site-packages (사이트 디렉터리)
Java	java/lib (CLASSPATH)
Ruby	ruby/gems/3.2.0 (GEM_PATH)
	ruby/lib (RUBYLIB)
모든 런타임	bin (PATH)
	lib (LD_LIBRARY_PATH)

다음 예제는 계층 .zip 아카이브에 폴더를 구성하는 방법을 보여줍니다.

Node.js

Example Node.js 용 AWS X-Ray SDK 파일 구조

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example Requests 라이브러리의 파일 구조

```
layer_content.zip
# python
    # lib
        # python3.11
            # site-packages
                # requests
                # <other_dependencies> (i.e. dependencies of the requests package)
                # ...
```

Ruby

Example JSON gem에 대한 파일 구조

```
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 파일의 파일 구조

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example jq 라이브러리의 파일 구조

```
jq.zip
# bin/jq
```

계층 패키징, 생성, 추가에 대한 언어별 지침은 다음 페이지를 참조하세요.

- Python – the section called “계층”
- Java – the section called “계층”

Lambda에서 계층 생성 및 삭제

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다.

이 섹션에서는 Lambda에서 계층을 생성하고 삭제하는 방법을 설명합니다. 계층에 대한 개념 정보와 계층 사용을 고려하는 이유에 대한 자세한 내용은 Lambda 계층 섹션을 참조하세요.

계층 콘텐츠를 패키징한 후 다음 단계는 Lambda에서 계층을 생성하는 것입니다 이 섹션에서는 Lambda 콘솔 또는 Lambda API만 사용하여 계층을 생성하고 삭제하는 방법을 보여줍니다. AWS CloudFormation을 사용하여 계층을 생성하려면 the section called “AWS CloudFormation을 사용한 계층” 섹션을 참조하세요. AWS Serverless Application Model(AWS SAM)을 사용하여 계층을 생성하려면 the section called “AWS SAM을 사용한 계층” 섹션을 참조하세요.

주제

계층 생성

계층을 생성하려면 로컬 시스템이나 Amazon Simple Storage Service(S3)에서 계층으로 .zip 파일 아카이브를 업로드합니다. Lambda는 함수에 대한 실행 환경을 설정할 때 계층 콘텐츠를 `/opt` 디렉토리에 추출합니다.

계층에는 하나 이상의 계층 버전이 있을 수 있습니다. 계층을 작성할 때 Lambda에서 계층 버전을 버전 1로 설정합니다. 언제든지 기존 계층 버전에 대한 권한을 변경할 수 있습니다. 그러나 코드를 업데이트하거나 다른 구성을 변경하려면 계층의 새 버전을 생성해야 합니다.

계층을 생성하려면(콘솔)

1. Lambda 콘솔의 계층 페이지를 엽니다.
2. 계층 생성을 선택합니다.
3. [계층 구성(Layer configuration)]에서 [이름(Name)]에 계층 이름을 입력합니다.
4. (선택 사항) 설명에 계층에 대한 설명을 입력합니다.
5. 계층 코드를 업로드하려면 다음 중 하나를 수행하세요.
    - 컴퓨터에서 .zip 파일을 업로드하려면 [zip 파일 업로드(Upload a .zip file)]를 선택합니다. 그리고 [업로드(Upload)]를 선택하여 로컬 .zip 파일을 선택합니다.

- Amazon S3에서 파일을 업로드하려면 Amazon S3에서 파일 업로드(Upload a file from Amazon S3)를 선택합니다. 그런 다음 Amazon S3 링크 URL에 파일 링크를 입력합니다.

6. (선택 사항) 호환 아키텍처에서 값을 하나 또는 둘 다 선택합니다. 자세한 설명은 the section called "명령 세트(ARM/x86)" 섹션을 참조하세요.
7. (선택 사항) 호환 런타임에서 계층과 호환되는 런타임을 선택합니다.
8. (선택 사항) 라이선스의 경우 필요한 라이선스 정보를 입력합니다.
9. 생성을 선택하세요.

또는 PublishLayerVersionAPI를 사용하여 레이어를 생성할 수도 있습니다. 예를 들어, 이름, 설명 및 .zip 파일 아카이브를 지정하여 `publish-layer-version` AWS Command Line Interface(CLI) 명령을 사용할 수 있습니다. 라이선스 정보, 호환 런타임 및 호환 아키텍처 파라미터는 선택 사항입니다.

```
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"
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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`을 호출할 때마다 계층의 새 버전이 생성됩니다.

계층 버전 삭제

레이어 버전을 삭제하려면 DeleteLayerVersionAPI를 사용합니다. 예를 들어, 계층 이름과 계층 버전을 지정하여 `delete-layer-version` CLI 명령을 사용할 수 있습니다.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

계층 버전을 삭제하면 이를 사용하기 위한 Lambda 함수를 더 이상 구성할 수 없습니다. 그러나 해당 버전을 이미 사용 중인 모든 함수는 이 버전에 계속 액세스할 수 있습니다. 또한 Lambda는 계층 이름에 버전 번호를 재사용하지 않습니다.

함수에 계층 추가

Lambda 계층은 추가 코드 또는 데이터를 포함하는 .zip 파일 아카이브입니다. 계층에는 일반적으로 라이브러리 종속 항목, 사용자 지정 런타임 또는 구성 파일이 포함됩니다.

이 섹션에서는 Lambda 함수에 계층을 추가하는 방법을 설명합니다. 계층에 대한 개념 정보와 계층 사용을 고려하는 이유에 대한 자세한 내용은 Lambda 계층 섹션을 참조하세요.

계층을 사용하도록 Lambda 함수를 구성하려면 먼저 다음을 수행해야 합니다.

- 계층 콘텐츠 패키징
- Lambda에서 계층 생성
- 레이어 버전에서 GetLayerVersionAPI를 호출할 권한이 있는지 확인하세요. AWS 계정의 함수인 경우 사용자 정책에 이 권한이 있어야 합니다. 다른 계정에서 계층을 사용하려면 해당 계정의 소유자가 리소스 기반 정책에서 계정 권한을 부여해야 합니다. 예제는 the section called “계층에 다른 계정에 대한 액세스 권한 부여”을 참조하세요.

Lambda 함수에 최대 5개의 계층을 추가할 수 있습니다. 함수 및 모든 계층의 압축되지 않은 총 크기는 압축 해제된 배포 패키지 크기 할당량인 250MB를 초과할 수 없습니다. 자세한 설명은 Lambda 할당량 섹션을 참조하세요.

함수는 이미 추가한 모든 계층 버전을 계속 사용할 수 있습니다. 이는 해당 계층 버전이 삭제되거나 계층 액세스 권한이 취소된 후에도 마찬가지입니다. 그러나 삭제된 계층 버전을 사용하는 새 함수는 작성할 수 없습니다.

> **Note**
>
> 함수에 추가하는 계층이 함수의 런타임 및 명령 세트 아키텍처와 호환되는지 확인합니다.

함수에 태그 추가(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 구성할 함수를 선택합니다.
3. [계층(Layers)]에서 [계층 추가(Add a layer)]를 선택합니다.
4. 계층 선택에서 계층 소스를 선택합니다.

a. AWS 계층 또는 사용자 지정 계층 계층 소스의 경우 풀다운 메뉴에서 계층을 선택합니다. [버전(Version)의 풀다운 메뉴에서 계층 버전을 선택합니다.

b. ARN 지정 계층 소스의 경우 텍스트 상자에 ARN을 입력하고 확인을 선택합니다. 그런 다음 추가를 선택합니다.

계층을 추가하는 순서에 따라 나중에 Lambda가 계층 콘텐츠를 실행 환경에 병합하는 순서가 결정됩니다. 콘솔을 사용하여 계층 병합 순서를 변경할 수 있습니다.

함수의 계층 병합 순서 업데이트(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 구성할 함수를 선택합니다.
3. [계층(Layers)]에서 [편집(Edit)]을 선택합니다.
4. 계층 중 하나를 선택합니다.
5. [앞에 병합(Merge earlier)] 또는 [뒤에 병합(Merge later)]을 선택하여 계층의 순서를 조정합니다.
6. 저장을 선택합니다.

계층의 버전이 지정됩니다. 각 계층 버전의 콘텐츠는 변경할 수 없습니다. 계층의 소유자는 새 계층 버전을 릴리스하여 업데이트된 콘텐츠를 제공할 수 있습니다. 콘솔을 사용하여 함수에 연결된 계층 버전을 업데이트할 수 있습니다.

함수의 계층 버전 업데이트(콘솔)

1. Lambda 콘솔의 계층 페이지를 엽니다.
2. 버전을 업데이트하려는 계층을 선택합니다.
3. 이 버전을 사용하는 함수 탭을 선택합니다.
4. 수정하려는 함수를 선택하고 편집을 선택합니다.
5. 계층 버전에서 변경할 계층 버전을 선택합니다.
6. 함수 업데이트(Update functions)를 선택합니다.

여러 AWS 계정에 걸쳐 함수의 계층 버전을 업데이트할 수 없습니다.

주제

- 함수에서 계층 콘텐츠 액세스

- 계층 정보 찾기

함수에서 계층 콘텐츠 액세스

Lambda 함수에 계층이 포함되어 있는 경우 Lambda는 함수 실행 환경의 `/opt` 디렉터리에 계층 콘텐츠를 추출합니다. Lambda는 함수에 의해 나열된 순서(낮음에서 높음)로 계층을 추출합니다. Lambda는 이름이 같은 폴더를 병합합니다. 동일한 파일이 여러 계층에 나타나면 마지막으로 적용된 계층의 버전이 사용됩니다.

각 Lambda 런타임은 `PATH` 변수에 특정 `/opt` 디렉터리 폴더를 추가합니다. 함수 코드는 경로를 지정하지 않고도 계층 콘텐츠에 액세스할 수 있습니다. Lambda 실행 환경의 경로 설정에 관한 자세한 내용은 the section called “정의된 런타임 환경 변수” 섹션을 참조하세요.

계층을 생성할 때 라이브러리를 어디에 포함해야 하는지 알아보려면 the section called “각 Lambda 런타임에 대한 계층 경로” 섹션을 참조하세요.

Node.js 또는 Python 런타임을 사용하는 경우 Lambda 콘솔에서 기본 제공 코드 편집기를 사용할 수 있습니다. 현재 함수에 계층으로 추가한 모든 라이브러리를 가져올 수 있어야 합니다.

계층 정보 찾기

계정에서 함수의 런타임과 호환되는 레이어를 찾으려면 ListLayersAPI를 사용하세요. 예를 들어, 다음 `list-layers` AWS Command Line Interface(CLI) 명령을 사용할 수 있습니다.

```
aws lambda list-layers --compatible-runtime python3.9
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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",
                ]
            }
        }
    ]
}
```

계정의 모든 계층을 나열하려면 `--compatible-runtime` 옵션을 생략합니다. 응답 세부 정보에는 각 계층의 최신 버전이 표시됩니다.

ListLayerVersionsAPI를 사용하여 최신 버전의 레이어를 가져올 수도 있습니다. 예를 들어, 다음 `list-layer-versions` CLI 명령을 사용할 수 있습니다.

```
aws lambda list-layer-versions --layer-name my-layer
```

다음과 유사한 출력 화면이 표시되어야 합니다.

```
{
    "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"
            ]
        }
    ]
```

```
}
```

계층으로 AWS CloudFormation 작업

AWS CloudFormation을 사용하여 계층을 생성하고 해당 계층을 Lambda 함수와 연결할 수 있습니다. 다음 예제 템플릿에서는 `my-lambda-layer`라는 계층을 만들고 계층 속성을 사용하여 Lambda 함수에 계층을 연결합니다.

```
---
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
```

계층으로 AWS SAM 작업

AWS Serverless Application Model(AWS SAM)을 사용하여 애플리케이션에서 계층 생성을 자동화할 수 있습니다. `AWS::Serverless::LayerVersion` 리소스 유형은 Lambda 함수 구성에서 참조할 수 있는 계층 버전을 생성합니다.

```
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 확장을 사용하여 Lambda 함수 보강

Lambda 익스텐션을 사용하여 Lambda 함수를 보강할 수 있습니다. 예를 들어 Lambda 익스텐션을 사용하여 원하는 모니터링, 관찰, 보안 및 거버넌스 도구와 함수를 통합할 수 있습니다. AWS Lambda 파트너가 제공하는 다양한 도구 중에서 선택하거나 자체적인 Lambda 익스텐션을 만들 수 있습니다.

Lambda는 외부 및 내부 익스텐션을 지원합니다. 외부 익스텐션은 실행 환경에서 독립 프로세스로 실행되며 함수 호출이 완전히 처리된 후에도 계속 실행됩니다. 익스텐션은 별도의 프로세스로 실행되므로 함수와 다른 언어로 작성할 수 있습니다. 모든 Lambda 런타임은 익스텐션을 지원합니다.

내부 익스텐션은 런타임 프로세스의 일부로 실행됩니다. 함수는 래퍼 스크립트 또는 `JAVA_TOOL_OPTIONS`와 같은 프로세스 내 메커니즘을 사용하여 내부 익스텐션에 액세스합니다. 자세한 내용은 런타임 환경 수정 섹션을 참조하세요.

Lambda 콘솔, AWS Command Line Interface(AWS CLI) 또는 AWS CloudFormation, AWS Serverless Application Model (AWS SAM), Terraform과 같은 IaC(Infrastructure as code) 서비스 및 도구를 사용하여 함수에 익스텐션을 추가할 수 있습니다.

익스텐션에서 소비한 실행 시간(1밀리초 단위)에 대해 요금이 청구됩니다. 자체 익스텐션을 설치하는 데는 비용이 들지 않습니다. 익스텐션의 요금에 대한 자세한 내용은 AWS Lambda 요금을 참조하십시오. 파트너 익스텐션에 대한 요금 정보는 해당 파트너의 웹 사이트를 참조하십시오. 공식 파트너 확장 프로그램 목록은 the section called “확장 파트너” 섹션을 참조하세요.

확장에 대한 자습서 및 Lambda 함수와 함께 사용하는 방법은 AWS Lambda 확장 워크숍을 참조하세요.

주제

실행 환경

Lambda는 안전하고 격리된 런타임 환경을 제공하는 실행 환경에서 함수를 호출합니다. 실행 환경은 함수를 실행하는 데 필요한 리소스를 관리하고 함수의 런타임 및 익스텐션에 대한 수명 주기 지원을 제공합니다.

실행 환경의 수명 주기에는 다음 단계가 포함됩니다.

- **`Init`**: 이 단계 중에 Lambda는 구성된 리소스로 실행 환경을 만들거나 고정 해제하고, 함수와 모든 계층의 코드를 다운로드하고, 모든 익스텐션을 초기화하고, 런타임을 초기화한 다음 함수의 초기화 코드(기본 핸들러 외부의 코드)를 실행합니다. `Init` 단계는 첫 번째 호출 중에 발생하거나, 프로비저닝된 동시성을 사용하도록 설정한 경우 함수 호출 전에 발생합니다.

  `Init` 단계는 세 가지 하위 단계(`Extension init`, `Runtime init`, `Function init`.)로 나뉩니다. 이러한 하위 단계는 함수 코드가 실행되기 전에 모든 익스텐션과 런타임이 자신의 설정 작업을 완료하도록 보장합니다.

  Lambda SnapStart가 활성화되면 함수 버전을 게시할 때 `Init` 단계가 발생합니다. Lambda는 초기화된 실행 환경의 메모리 및 디스크 상태 스냅샷을 저장하고 암호화된 스냅샷을 유지하며 짧은 지연 시간으로 액세스할 수 있도록 스냅샷을 캐싱합니다. `beforeCheckpoint` 런타임 후크가 있는 경우 코드는 `Init` 단계가 끝날 때 실행됩니다.
- **`Restore`**(SnapStart만 해당): SnapStart 함수를 처음 호출하고 함수가 스케일 업되면 Lambda는 이 함수를 처음부터 초기화하는 대신 유지된 스냅샷에서 새 실행 환경을 재개합니다. `afterRestore()` 런타임 후크가 있는 경우 코드는 `Restore` 단계가 끝날 때 실행됩니다. `afterRestore()` 런타임 후크 지속 시간에 대해 요금이 청구됩니다. 런타임(JVM)이 로드되고 `afterRestore()` 런타임 후크가 제한 시간(10초) 내에 완료되어야 합니다. 그렇지 않으면 SnapStartTimeoutException이 발생합니다. `Restore` 단계가 완료되면 Lambda가 함수 핸들러(호출 단계)를 간접적으로 호출합니다.
- **`Invoke`**: 이 단계에서 Lambda는 함수 핸들러를 호출합니다. 함수의 실행이 완료한 후 Lambda는 다른 함수 호출을 처리할 준비를 합니다.
- **`Shutdown`**: Lambda 함수가 일정 기간 동안 호출을 받지 않으면 이 단계가 트리거됩니다. `Shutdown` 단계에서 Lambda는 런타임을 종료하고 익스텐션이 완전히 중지되도록 알림을 보낸 다음 환경을 제거합니다. Lambda는 각 익스텐션에 `Shutdown` 이벤트를 보냅니다. 이 이벤트는 환경이 곧 종료됨을 익스텐션에 알립니다.

Init 단계 중에 Lambda는 익스텐션을 포함한 계층을 실행 환경의 /opt 디렉터리로 추출합니다. Lambda는 /opt/extensions/ 디렉터리에서 익스텐션을 찾고, 익스텐션을 실행하기 위해 각 파일을 실행 가능한 부트스트랩으로 해석하고, 모든 익스텐션을 병행하여 시작합니다.

성능 및 리소스에 미치는 영향

함수의 익스텐션 크기는 배포 패키지 크기 제한에 포함됩니다. .zip 파일 아카이브의 경우 함수 및 모든 익스텐션의 압축 해제된 총 크기는 압축 해제된 배포 패키지 크기 제한인 250MB를 초과할 수 없습니다.

익스텐션은 CPU, 메모리, 스토리지와 같은 함수 리소스를 공유하므로 함수의 성능에 영향을 줄 수 있습니다. 예를 들어 익스텐션이 컴퓨팅 집약적 작업을 수행하는 경우 함수의 실행 시간이 늘어날 수 있습니다.

각 익스텐션은 Lambda가 함수를 호출하기 전에 초기화를 완료해야 합니다. 따라서 상당한 초기화 시간을 소비하는 익스텐션은 함수 호출의 지연 시간을 증가시킬 수 있습니다.

함수 실행 후 익스텐션에 소요되는 추가 시간을 측정하려면 PostRuntimeExtensionsDuration 함수 지표를 사용하고, 사용된 메모리 증가를 측정하려면 MaxMemoryUsed 지표를 사용합니다. 특정 익스텐션의 영향을 파악하려는 경우 서로 다른 버전의 함수를 나란히 실행하면 됩니다.

권한

익스텐션은 함수와 동일한 리소스에 액세스할 수 있습니다. 익스텐션은 함수와 동일한 환경 내에서 실행되므로 함수와 익스텐션 간에 권한이 공유됩니다.

.zip 파일 아카이브의 경우 AWS CloudFormation 템플릿을 생성하여 AWS Identity and Access Management(IAM) 권한을 포함한 동일한 익스텐션 구성을 여러 함수에 연결하는 작업을 간소화할 수 있습니다.

Lambda 확장 구성

익스텐션 구성(.zip 파일 아카이브)

익스텐션을 함수에 Lambda 계층으로 추가할 수 있습니다. 계층을 사용하면 조직 내에서 또는 전체 Lambda 개발자 커뮤니티에서 익스텐션 프로그램을 공유할 수 있습니다. 계층에 하나 이상의 익스텐션을 추가할 수 있습니다. 각 함수에 최대 10개의 익스텐션을 등록할 수 있습니다.

계층에 대해 사용하는 것과 동일한 방법으로 함수에 익스텐션을 추가합니다. 자세한 설명은 Lambda 계층 섹션을 참조하세요.

함수에 익스텐션 프로그램 추가(콘솔)

1. Lambda 콘솔의 함수 페이지를 엽니다.
2. 함수를 선택합니다.
3. 아직 선택되지 않은 경우 코드(Code) 탭을 선택합니다.
4. 계층(Layers)에서 편집(Edit)을 선택합니다.
5. 계층 선택(Choose a layer)에서 ARN 지정(Specify an ARN)을 선택합니다.
6. ARN 지정(Specify an ARN)에서 익스텐션 계층의 Amazon 리소스 이름(ARN)을 입력합니다.
7. 추가를 선택합니다.

컨테이너 이미지에서 익스텐션 사용

컨테이너 이미지에 익스텐션을 추가할 수 있습니다. ENTRYPOINT 컨테이너 이미지 설정은 함수의 주 프로세스를 지정합니다. Dockerfile에서 ENTRYPOINT 설정을 구성하거나 함수 구성에서 재정의로 구성합니다.

컨테이너 내에서 여러 프로세스를 실행할 수 있습니다. Lambda는 주 프로세스와 모든 추가 프로세스의 수명 주기를 관리합니다. Lambda는 익스텐션 API를 사용하여 익스텐션 수명 주기를 관리합니다.

예: 외부 익스텐션 추가

외부 익스텐션은 Lambda 함수와 별도의 프로세스에서 실행됩니다. Lambda는 `/opt/extensions/` 디렉터리의 각 익스텐션에 대해 프로세스를 시작합니다. Lambda는 익스텐션 API를 사용하여 익스텐션 수명 주기를 관리합니다. 함수가 완료될 때까지 실행된 후 Lambda는 각 외부 익스텐션에 `Shutdown` 이벤트를 보냅니다.

Example Python 기본 이미지에 외부 익스텐션 추가

```
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
```

다음 단계

익스텐션에 대해 자세히 알아보려면 다음 리소스를 사용하는 것이 좋습니다.

- 기본적인 작업 예는 AWS Lambda 컴퓨팅 블로그에서 AWS용 익스텐션 빌드를 참조하세요.
- AWS Lambda 파트너가 제공하는 익스텐션에 대한 자세한 내용은 AWS Lambda 컴퓨팅 블로그에서 AWS 익스텐션 소개를 참조하세요.
- 사용 가능한 예제 확장 및 래퍼 스크립트를 보려면 AWS Samples GitHub 저장소의 AWS Lambda확장을 참조하십시오.

AWS Lambda 확장 파트너

AWS Lambda는 여러 서드 파티 엔터티와 협력하여 Lambda 함수와 통합할 확장을 제공합니다. 다음 목록에서는 언제든 사용할 수 있는 서드 파티 확장 프로그램에 대해 자세히 설명합니다.

- AppDynamics - Node.js 또는 Python Lambda 함수를 자동으로 계측하여 함수 성능에 대한 가시성 및 경고를 제공합니다.
- Check Point CloudGuard - 서버리스 애플리케이션에 대한 전체 수명 주기 보안을 제공하는 확장 기반 런타임 솔루션입니다.
- Datadog - 메트릭, 추적 및 로그를 사용하여 서버리스 애플리케이션에 대한 종합적인 실시간 가시성을 제공합니다.
- Dynatrace - 추적 및 지표에 대한 가시성을 제공하고 AI를 활용하여 전체 애플리케이션 스택에서 자동화된 오류 감지 및 근본 원인 분석을 수행합니다.
- Elastic - 상관 관계가 있는 추적, 지표 및 로그를 사용하여 근본 원인 문제를 식별하고 해결하는 애플리케이션 성능 모니터링(APM)을 제공합니다.
- Epsagon - 호출 이벤트를 수신하고 추적을 저장한 다음, Lambda 함수 실행에 병렬로 전송합니다.
- Fastly - 인젝션 방식 공격, 자격 증명 스터핑을 통한 계정 탈취, 악성 봇, API 남용과 같은 의심스러운 활동으로부터 Lambda 함수를 보호합니다.
- HashiCorp Vault - 비밀 정보를 관리하고 개발자가 Vault가 인식하는 기능을 만들지 않고도 함수 코드 내에서 사용할 수 있도록 합니다.
- Honeycomb - 앱 스택 디버깅을 위한 가시성 도구입니다.
- Lumigo - Lambda 함수 호출을 프로파일링하고 지표를 수집하여 서버리스 및 마이크로서비스 환경에서 문제를 해결합니다.
- New Relic - Lambda 함수와 함께 실행하여 원격 분석 정보를 자동으로 수집 및 향상하고 New Relic의 통합 가시성 플랫폼으로 전송합니다.
- Sedai - AI/ML을 기반으로 하는 자율 운영 관리 플랫폼으로, 클라우드 운영 팀이 클라우드 비용 절감, 성능 및 대규모 가용성을 극대화할 수 있도록 지속적인 최적화를 제공합니다.
- Sentry - Lambda 함수의 성능을 진단, 수정 및 최적화합니다.
- Site24x7 - Lambda 환경에 대한 실시간 가시성을 달성합니다
- Splunk - Lambda 함수의 효율적이고 효과적인 모니터링을 위해 지연 시간이 짧은 고해상도 지표를 수집합니다.
- Sumo Logic - 서버리스 애플리케이션의 상태 및 성능에 대한 가시성을 제공합니다.
- Thundra - 추적, 지표 및 로그 등에 대한 비동기 원격 분석 보고를 제공합니다.

- 솔트 보안 - 다양한 런타임에 대한 자동 설정 및 지원을 통해 Lambda 함수의 API 상태 거버넌스와 API 보안을 간소화합니다.

AWS 관리형 확장

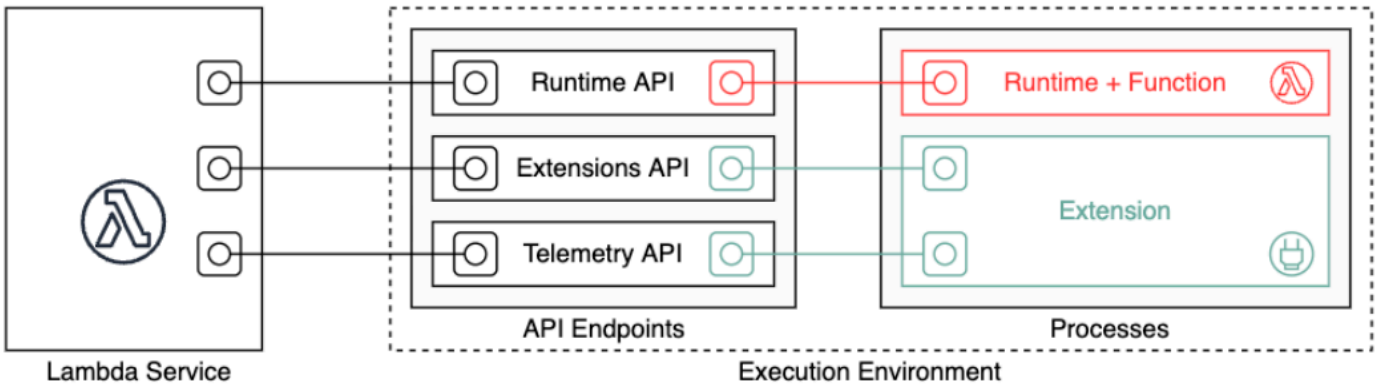
AWS는 다음과 같은 자체 관리형 확장을 제공합니다.

- AWS AppConfig - 기능 플래그와 동적 데이터를 사용하여 Lambda 함수를 업데이트합니다. 이 확장을 사용하여 운영 제한 및 튜닝과 같은 다른 동적 구성을 업데이트할 수도 있습니다.
- Amazon CodeGuru Profiler - 애플리케이션에서 가장 비싼 코드 라인을 정확히 파악하고 코드 개선을 위한 권장 사항을 제공하여 애플리케이션 성능을 개선하고 비용을 절감합니다.
- CloudWatch Lambda Insights - 자동화된 대시보드를 통해 Lambda 함수의 성능을 모니터링하고, 문제를 해결하고, 최적화합니다.
- AWS Distro for OpenTelemetry(ADOT) - 함수를 활성화하여 AWS X-Ray 등의 AWS 모니터링 서비스와 Honeycomb 및 Lightstep 등의 OpenTelemetry 지원 대상에 추적 데이터를 전송합니다.
- AWS 파라미터 및 보안 암호 — 고객이 AWS Systems Manager 파라미터 스토어에서 파라미터를, AWS Secrets Manager에서 보안 암호를 안전하게 검색할 수 있도록 합니다.

추가 확장 샘플 및 데모 프로젝트는 AWS Lambda 확장 섹션을 참조하세요.

Lambda 확장 API를 사용하여 확장 생성

Lambda 함수 작성자는 익스텐션을 사용하여 모니터링, 관찰, 보안 및 거버넌스를 위해 선호하는 도구와 Lambda를 통합합니다. 함수 작성자는 AWS, AWS 파트너 및 오픈 소스 프로젝트의 익스텐션을 사용할 수 있습니다. 익스텐션 사용에 대한 자세한 내용은 AWS Lambda 컴퓨팅 블로그의 AWS 익스텐션 소개를 참조하세요. 이 섹션에서는 Lambda 확장 API, Lambda 실행 환경 수명 주기 및 Lambda 확장 API 참조를 사용하는 방법을 설명합니다.



익스텐션 작성자는 Lambda 익스텐션 API를 사용하여 Lambda 실행 환경과 완전히 통합할 수 있습니다. 익스텐션은 함수 및 실행 환경 수명 주기 이벤트에 등록할 수 있습니다. 이러한 이벤트에 대한 응답으로 새 프로세스를 시작하고, 논리를 실행하고, Lambda 수명 주기의 모든 단계(초기화, 호출 및 종료)를 관리하고 참여할 수 있습니다. 또한 런타임 로그 API를 사용하여 로그 스트림을 수신할 수 있습니다.

익스텐션은 실행 환경에서 독립 프로세스로 실행되며 함수 호출이 완전히 처리된 후에도 계속 실행됩니다. 익스텐션은 프로세스로 실행되므로 함수와 다른 언어로 작성할 수 있습니다. 컴파일된 언어를 사용하여 익스텐션을 구현하는 것이 좋습니다. 이 경우 익스텐션은 지원되는 런타임과 호환되는 독립형 바이너리입니다. 모든 Lambda 런타임은 익스텐션을 지원합니다. 컴파일되지 않은 언어를 사용하는 경우 호환되는 런타임을 익스텐션에 포함해야 합니다.

Lambda는 내부 익스텐션도 지원합니다. 내부 익스텐션은 런타임 프로세스에서 별도의 스레드로 실행됩니다. 런타임은 내부 익스텐션을 시작하고 중지합니다. Lambda 환경과 통합하는 다른 방법은 언어별 환경 변수 및 래퍼 스크립트를 사용하는 것입니다. 이 방법을 통해 런타임 환경을 구성하고 런타임 프로세스의 시작 동작을 수정할 수 있습니다.

두 가지 방법으로 익스텐션을 함수에 추가할 수 있습니다. .zip 파일 아카이브로 배포된 함수의 경우 익스텐션을 계층으로 배포합니다. 컨테이너 이미지로 정의된 함수의 경우 익스텐션을 컨테이너 이미지에 추가합니다.

> **Note**
>
> 예제 익스텐션 및 래퍼 스크립트에 대해서는 AWS Lambda 샘플 GitHub 리포지토리에서 AWS 익스텐션을 참조하세요.

주제

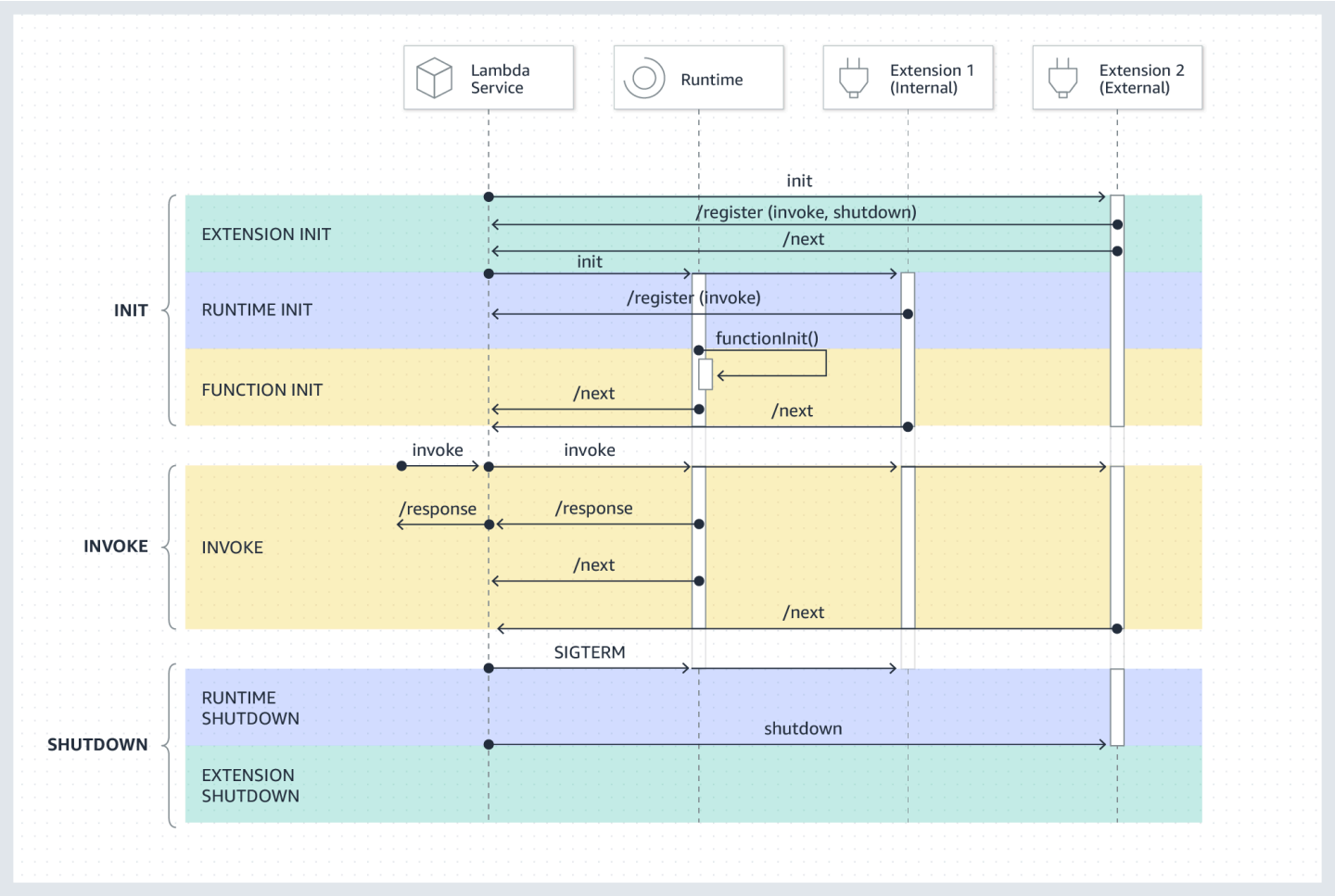
Lambda 실행 환경 수명 주기

실행 환경의 수명 주기에는 다음 단계가 포함됩니다.

- **`Init`**: 이 단계 중에 Lambda는 구성된 리소스로 실행 환경을 만들거나 고정 해제하고, 함수와 모든 계층의 코드를 다운로드하고, 모든 익스텐션을 초기화하고, 런타임을 초기화한 다음 함수의 초기화 코드(기본 핸들러 외부의 코드)를 실행합니다. `Init` 단계는 첫 번째 호출 중에 발생하거나, 프로비저닝된 동시성을 사용하도록 설정한 경우 함수 호출 전에 발생합니다.

  `Init` 단계는 세 가지 하위 단계(`Extension init`, `Runtime init`, `Function init`.)로 나뉩니다. 이러한 하위 단계는 함수 코드가 실행되기 전에 모든 익스텐션과 런타임이 자신의 설정 작업을 완료하도록 보장합니다.
- **`Invoke`**: 이 단계에서 Lambda는 함수 핸들러를 호출합니다. 함수의 실행이 완료한 후 Lambda는 다른 함수 호출을 처리할 준비를 합니다.
- **`Shutdown`**: Lambda 함수가 일정 기간 동안 호출을 받지 않으면 이 단계가 트리거됩니다. `Shutdown` 단계에서 Lambda는 런타임을 종료하고 익스텐션이 완전히 중지되도록 알림을 보낸 다음 환경을 제거합니다. Lambda는 각 익스텐션에 `Shutdown` 이벤트를 보냅니다. 이 이벤트는 환경이 곧 종료됨을 익스텐션에 알립니다.

각 단계는 Lambda에서 런타임 및 등록된 모든 익스텐션에 전송하는 이벤트로 시작됩니다. 런타임 및 각 익스텐션이 `Next` API 요청을 전송하여 완료 신호를 보냅니다. 각 프로세스가 완료되고 대기 중인 이벤트가 없으면 Lambda는 실행 환경을 중지합니다.



주제

- 초기화 단계
- 호출 단계
- 종료 단계
- 권한 및 구성
- 장애 처리
- 익스텐션 문제 해결

초기화 단계

`Extension init` 단계 중에 이벤트를 수신하려면 각 익스텐션은 Lambda에 등록해야 합니다. Lambda는 익스텐션의 전체 파일 이름을 사용하여 익스텐션이 부트스트랩 시퀀스를 완료했는지 확인

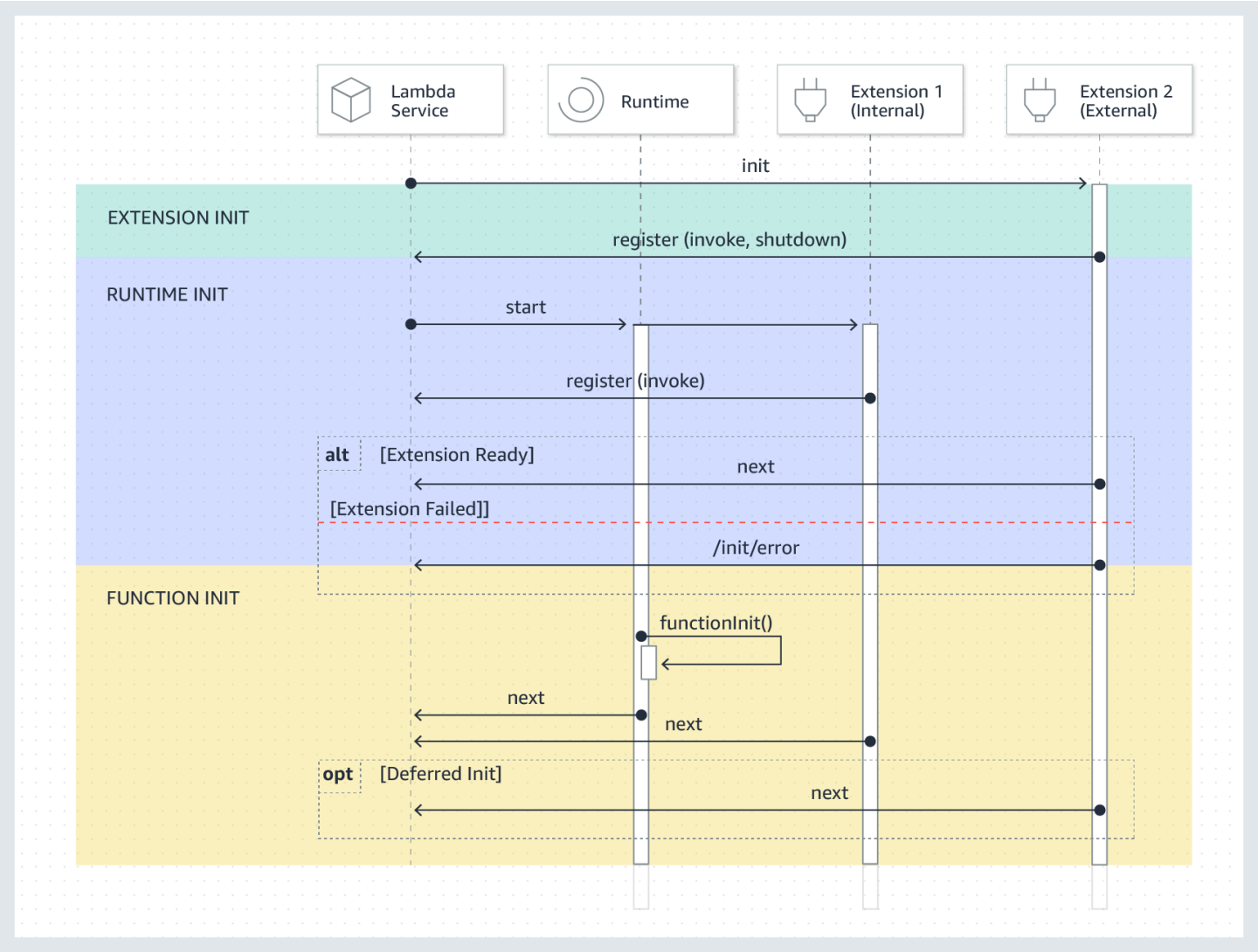
합니다. 따라서 각 `Register` API 호출에는 익스텐션의 전체 파일 이름이 있는 `Lambda-Extension-Name` 헤더를 포함해야 합니다.

각 함수에 최대 10개의 익스텐션을 등록할 수 있습니다. 이 제한은 `Register` API 호출을 통해 적용됩니다.

각 익스텐션이 등록되면 Lambda는 `Runtime init` 단계를 시작합니다. 런타임 프로세스는 `functionInit`를 호출하여 `Function init` 단계를 시작합니다.

`Init` 단계는 런타임 및 등록된 각 익스텐션이 `Next` API 요청을 전송하여 완료를 나타낸 후에 완료됩니다.

> **Note**
>
> 익스텐션은 `Init` 단계의 어느 지점에서나 초기화를 완료할 수 있습니다.



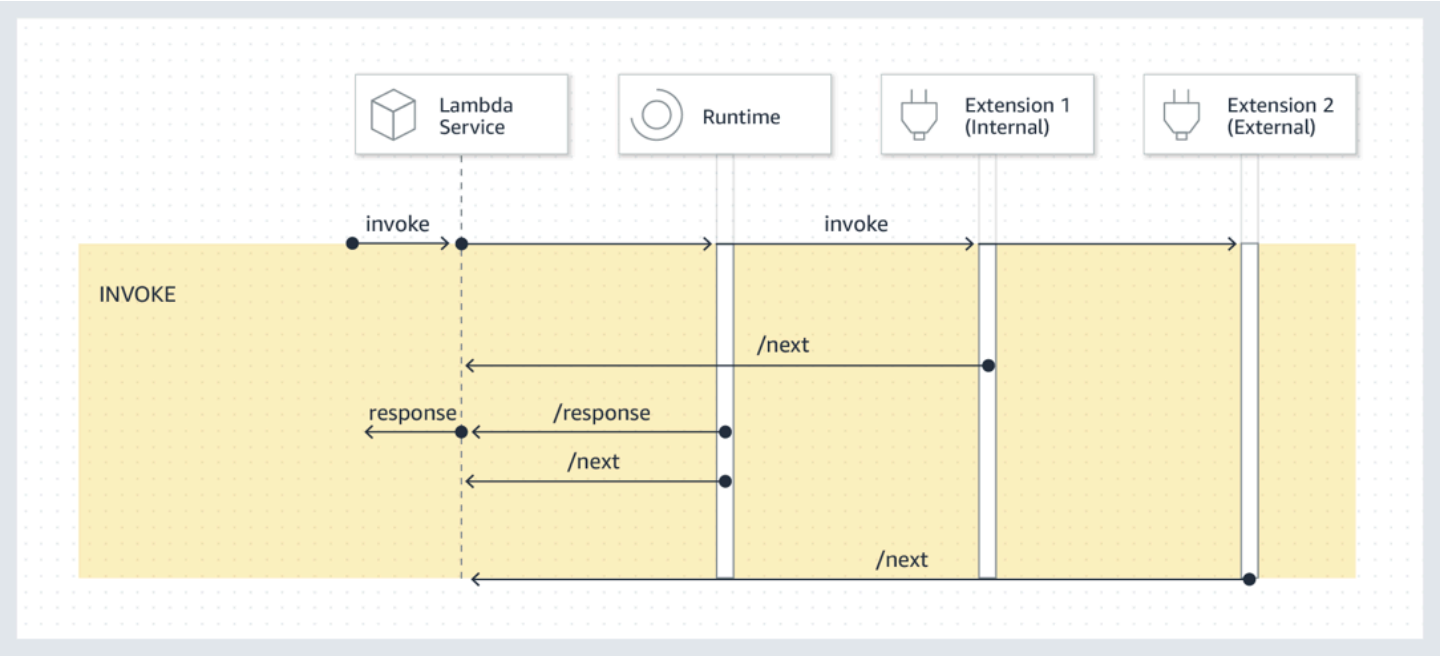
호출 단계

Next API 요청에 대한 응답으로 Lambda 함수가 호출되면 Lambda는 런타임과 Invoke 이벤트에 등록된 각 익스텐션에 Invoke 이벤트를 전송합니다.

호출하는 동안 외부 익스텐션은 함수와 동시에 실행됩니다. 또한 함수가 완료된 후에도 외부 익스텐션이 계속 실행됩니다. 따라서 진단 정보를 캡처하거나 로그, 지표 및 추적 정보를 선택한 위치로 보낼 수 있습니다.

런타임에서 함수 응답을 수신한 후 Lambda는 익스텐션이 여전히 실행 중이더라도 클라이언트에 대한 응답을 반환합니다.

Invoke 단계는 런타임 및 모든 익스텐션이 Next API 요청을 전송하여 완료되었음을 알린 후에 종료됩니다.



이벤트 페이로드: 런타임(및 Lambda 함수)으로 전송되는 이벤트에는 전체 요청, 헤더(예: RequestId) 및 페이로드가 포함됩니다. 각 익스텐션으로 전송되는 이벤트에는 이벤트 내용을 설명하는 메타데이터가 포함됩니다. 이 수명 주기 이벤트에는 이벤트 유형, 함수가 시간 초과되는 시간(deadlineMs), requestId, 호출된 함수의 Amazon 리소스 이름(ARN) 및 추적 헤더가 포함됩니다.

함수 이벤트 본문에 액세스하려는 익스텐션은 익스텐션과 통신하는 런타임 내 SDK를 사용할 수 있습니다. 함수 개발자는 런타임 내 SDK를 사용하여 함수가 호출될 때 익스텐션에 페이로드를 전송합니다.

다음은 페이로드의 예입니다.

```
{
    "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"
    }
 }
```

기간 제한: 함수의 제한 시간 설정은 전체 Invoke 단계의 기간을 제한합니다. 예를 들어 함수 제한 시간을 360초로 설정하면 함수와 모든 익스텐션이 360초 내에 완료되어야 합니다. 독립적인 호출 후 단

계는 없습니다. 기간은 런타임 및 모든 확장의 호출이 완료되는 데 걸리는 총 시간이며 함수와 모든 확장의 실행이 완료될 때까지 계산되지 않습니다.

성능 영향 및 익스텐션 오버헤드: 익스텐션은 함수 성능에 영향을 줄 수 있습니다. 익스텐션 작성자는 익스텐션이 성능에 미치는 영향을 제어할 수 있습니다. 예를 들어 익스텐션이 컴퓨팅 집약적 작업을 수행하는 경우 함수 소요 시간이 늘어납니다. 익스텐션과 함수 코드가 동일한 CPU 리소스를 공유하기 때문입니다. 또한 함수 호출이 완료된 후 익스텐션에서 대규모 연산을 수행하는 경우 모든 익스텐션이 완료되었음을 알릴 때까지 `Invoke` 단계가 계속되므로 함수 소요 시간이 늘어납니다.

> **Note**
>
> Lambda는 함수의 메모리 설정에 비례하여 CPU 용량을 할당합니다. 함수 및 익스텐션 프로세스가 동일한 CPU 리소스를 놓고 경쟁하기 때문에 낮은 메모리 설정에서 실행 및 초기화 기간이 늘어날 수 있습니다. 실행 및 초기화 기간을 줄이려면 메모리 설정을 늘려보세요.

`Invoke` 단계에서 익스텐션에 의한 성능 영향을 식별할 수 있도록 Lambda는 `PostRuntimeExtensionsDuration` 지표를 출력합니다. 이 지표는 런타임 `Next` API 요청과 마지막 익스텐션 `Next` API 요청 사이의 누적 시간을 측정합니다. 사용된 메모리 증가를 측정하려면 `MaxMemoryUsed` 지표를 사용합니다. 함수 지표에 대한 자세한 내용은 Lambda 함수 지표 작업 섹션을 참조하세요.

함수 개발자는 다양한 버전의 함수를 나란히 실행하여 특정 익스텐션의 영향을 파악할 수 있습니다. 익스텐션 작성자는 함수 개발자가 적합한 익스텐션을 쉽게 선택할 수 있도록 예상되는 리소스 소비를 게시하는 것이 좋습니다.

종료 단계

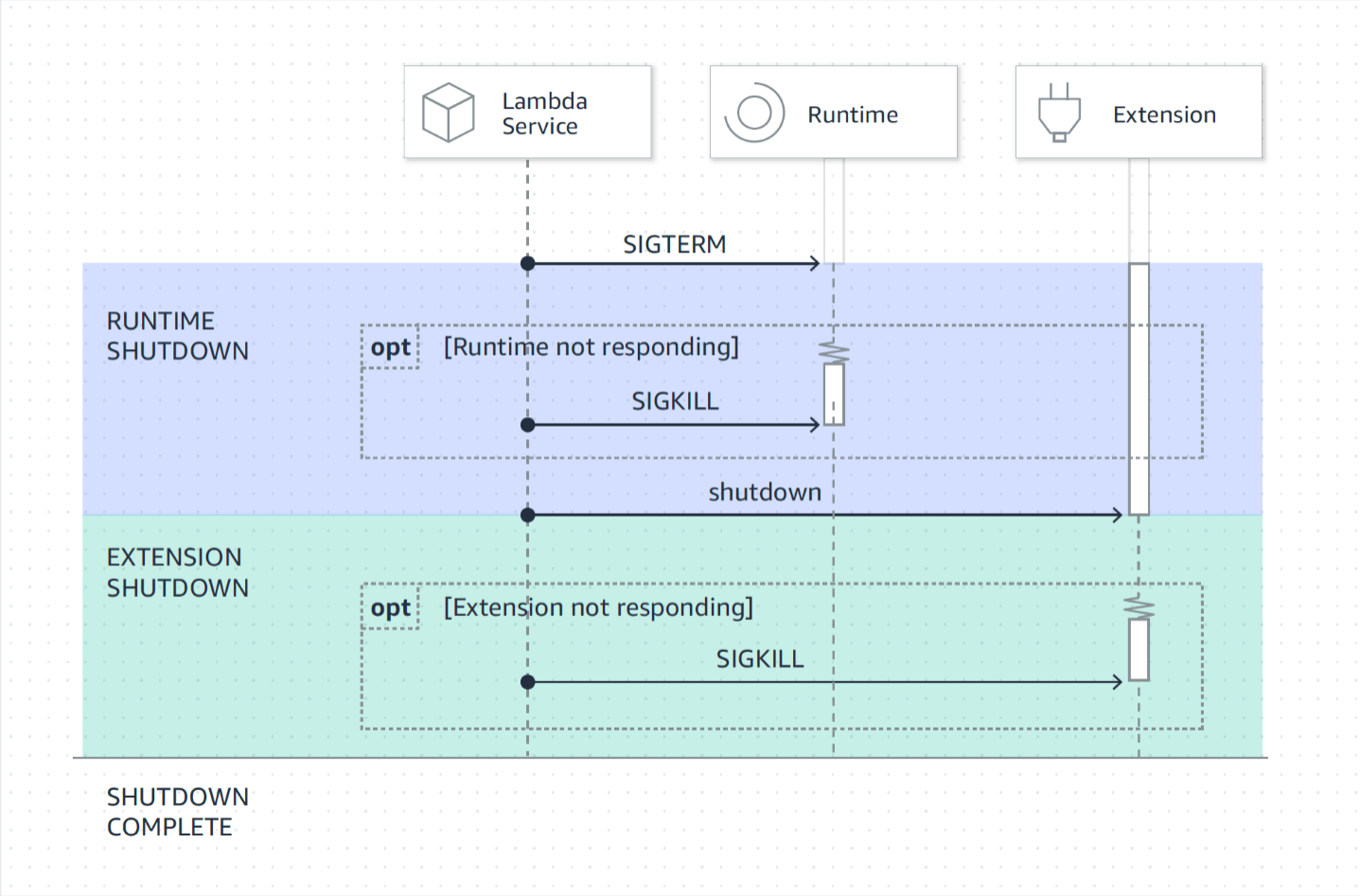
Lambda는 런타임을 종료하려고 할 때 `Shutdown`을 등록된 각 외부 익스텐션에 전송합니다. 익스텐션은 이 시간 동안 최종 정리 작업을 수행할 수 있습니다. `Shutdown` 이벤트는 `Next` API 요청에 대한 응답으로 전송됩니다.

기간 제한: `Shutdown` 단계의 최대 기간은 등록된 익스텐션 구성에 따라 다릅니다.

- 0밀리초 - 등록된 익스텐션이 없는 함수
- 500밀리초 - 등록된 내부 익스텐션이 하나 있는 함수
- 2,000밀리초 - 등록된 외부 익스텐션이 하나 이상 있는 함수

함수에 외부 익스텐션이 있는 경우 Lambda는 런타임 프로세스가 정상 종료를 수행하도록 최대 300밀리초(내부 익스텐션이 있는 런타임의 경우 500밀리초)를 예약합니다. Lambda는 외부 익스텐션이 종료되도록 2,000밀리초 제한에서 남은 시간을 할당합니다.

런타임 또는 익스텐션이 제한 시간 내에 `Shutdown` 이벤트에 응답하지 않는 경우 Lambda는 `SIGKILL` 신호를 사용하여 프로세스를 종료합니다.



이벤트 페이로드: `Shutdown` 이벤트에는 종료 이유와 남은 시간(밀리초)이 포함됩니다.

`shutdownReason`에는 다음과 같은 값이 포함됩니다.

- SPINDOWN - 정상 종료
- TIMEOUT - 기간 제한 초과
- FAILURE - 오류 상태(예: `out-of-memory` 이벤트)

```
{
```

```
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

권한 및 구성

익스텐션은 Lambda 함수와 동일한 실행 환경에서 실행됩니다. 익스텐션은 CPU, 메모리, /tmp 디스크 스토리지와 같은 리소스를 함수와 공유합니다. 또한 익스텐션은 함수와 동일한 AWS Identity and Access Management(IAM) 역할 및 보안 컨텍스트를 사용합니다.

파일 시스템 및 네트워크 액세스 권한: 익스텐션은 함수 런타임과 동일한 파일 시스템 및 네트워크 이름 네임스페이스에서 실행됩니다. 따라서 익스텐션은 해당 운영 체제와 호환되어야 합니다. 익스텐션에 추가 아웃바운드 네트워크 트래픽 규칙이 필요한 경우 이러한 규칙을 함수 구성에 적용해야 합니다.

> **Note**
>
> 함수 코드 디렉터리는 읽기 전용이므로 익스텐션에서 함수 코드를 수정할 수 없습니다.

환경 변수: 익스텐션은 런타임 프로세스와 관련된 다음 변수를 제외하고 함수의 환경 변수에 액세스할 수 있습니다.

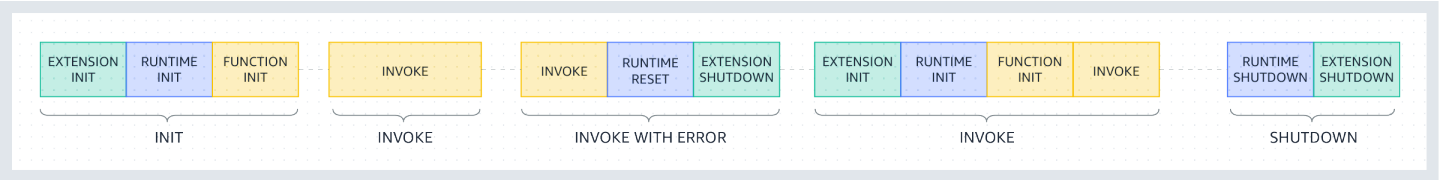
- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR
- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

장애 처리

초기화 오류: 익스텐션에 오류가 발생하면 Lambda는 실행 환경을 다시 시작하여 일관된 동작을 적용하고 익스텐션에 대해 빠른 실패(fast fail)를 장려합니다. 또한 일부 고객의 경우 익스텐션이 로깅, 보안, 거버넌스 및 텔레메트리 수집과 같은 미션 크리티컬한 요구 사항을 충족해야 합니다.

호출 오류(예: 메모리 부족, 함수 시간 초과): 익스텐션은 런타임과 리소스를 공유하므로 메모리 부족의 영향을 받습니다. 런타임에 오류가 발생하면 모든 익스텐션과 런타임 자체가 Shutdown 단계에 참여합니다. 또한 런타임은 현재 호출의 일부로 자동으로 다시 시작되거나 지연된 재초기화 메커니즘을 통해 다시 시작됩니다.

함수 시간 초과 또는 런타임 오류와 같은 오류가 발생하면 Invoke 시 Lambda 서비스가 재설정을 수행합니다. 재설정은 Shutdown 이벤트처럼 동작합니다. 먼저 Lambda는 런타임을 종료한 다음 등록된 각 외부 익스텐션에 Shutdown 이벤트를 전송합니다. 이벤트에는 종료 이유가 포함됩니다. 이 환경이 새 호출에 사용되는 경우 익스텐션 및 런타임은 다음 호출의 일부로 다시 초기화됩니다.



이전 다이어그램에 대한 자세한 설명은 호출 단계 중 실패을 참조하십시오.

익스텐션 로그: Lambda가 익스텐션의 로그 출력을 CloudWatch Logs에 보냅니다. Lambda는 Init 시각 익스텐션에 대한 추가 로그 이벤트도 생성합니다. 로그 이벤트에는 성공 시 이름 및 등록 기본 설정(이벤트, 구성)과 실패 시 실패 이유가 기록됩니다.

익스텐션 문제 해결

- Register 요청이 실패할 경우 Lambda-Extension-Name API 호출의 Register 헤더에 익스텐션의 전체 파일 이름이 포함되어 있는지 확인합니다.
- 내부 익스텐션에 대한 Register 요청이 실패할 경우 요청이 Shutdown 이벤트에 등록되지 않았는지 확인합니다.

익스텐션 API 참조

익스텐션 API 버전 2020-01-01에 대한 OpenAPI 사양은 extensions-api.zip에서 사용할 수 있습니다.

AWS_LAMBDA_RUNTIME_API 환경 변수에서 API 엔드포인트의 값을 검색할 수 있습니다. Register 요청을 보내려면 각 API 경로 앞에 접두사 2020-01-01/을 사용합니다. 예:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API 메서드

- 등록
- Next
- 초기화 오류
- 종료 오류

등록

Extension init 중에 이벤트를 수신하려면 모든 익스텐션은 Lambda에 등록해야 합니다. Lambda는 익스텐션의 전체 파일 이름을 사용하여 익스텐션이 부트스트랩 시퀀스를 완료했는지 확인합니다. 따라서 각 Register API 호출에는 익스텐션의 전체 파일 이름이 있는 Lambda-Extension-Name 헤더를 포함해야 합니다.

내부 익스텐션은 런타임 프로세스에 의해 시작 및 중지되므로 Shutdown 이벤트에 등록할 수 없습니다.

경로 – /extension/register

메서드 – POST

요청 헤더

- Lambda-Extension-Name - 익스텐션의 전체 파일 이름입니다. 필수 항목 여부: 예. 유형: 문자열.
- Lambda-Extension-Accept-Feature - 이 옵션을 사용하여 등록 시 선택적 확장 기능을 지정할 수 있습니다. 필수 항목 여부: 아니요 유형: 쉼표로 구분된 문자열입니다. 이 설정을 사용하여 지정할 수 있는 기능은 다음과 같습니다.
  - accountId - 지정된 경우 확장 등록 응답에는 확장을 등록하려는 Lambda 함수와 연결된 계정 ID가 포함됩니다.

요청 본문 파라미터

- events - 등록할 이벤트의 배열입니다. 필수 항목 여부: 아니요 유형: 문자열 배열 유효한 문자열: INVOKE, SHUTDOWN.

응답 헤더

- `Lambda-Extension-Identifier` - 모든 후속 요청에 필요한, 생성된 고유 에이전트 식별자(UUID 문자열)입니다.

응답 코드

- 200 - 응답 본문에는 함수 이름, 함수 버전 및 핸들러 이름이 포함됩니다.
- 400 - 잘못된 요청
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 익스텐션을 즉시 종료해야 합니다.

Example 요청 본문의 예

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example 응답 본문의 예

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example 선택적 accountId 기능이 포함된 응답 본문의 예

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Next

익스텐션은 Next API 요청을 전송하여 다음 이벤트(Invoke 이벤트 또는 Shutdown 이벤트가 될 수 있음)를 수신합니다. 응답 본문에는 이벤트 데이터가 포함된 JSON 문서인 페이로드가 포함되어 있습니다.

익스텐션은 Next API 요청을 전송하여 새 이벤트를 수신할 준비가 되었음을 알립니다. 이는 차단 호출입니다.

반환할 이벤트가 발생할 때까지 일정 기간 동안 익스텐션이 일시 중단될 수 있으므로 GET 호출에 시간 제한을 설정하지 마세요.

경로 – /extension/event/next

매서드 – GET

요청 헤더

- Lambda-Extension-Identifier - 익스텐션의 고유 식별자(UUID 문자열). 필수 항목 여부: 예. 유형: UUID 문자열.

응답 헤더

- Lambda-Extension-Event-Identifier - 이벤트의 고유 식별자(UUID 문자열)입니다.

응답 코드

- 200 - 응답에는 다음 이벤트(EventInvoke 또는 EventShutdown)에 대한 정보가 포함됩니다.
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 익스텐션을 즉시 종료해야 합니다.

초기화 오류

익스텐션은 이 메서드를 사용하여 Lambda에 초기화 오류를 보고합니다. 익스텐션이 등록된 후 초기화에 실패할 경우 이 메서드를 호출합니다. Lambda가 오류를 수신하면 더 이상 API 호출이 성공하지 못합니다. 익스텐션은 Lambda로부터 응답을 수신한 후에 종료되어야 합니다.

경로 – /extension/init/error

메서드 – POST

요청 헤더

- `Lambda-Extension-Identifier` - 익스텐션의 고유 식별자. 필수 항목 여부: 예. 유형: UUID 문자열.
- `Lambda-Extension-Function-Error-Type` – 익스텐션에서 발생한 오류 유형입니다. 필수 항목 여부: 예. 이 헤더는 문자열 값으로 구성됩니다. Lambda는 모든 문자열을 허용하지만 <category.reason> 형식을 사용하는 것이 좋습니다. 예:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

요청 본문 파라미터

- `ErrorRequest` – 오류에 대한 정보입니다. 필수 항목 여부: 아니요

이 필드는 다음과 같은 구조의 JSON 객체입니다.

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

Lambda는 `errorType`에 대한 모든 값을 허용합니다.

다음 예제에서는 Lambda 함수가 호출에 제공된 이벤트 데이터를 구문 분석할 수 없는 함수 오류 메시지를 보여 줍니다.

Example 함수 오류

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

응답 코드

- 202 - 수락됨
- 400 - 잘못된 요청
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 익스텐션을 즉시 종료해야 합니다.

종료 오류

익스텐션은 종료 전에 이 메서드를 사용하여 Lambda에 오류를 보고합니다. 예기치 않은 오류가 발생하면 이 메서드를 호출합니다. Lambda가 오류를 수신하면 더 이상 API 호출이 성공하지 못합니다. 익스텐션은 Lambda로부터 응답을 수신한 후에 종료되어야 합니다.

경로 – `/extension/exit/error`

메서드 – POST

요청 헤더

- `Lambda-Extension-Identifier` - 익스텐션의 고유 식별자. 필수 항목 여부: 예. 유형: UUID 문자열.
- `Lambda-Extension-Function-Error-Type` – 익스텐션에서 발생한 오류 유형입니다. 필수 항목 여부: 예. 이 헤더는 문자열 값으로 구성됩니다. Lambda는 모든 문자열을 허용하지만 <category.reason> 형식을 사용하는 것이 좋습니다. 예:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

요청 본문 파라미터

- `ErrorRequest` – 오류에 대한 정보입니다. 필수 항목 여부: 아니요

이 필드는 다음과 같은 구조의 JSON 객체입니다.

```
{
      errorMessage: string (text description of the error),
```

```
      errorType: string,
      stackTrace: array of strings
}
```

Lambda는 errorType에 대한 모든 값을 허용합니다.

다음 예제에서는 Lambda 함수가 호출에 제공된 이벤트 데이터를 구문 분석할 수 없는 함수 오류 메시지를 보여 줍니다.

Example 함수 오류

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

응답 코드

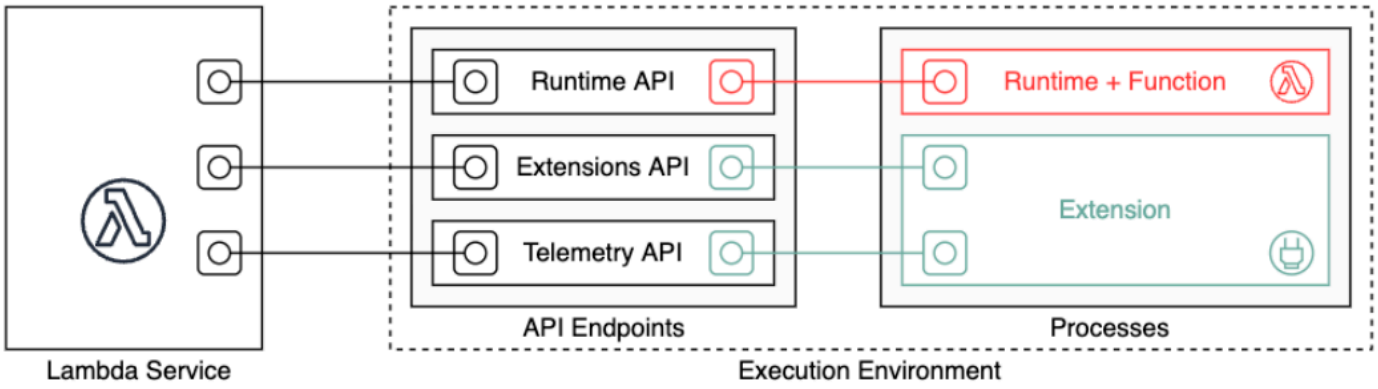
- 202 - 수락됨
- 400 - 잘못된 요청
- 403 - 금지됨
- 500 - 컨테이너 오류. 복구 불능 상태입니다. 익스텐션을 즉시 종료해야 합니다.

Lambda 텔레메트리 API

텔레메트리 API를 사용하면 확장에서 Lambda로부터 텔레메트리 데이터를 직접 수신할 수 있습니다. 함수 초기화 및 간접 호출 중에 Lambda는 로그, 플랫폼 지표, 플랫폼 트레이스와 같은 텔레메트리를 자동으로 캡처합니다. 텔레메트리 API를 사용하면 확장에서 Lambda로부터 거의 실시간으로 이 텔레메트리 데이터에 직접 액세스할 수 있습니다.

Lambda 실행 환경 내에서 텔레메트리 스트림에 대한 Lambda 확장을 구독할 수 있습니다. 구독 후 Lambda는 모든 텔레메트리 데이터를 확장으로 자동 전송합니다. 그런 다음 유연하게 데이터를 처리 및 필터링하고 Amazon Simple Storage Service(S3) 버킷 또는 타사 관찰성 도구 제공업체와 같은 원하는 대상으로 전송할 수 있습니다.

다음 다이어그램은 확장 API와 텔레메트리 API가 실행 환경 내에서 확장을 Lambda에 연결하는 방법을 보여줍니다. 또한 런타임 API는 런타임 및 함수를 Lambda에 연결합니다.



> **Important**
>
> Lambda 텔레메트리 API는 Lambda 로그 API를 대체합니다. 로그 API도 계속 정상적으로 작동하지만, 앞으로는 텔레메트리 API만 사용하는 것이 좋습니다. 텔레메트리 API 또는 로그 API를 사용하여 확장에서 텔레메트리 스트림을 구독할 수 있습니다. 이러한 API 중 하나를 사용하여 구독한 후 다른 API를 사용하여 구독하려고 하면 오류가 반환됩니다.

확장은 텔레메트리 API를 사용하여 세 가지 텔레메트리 스트림을 구독할 수 있습니다.

- 플랫폼 텔레메트리 - 실행 환경 런타임 수명 주기, 확장 수명 주기 및 함수 호출과 관련한 이벤트와 오류를 설명하는 로그, 지표 및 트레이스입니다.
- 함수 로그 - Lambda 함수 코드가 생성하는 사용자 지정 로그입니다.

- 확장 로그 - Lambda 확장 코드가 생성하는 사용자 지정 로그입니다.

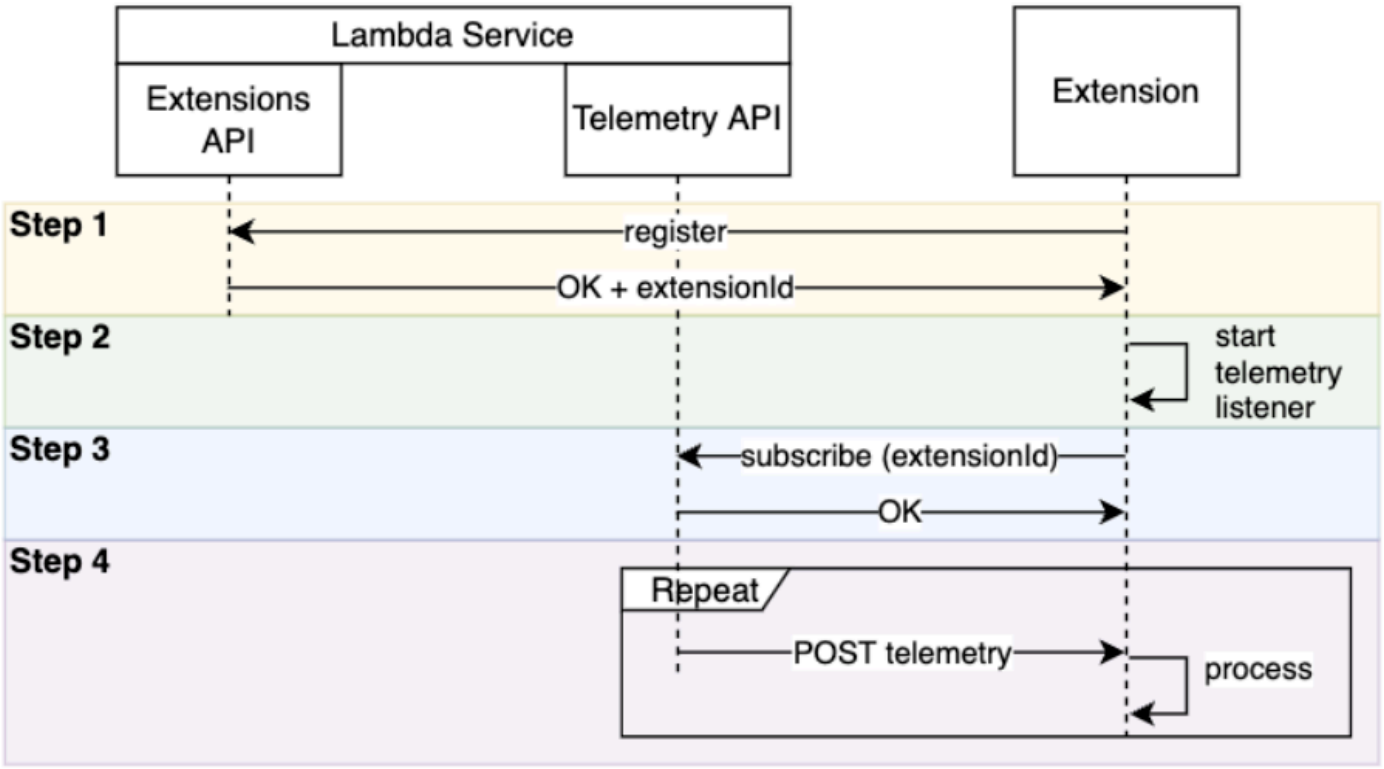
> **Note**
>
> 확장 프로그램이 원격 분석 스트림을 CloudWatch 구독하는 경우에도 Lambda는 로그와 지표를 X-Ray로 보내고 추적합니다 (추적을 활성화한 경우).

Sections

텔레메트리 API를 사용하여 확장 생성

Lambda 확장은 실행 환경에서 독립 프로세스로 실행됩니다. 함수 간접 호출이 완료된 후에도 확장을 계속 실행할 수 있습니다. 화장은 별도의 프로세스로 실행되므로 함수 코드와 다른 언어로 작성할 수 있습니다. Golang 또는 Rust와 같은 컴파일된 언어를 사용하여 확장을 작성하는 것이 좋습니다. 이 경우 확장은 지원되는 모든 런타임과 호환되는 독립형 바이너리입니다.

다음 다이어그램은 텔레메트리 API를 사용하여 텔레메트리 데이터를 수신하고 처리하는 확장을 생성하는 4단계 프로세스를 보여줍니다.



다음은 각 단계에 대한 자세한 내용입니다.

1. the section called “익스텐션 API”를 사용하여 확장을 등록합니다. 이렇게 하면 다음 단계에서 필요한 `Lambda-Extension-Identifier`가 제공됩니다. 확장을 등록하는 자세한 방법은 the section called “확장 등록” 섹션을 참조하세요.
2. 텔레메트리 리스너를 생성합니다. 기본 HTTP 또는 TCP 서버일 수 있습니다. Lambda는 텔레메트리 리스너의 URI를 사용하여 텔레메트리 데이터를 확장에 전송합니다. 자세한 설명은 the section called “텔레메트리 리스너 생성” 섹션을 참조하세요.
3. 텔레메트리 API에서 구독 API를 사용하여 원하는 텔레메트리 스트림에 대한 확장을 구독합니다. 이 단계에는 텔레메트리 리스너의 URI가 필요합니다. 자세한 설명은 the section called “텔레메트리 API에 구독 요청 전송” 섹션을 참조하세요.
4. 텔레메트리 리스너를 통해 Lambda에서 텔레메트리 데이터를 가져옵니다. Amazon S3나 외부 관측성 서비스로 데이터를 디스패치하는 등 이 데이터에 대한 모든 사용자 지정 처리 작업을 수행할 수 있습니다.

> **Note**
>
> Lambda 함수의 실행 환경은 수명 주기의 일부로 여러 번 시작 및 중지할 수 있습니다. 일반적으로 확장 코드는 함수 호출 중에 실행되며 종료 단계에서도 최대 2초간 실행됩니다. 텔레메트리가 리스너에 도착하면 일괄 처리하는 것이 좋습니다. 그런 다음, `Invoke` 및 `Shutdown` 수명 주기 이벤트를 사용하여 각 배치를 원하는 대상으로 전송합니다.

확장 등록

텔레메트리 데이터를 구독하려면 먼저 Lambda 확장을 등록해야 합니다. 등록은 확장 초기화 단계에서 이루어집니다. 다음 예제에서는 확장을 등록하는 HTTP 요청을 보여줍니다.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

요청이 성공하면 구독자는 HTTP 200 성공 응답을 받습니다. 응답 헤더에는 `Lambda-Extension-Identifier`가 포함되어 있습니다. 응답 본문에는 함수의 다른 속성이 포함되어 있습니다.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

자세한 내용은 the section called “익스텐션 API 참조” 섹션을 참조하십시오.

텔레메트리 리스너 생성

Lambda 확장에는 텔레메트리 API로부터 수신되는 요청을 처리하는 리스너가 있어야 합니다. 다음 코드는 Golang의 텔레메트리 리스너 구현 예제를 보여줍니다.

```
// 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
```

```
}
```

대상 프로토콜 지정

텔레메트리를 수신하기 위해 텔레메트리 API를 사용하여 구독하는 경우, 대상 URI 외에 대상 프로토콜도 지정할 수 있습니다.

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda는 텔레메트리를 수신하는 데 두 가지 프로토콜을 허용합니다.

- HTTP(권장) – Lambda가 텔레메트리를 로컬 HTTP 엔드포인트(`http://sandbox.localdomain:${PORT}/${PATH}`)에 JSON 형식의 레코드 배열로 전달합니다. `$PATH` 파라미터는 선택 항목입니다. Lambda는 HTTP만 지원하고 HTTPS는 지원하지 않습니다. Lambda는 POST 요청을 통해 텔레메트리를 제공합니다.
- TCP – Lambda가 텔레메트리를 새 줄 구분 JSON(NDJSON) 형식으로 TCP 포트에 전달합니다.

> **Note**
>
> TCP 대신 HTTP를 사용하는 것이 좋습니다. TCP를 사용하면 Lambda 플랫폼에서는 텔레메트리를 애플리케이션 계층에 전송할 때 인식할 수 없습니다. 따라서 확장이 작동 중지될 경우 텔레메트리가 손실될 수 있습니다. HTTP에는 이러한 제한이 없습니다.

텔레메트리 수신을 위해 구독하기 전에 로컬 HTTP 리스너 또는 TCP 포트를 설정합니다. 설치하는 동안 다음 사항에 유의하세요.

- Lambda는 실행 환경 내에 있는 대상에만 텔레메트리를 보냅니다.
- Lambda는 리스너가 없거나 POST 요청에서 오류가 발생하는 경우 텔레메트리 전송을 다시 시도합니다(백오프 포함). 텔레메트리 리스너에서 충돌이 발생하는 경우 Lambda가 실행 환경을 다시 시작한 후 텔레메트리를 계속 수신합니다.
- Lambda는 포트 9001을 예약합니다. 다른 포트 번호 제한이나 권장 사항은 없습니다.

메모리 사용량 및 버퍼링 구성

실행 환경에서 메모리 사용량은 구독자 수에 따라 선형적으로 증가합니다. 각 구독은 새 메모리 버퍼를 열어 텔레메트리 데이터를 저장하기 때문에 메모리 리소스를 사용합니다. 버퍼 메모리 사용량은 실행 환경의 전체 메모리 소비에 포함됩니다.

텔레메트리 API를 사용하여 텔레메트리를 수신하기 위해 구독하면 텔레메트리 데이터를 버퍼링하여 구독자에게 배치로 전달하는 옵션이 제공됩니다. 메모리 사용량을 최적화하기 위해 버퍼링 구성을 지정할 수 있습니다.

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

버퍼링 구성 설정

파라미터	설명	기본값 및 한도
`maxBytes`	메모리에 버퍼링할 텔레메트리의 최대 볼륨(바이트)입니다.	기본값: 26만 2,144 최소: 26만 2,144 최대: 1,04만 8,576
`maxItems`	메모리에 버퍼링할 최대 이벤트 수입니다.	기본값: 1만 최소: 1,000 최대값: 10,000
`timeoutMs`	배치를 버퍼링할 최대 시간(밀리초)입니다.	기본값: 1,000 최소: 25 최대: 3만

버퍼링을 설정할 때 다음 사항을 고려합니다.

- 입력 스트림이 닫히면 Lambda가 로그를 플러시합니다. 예를 들어 런타임에서 충돌이 발생한 경우 이 상황이 나타날 수 있습니다.
- 각 구독자는 구독 요청에서 버퍼링 구성을 사용자 지정할 수 있습니다.
- 데이터를 읽기 위한 버퍼 크기를 결정할 때 수신 페이로드를 최대 `2 * maxBytes + metadataBytes`로 예상합니다. 여기서, `maxBytes`는 버퍼링 설정의 구성 요소입니다. 고려할 `metadataBytes`의 크기를 측정하려면 다음 메타데이터를 검토합니다. Lambda는 다음과 유사한 메타데이터를 각 레코드에 추가합니다.

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- 구독자가 들어오는 텔레메트리를 충분히 빠르게 처리할 수 없거나 함수 코드가 매우 높은 로그 볼륨을 생성하는 경우 Lambda는 메모리 사용률을 범위 내로 유지하기 위해 레코드를 삭제할 수 있습니다. 이 경우 Lambda는 `platform.logsDropped` 이벤트를 전송합니다.

텔레메트리 API에 구독 요청 전송

Lambda 확장은 텔레메트리 API에 구독 요청을 전송하여 텔레메트리 데이터를 수신하도록 구독할 수 있습니다. 이 구독 요청에는 확장이 구독하도록 하려는 이벤트 유형에 대한 정보가 포함되어야 합니다. 또한 이 요청에는 전송 대상 정보와 버퍼링 구성이 포함될 수 있습니다.

구독 요청을 전송하려면 먼저 확장 ID(`Lambda-Extension-Identifier`)가 있어야 합니다. 확장 API를 사용하여 확장을 등록할 때 API 응답에서 확장 ID를 얻습니다.

구독은 확장 초기화 단계에서 이루어집니다. 다음 예에서는 플랫폼 텔레메트리, 함수 로그, 확장 로그의 세 가지 텔레메트리 스트림을 모두 구독하는 HTTP 요청을 보여줍니다.

```
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"
    }
}
```

요청이 성공하면 구독자는 HTTP 200 성공 응답을 받습니다.

```
HTTP/1.1 200 OK
"OK"
```

인바운드 텔레메트리 API 메시지

텔레메트리 API를 사용하여 구독하고 나면 확장이 POST 요청을 통해 Lambda로부터 텔레메트리를 자동으로 수신하기 시작합니다. 각 POST 요청 본문에는 Event 객체 배열이 포함되어 있습니다. 각 Event에는 다음 스키마가 있습니다.

```
{
   time: String,
   type: String,
   record: Object
}
```

- time 속성은 Lambda 플랫폼에서 이벤트를 생성한 시점을 정의합니다. 이벤트가 실제로 발생한 시점과는 다릅니다. time의 문자열 값은 ISO 8601 형식의 타임스탬프입니다.
- type 속성은 이벤트 유형을 정의합니다. 다음 표에서는 모든 가능한 값을 설명합니다.
- record 속성은 텔레메트리 데이터가 포함된 JSON 객체를 정의합니다. 이 JSON 객체의 스키마는 type에 따라 달라집니다.

다음 표에는 모든 유형의 Event 객체가 요약되어 있으며 각 이벤트 유형의 텔레메트리 APIEvent 스키마 참조에 대한 링크가 나와 있습니다.

텔레메트리 API 메시지 유형

범주	이벤트 유형	설명	이벤트 레코드 스키마
플랫폼 이벤트	`platform.initStart`	함수 초기화가 시작되었습니다.	the section called "`platform.initStart`" 스키마
플랫폼 이벤트	`platform.initRuntimeDone`	함수 초기화가 완료되었습니다.	the section called "`platform.initRuntimeDone`" 스키마
플랫폼 이벤트	`platform.initReport`	함수 초기화 보고서입니다.	the section called "`platform.initReport`" 스키마
플랫폼 이벤트	`platform.start`	함수 호출이 시작되었습니다.	the section called "`platform.start`" 스키마
플랫폼 이벤트	`platform.runtimeDone`	런타임이 이벤트 처리를 성공 또는 실패로 완료했습니다.	the section called "`platform.runtimeDone`" 스키마
플랫폼 이벤트	`platform.report`	함수 호출 보고서입니다.	the section called "`platform.report`" 스키마
플랫폼 이벤트	`platform.restoreStart`	런타임 복원이 시작되었습니다.	the section called "`platform.restoreStart`" 스키마

범주	이벤트 유형	설명	이벤트 레코드 스키마
플랫폼 이벤트	`platform.restoreRuntimeDone`	런타임 복원이 완료되었습니다.	the section called “`platform.restoreRuntimeDone`” 스키마
플랫폼 이벤트	`platform.restoreReport`	런타임 복원 보고서.	the section called “`platform.restoreReport`” 스키마
플랫폼 이벤트	`platform.telemetrySubscription`	텔레메트리 API를 구독하는 확장입니다.	the section called “`platform.telemetrySubscription`” 스키마
플랫폼 이벤트	`platform.logsDropped`	Lambda가 로그 항목을 삭제했습니다.	the section called “`platform.logsDropped`” 스키마
함수 로그	`function`	함수 코드의 로그 줄입니다.	the section called “`function`” 스키마
익스텐션 로그	`extension`	확장 코드의 로그 줄입니다.	the section called “`extension`” 스키마

Lambda 텔레메트리 API 참조

Lambda 텔레메트리 API 엔드포인트를 사용하여 텔레메트리 스트림에 대한 확장을 구독할 수 있습니다. `AWS_LAMBDA_RUNTIME_API` 환경 변수에서 텔레메트리 API 엔드포인트를 검색할 수 있습니다. API 요청을 보내려면 API 버전(`2022-07-01/`) 및 `telemetry/`를 추가합니다. 예:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

구독 응답 버전 `2022-12-13`의 OpenAPI 사양(OAS) 정의는 다음을 참조하세요.

- HTTP — telemetry-api-http-schema.zip
- TCP — .zip telemetry-api-tcp-schema

API 작업

- Subscribe

Subscribe

Lambda 확장은 텔레메트리 스트림을 구독하기 위해 구독 API 요청을 보낼 수 있습니다.

- 경로 – `/telemetry`
- 메서드 – `PUT`
- 헤더
  - `Content-Type: application/json`
- 요청 본문 파라미터
  - schemaVersion
    - 필수 항목 여부: 예
    - 유형: String
    - 유효한 값: `"2022-12-13"`또는 `"2022-07-01"`
  - 대상 - 텔레메트리 이벤트 대상과 이벤트 전달을 위한 프로토콜을 정의하는 구성 설정입니다.
    - 필수 항목 여부: 예
    - 유형: 객체

```
{
    "protocol": "HTTP",
```

```
    "URI": "http://sandbox.localdomain:8080"
}
```

- 프로토콜 - Lambda가 텔레메트리 데이터를 전송하는 데 사용하는 프로토콜입니다.
  - 필수 항목 여부: 예
  - 유형: String
  - 유효한 값: "HTTP"|"TCP"
- URI - 텔레메트리 데이터를 전송할 URI입니다.
  - 필수 항목 여부: 예
  - 유형: String
- 자세한 설명은 the section called “대상 프로토콜 지정” 섹션을 참조하세요.

- 유형 - 확장에서 구독하려는 텔레메트리 유형입니다.
  - 필수 항목 여부: 예
  - 유형: 문자열 어레이
  - 유효한 값: "platform"|"function"|"extension"
- 버퍼링 - 이벤트 버퍼링을 위한 구성 설정입니다.
  - 필수 항목 여부: 아니요
  - 유형: 객체

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – 메모리에 버퍼링할 최대 이벤트 수입니다.
    - 필수 항목 여부: 아니요
    - 유형: 정수
    - 기본값: 1,000
    - 최소: 1,000
    - 최대값: 10,000
  - maxBytes – 메모리에 버퍼링할 텔레메트리의 최대 볼륨(바이트)입니다.

    - 필수 항목 여부: 아니요
    - 유형: 정수
    - 기본값: 26만 2,144
    - 최소: 26만 2,144
    - 최대: 104만 8,576
- timeoutMs – 배치를 버퍼링할 최대 시간(밀리초)입니다.
    - 필수 항목 여부: 아니요
    - 유형: 정수
    - 기본값: 1,000
    - 최소: 25
    - 최대: 3만
- 자세한 설명은 the section called “메모리 사용량 및 버퍼링 구성” 섹션을 참조하세요.

구독 API 요청의 예

```
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"
   }
}
```

구독 요청이 성공하면 이 확장은 HTTP 200 성공 응답을 받습니다.

```
HTTP/1.1 200 OK
```

```
"OK"
```

구독 요청이 실패하면 이 확장은 오류 응답을 받습니다. 예:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

확장이 수신할 수 있는 몇 가지 추가 응답 코드는 다음과 같습니다.

- 200 - 요청이 성공적으로 완료되었습니다.
- 202 - 요청이 수락되었습니다. 로컬 테스트 환경의 구독 요청 응답
- 400 - 잘못된 요청
- 500 - 서비스 오류

Lambda 텔레메트리 API **Event** 스키마 참조

Lambda 텔레메트리 API 엔드포인트를 사용하여 텔레메트리 스트림에 대한 확장을 구독할 수 있습니다. `AWS_LAMBDA_RUNTIME_API` 환경 변수에서 텔레메트리 API 엔드포인트를 검색할 수 있습니다. API 요청을 보내려면 API 버전(`2022-07-01/`) 및 `telemetry/`를 추가합니다. 예:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

구독 응답 버전 `2022-12-13`의 OpenAPI 사양(OAS) 정의는 다음을 참조하세요.

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

다음 표에는 텔레메트리 API가 지원하는 모든 유형의 `Event` 객체가 요약되어 있습니다.

텔레메트리 API 메시지 유형

범주	이벤트 유형	설명	이벤트 레코드 스키마
플랫폼 이벤트	`platform.initStart`	함수 초기화가 시작되었습니다.	the section called “`platform.initStart`” 스키마
플랫폼 이벤트	`platform.initRuntimeDone`	함수 초기화가 완료되었습니다.	the section called “`platform.initRuntimeDone`” 스키마
플랫폼 이벤트	`platform.initReport`	함수 초기화 보고서입니다.	the section called “`platform.initReport`” 스키마
플랫폼 이벤트	`platform.start`	함수 호출이 시작되었습니다.	the section called “`platform.start`” 스키마

범주	이벤트 유형	설명	이벤트 레코드 스키마
플랫폼 이벤트	platform.runtimeDone	런타임이 이벤트 처리를 성공 또는 실패로 완료했습니다.	the section called “platform.runtimeDone” 스키마
플랫폼 이벤트	platform.report	함수 호출 보고서입니다.	the section called “platform.report” 스키마
플랫폼 이벤트	platform.restoreStart	런타임 복원이 시작되었습니다.	the section called “platform.restoreStart” 스키마
플랫폼 이벤트	platform.restoreRuntimeDone	런타임 복원이 완료되었습니다.	the section called “platform.restoreRuntimeDone” 스키마
플랫폼 이벤트	platform.restoreReport	런타임 복원 보고서.	the section called “platform.restoreReport” 스키마
플랫폼 이벤트	platform.telemetrySubscription	텔레메트리 API를 구독하는 확장입니다.	the section called “platform.telemetrySubscription” 스키마
플랫폼 이벤트	platform.logsDropped	Lambda가 로그 항목을 삭제했습니다.	the section called “platform.logsDropped” 스키마

범주	이벤트 유형	설명	이벤트 레코드 스키마
함수 로그	`function`	함수 코드의 로그 줄입니다.	the section called “`function`” 스키마
익스텐션 로그	`extension`	확장 코드의 로그 줄입니다.	the section called “`extension `” 스키마

목차

- Span
- Status
- TraceContext
- TracingType

텔레메트리 API **Event** 객체 유형

이 섹션에서는 Lambda 텔레메트리 API가 지원하는 Event 객체 유형을 자세히 설명합니다. 이벤트 설명에서 물음표(?)는 해당 속성이 객체에 없을 수 있음을 나타냅니다.

platform.initStart

platform.initStart 이벤트는 함수 초기화 단계가 시작되었음을 나타냅니다.
platform.initStart Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

PlatformInitStart 객체에는 다음 속성이 있습니다.

- functionName - String
- functionVersion - String
- initializationType – the section called “InitType” 객체
- instanceId? - String
- instanceMaxMemory? - Integer
- phase – the section called “InitPhase” 객체
- runtimeVersion? – String
- runtimeVersionArn? – String

다음은 platform.initStart 유형의 예제 Event입니다.

```
{
    "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 이벤트는 함수 초기화 단계가 완료되었음을 나타냅니다. platform.initRuntimeDone Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

PlatformInitRuntimeDone 객체에는 다음 속성이 있습니다.

- initializationType – the section called “InitType” 객체
- phase – the section called “InitPhase” 객체
- status – the section called “Status” 객체
- spans? - the section called “Span” 객체의 목록

다음은 platform.initRuntimeDone 유형의 예제 Event입니다.

```
{
    "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 이벤트에는 함수 초기화 단계의 전체 보고서가 포함됩니다. platform.initReport Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

PlatformInitReport 객체에는 다음 속성이 있습니다.

- errorType? - string
- initializationType – the section called “InitType” 객체
- phase – the section called “InitPhase” 객체
- metrics – the section called “InitReportMetrics” 객체
- spans? - the section called “Span” 객체의 목록
- status – the section called “Status” 객체

다음은 platform.initReport 유형의 예제 Event입니다.

```
{
    "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 이벤트는 함수 호출 단계가 시작되었음을 나타냅니다. platform.start Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

PlatformStart 객체에는 다음 속성이 있습니다.

- requestId – String
- version? – String
- tracing? – the section called “TraceContext”

다음은 platform.start 유형의 예제 Event입니다.

```
{
    "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 이벤트는 함수 호출 단계가 완료되었음을 나타냅니다. platform.runtimeDone Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

PlatformRuntimeDone 객체에는 다음 속성이 있습니다.

- errorType? – String
- metrics? – the section called “RuntimeDoneMetrics” 객체
- requestId – String
- status – the section called “Status” 객체
- spans? - the section called “Span” 객체의 목록
- tracing? – the section called “TraceContext” 객체

다음은 platform.runtimeDone 유형의 예제 Event입니다.

```
{
    "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 이벤트에는 함수 초기화 단계의 전체 보고서가 포함됩니다. platform.report Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

PlatformReport 객체에는 다음 속성이 있습니다.

- metrics – the section called “ReportMetrics” 객체
- requestId – String
- spans? - the section called “Span” 객체의 목록
- status – the section called “Status” 객체
- tracing? – the section called “TraceContext” 객체

다음은 platform.report 유형의 예제 Event입니다.

```
{
    "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 이벤트는 함수 환경 복원 이벤트가 시작되었음을 나타냅니다. 환경 복원 이벤트에서 Lambda는 처음부터 초기화하지 않고 캐싱된 스냅샷에서 환경을 생성합니다. 자세한 내용은 Lambda SnapStart 단원을 참조하십시오. platform.restoreStart Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

PlatformRestoreStart 객체에는 다음 속성이 있습니다.

- functionName - String
- functionVersion - String
- instanceId? - String
- instanceMaxMemory? - String
- runtimeVersion? – String
- runtimeVersionArn? – String

다음은 platform.restoreStart 유형의 예제 Event입니다.

```
{
    "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 이벤트는 함수 환경 복원 이벤트가 완료되었음을 나타냅니다. 환경 복원 이벤트에서 Lambda는 처음부터 초기화하지 않고 캐싱된 스냅샷에서 환경을 생성합니다. 자세한 내용은 Lambda SnapStart 단원을 참조하십시오. platform.restoreRuntimeDone Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

PlatformRestoreRuntimeDone 객체에는 다음 속성이 있습니다.

- errorType? – String
- spans? - the section called “Span” 객체의 목록
- status – the section called “Status” 객체

다음은 platform.restoreRuntimeDone 유형의 예제 Event입니다.

```
{
    "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 이벤트에는 함수 복원 이벤트의 전체 보고서가 포함됩니다. platform.restoreReport Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

PlatformRestoreReport 객체에는 다음 속성이 있습니다.

- errorType? - string
- metrics? – the section called “RestoreReportMetrics” 객체
- spans? - the section called “Span” 객체의 목록
- status – the section called “Status” 객체

다음은 platform.restoreReport 유형의 예제 Event입니다.

```
{
    "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 이벤트에는 확장 코드의 로그가 포함됩니다. extension Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = extension
```

```
- record: {}
```

PlatformExtension 객체에는 다음 속성이 있습니다.

- events – String의 목록
- name – String
- state – String

다음은 platform.extension 유형의 예제 Event입니다.

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

platform.telemetrySubscription

platform.telemetrySubscription 이벤트에는 확장 구독에 대한 정보가 포함됩니다. platform.telemetrySubscription Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

PlatformTelemetrySubscription 객체에는 다음 속성이 있습니다.

- name – String
- state – String
- types – String의 목록

다음은 platform.telemetrySubscription 유형의 예제 Event입니다.

```
{
```

```
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

platform.logsDropped 이벤트에는 삭제된 이벤트에 대한 정보가 포함됩니다. 함수의 로그 출력 속도가 높아 Lambda에서 처리할 수 없는 경우에도 Lambda는 platform.logsDropped 이벤트를 발생시킵니다. Lambda에서 함수의 로그 생성 속도에 맞춰 CloudWatch 또는 텔레메트리 API가 구독한 확장으로 로그를 전송할 수 없는 경우 로그를 삭제하여 함수 실행 속도가 느려지는 것을 방지합니다. platform.logsDropped Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

PlatformLogsDropped 객체에는 다음 속성이 있습니다.

- droppedBytes – Integer
- droppedRecords – Integer
- reason – String

다음은 platform.logsDropped 유형의 예제 Event입니다.

```
{
    "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 이벤트에는 함수 코드의 로그가 포함됩니다. function Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = function
- record: {}
```

record 필드 형식은 함수의 로그 형식이 일반 텍스트 형식인지 JSON 형식인지에 따라 달라집니다. 로그 형식 구성 옵션에 대한 자세한 내용은 the section called “JSON 및 일반 텍스트 로그 형식 구성”를 참조하십시오.

다음은 로그 형식이 일반 텍스트인 Event 예 function 유형입니다.

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

다음은 로그 형식이 JSON인 Event 예 function 유형입니다.

```
{
    "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**
>
> 사용 중인 스키마 버전이 2022-12-13 버전보다 이전인 경우 함수의 로깅 형식이 JSON으로 구성된 경우에도 "record"는 항상 문자열로 렌더링됩니다.

extension

extension 이벤트에는 확장 코드의 로그가 포함됩니다. extension Event 객체의 형식은 다음과 같습니다.

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

record 필드 형식은 함수의 로그 형식이 일반 텍스트 형식인지 JSON 형식인지에 따라 달라집니다. 로그 형식 구성 옵션에 대한 자세한 내용은 the section called “JSON 및 일반 텍스트 로그 형식 구성”를 참조하십시오.

다음은 로그 형식이 일반 텍스트인 Event 예 extension 유형입니다.

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

다음은 로그 형식이 JSON인 Event 예 extension 유형입니다.

```
{
    "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**
>
> 사용 중인 스키마 버전이 2022-12-13 버전보다 이전인 경우 함수의 로깅 형식이 JSON으로 구성된 경우에도 "record"는 항상 문자열로 렌더링됩니다.

공유 객체 유형

이 섹션에서는 Lambda 텔레메트리 API가 지원하는 공유 객체 유형을 자세히 설명합니다.

InitPhase

초기화 작업이 발생하는 단계를 설명하는 문자열 열거형입니다. 대부분의 경우 Lambda는 `init` 단계 중에 함수 초기화 코드를 실행합니다. 하지만 오류가 발생하면 Lambda가 `invoke` 단계 중에 함수 초기화 코드를 다시 실행할 수 있습니다. (이를 억제된 초기화라고 함)

- 유형 – `String`
- 유효한 값 – `init|invoke|snap-start`

InitReportMetrics

초기화 단계에 대한 지표를 포함하는 객체입니다.

- 유형 – `Object`

`InitReportMetrics` 객체의 형식은 다음과 같습니다.

```
InitReportMetrics: Object
- durationMs: Double
```

다음은 `InitReportMetrics` 객체의 예입니다.

```
{
    "durationMs": 247.88
}
```

InitType

Lambda가 환경을 초기화한 방법을 설명하는 문자열 열거형입니다.

- 유형 – `String`
- 유효한 값 – `on-demand|provisioned-concurrency`

ReportMetrics

완료된 단계에 대한 지표가 포함된 객체입니다.

- 유형 – Object

ReportMetrics 객체의 형식은 다음과 같습니다.

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

다음은 ReportMetrics 객체의 예입니다.

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

RestoreReportMetrics

완료된 복원 단계에 대한 지표가 포함된 객체입니다.

- 유형 – Object

RestoreReportMetrics 객체의 형식은 다음과 같습니다.

```
RestoreReportMetrics: Object
- durationMs: Double
```

다음은 RestoreReportMetrics 객체의 예입니다.

```
{
    "durationMs": 15.19
}
```

RuntimeDoneMetrics

호출 단계에 대한 지표가 포함된 객체입니다.

- 유형 – Object

RuntimeDoneMetrics 객체의 형식은 다음과 같습니다.

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

다음은 RuntimeDoneMetrics 객체의 예입니다.

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

범위에 대한 세부 정보가 포함된 객체입니다. 범위는 트레이스의 작업 또는 작업 단위를 나타냅니다. 범위에 대한 자세한 내용은 OpenTelemetry 문서 웹 사이트의 추적 API 페이지에서 범위를 참조하세요.

Lambda는 platform.RuntimeDone 이벤트에 대해 다음과 같은 범위를 지원합니다.

- responseLatency 범위는 Lambda 함수가 응답의 전송을 시작하는 데 걸린 시간을 나타냅니다.
- responseDuration 범위는 Lambda 함수가 전체 응답의 전송을 마치는 데 걸린 시간을 나타냅니다.
- runtimeOverhead 범위는 Lambda 런타임이 다음 함수 간접 호출을 처리할 준비가 되었음을 알리는 데 걸린 시간을 나타냅니다. 이는 런타임이 함수 응답을 반환한 후 다음 이벤트를 가져오기 위해 다음 간접 호출 API를 호출하는 데 걸린 시간입니다.

다음은 responseLatency 범위 객체의 예입니다.

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
```

```
        "durationMs": 23.02
      }
```

Status

초기화 또는 호출 단계의 상태를 설명하는 객체입니다. 상태가 `failure` 또는 `error` 인 경우`Status` 객체에는 오류를 설명하는 `errorType` 필드도 포함됩니다.

- 유형 – `Object`
- 유효한 상태 값 – `success|failure|error|timeout`

TraceContext

트레이스의 속성을 설명하는 객체입니다.

- 유형 – `Object`

`TraceContext` 객체의 형식은 다음과 같습니다.

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

다음은 `TraceContext` 객체의 예입니다.

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

the section called “TraceContext” 객체의 추적 유형을 설명하는 문자열 열거형입니다.

- 유형 – `String`
- 유효한 값 – `X-Amzn-Trace-Id`

Lambda **Event** 텔레메트리 API 객체를 스팬으로 변환 OpenTelemetry

AWS Lambda텔레메트리 API 스키마는 (oTel) 과 의미상 호환됩니다. OpenTelemetry 즉, AWS Lambda 텔레메트리 API `Event` 객체를 (oTel) 범위로 변환할 OpenTelemetry 수 있습니다. 변환할 때는 단일 `Event` 객체를 단일 OTel 범위에 매핑해서는 안 됩니다. 대신 수명 주기 단계와 관련한 세 가지 이벤트를 모두 단일 OTel 범위에 표시해야 합니다. 예를 들어 `start`, `runtimeDone` 및 `runtimeReport` 이벤트는 단일 함수 호출을 나타냅니다. 이 세 가지 이벤트를 모두 하나의 OTel 범위로 표시합니다.

범위 이벤트 또는 하위(중첩) 범위를 사용하여 이벤트를 변환할 수 있습니다. 이 페이지의 표에서는 두 접근 방식에 대한 텔레메트리 API 스키마 속성과 OTel 범위 속성 간의 매핑을 설명합니다. oTel 범위에 대한 자세한 내용은 문서 웹 사이트의 추적 API 페이지에서 범위를 참조하십시오. OpenTelemetry

Sections

OTel 범위를 범위 이벤트에 매핑

다음 표에서 e는 텔레메트리 소스에서 발생하는 이벤트를 나타냅니다.

`*Start` 이벤트 매핑

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Name`	확장은 `type` 필드를 기반으로 이 값을 생성합니다.
`Span.StartTime`	`e.time`를 사용합니다.
`Span.EndTime`	이벤트가 아직 완료되지 않았으므로 해당 사항이 없습니다.
`Span.Kind`	`Server`로 설정합니다.
`Span.Status`	`Unset`로 설정합니다.
`Span.TraceId`	`e.tracing.value` 에 있는 AWS X-Ray 헤더를 구문 분석한 다음 `TraceId` 값을 사용합니다.

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.ParentId`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Parent` 값을 사용합니다.
`Span.SpanId`	가능한 경우 `e.tracing.spanId` 를 사용합니다. 그렇지 않으면 새 `SpanId`를 생성합니다.
`Span.SpanContext.TraceState`	X-Ray 트레이스 컨텍스트의 경우 해당 사항이 없습니다.
`Span.SpanContext.TraceFlags`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Sampled` 값을 사용합니다.
`Span.Attributes`	확장은 여기에 사용자 지정 값을 추가할 수 있습니다.

***RuntimeDone** 이벤트 매핑

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Name`	확장은 `type` 필드를 기반으로 값을 생성합니다.
`Span.StartTime`	일치하는 `*Start` 이벤트의 `e.time`을 사용합니다. 또는 `e.time - e.metrics.durationMs` 를 사용합니다.
`Span.EndTime`	이벤트가 아직 완료되지 않았으므로 해당 사항이 없습니다.
`Span.Kind`	`Server`로 설정합니다.
`Span.Status`	`e.status`가 `success`와 같지 않으면 `Error`로 설정합니다. 그렇지 않으면 `Ok`(으)로 설정합니다.

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Events[]`	`e.spans[]` 를 사용합니다.
`Span.Events[i].Name`	`e.spans[i].name` 를 사용합니다.
`Span.Events[i].Time`	`e.spans[i].start` 를 사용합니다.
`Span.TraceId`	`e.tracing.value` 에 있는 AWS X-Ray 헤더를 구문 분석한 다음 `TraceId` 값을 사용합니다.
`Span.ParentId`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Parent` 값을 사용합니다.
`Span.SpanId`	`*Start` 이벤트와 동일한 `SpanId`를 사용합니다. 사용할 수 없는 경우 `e.tracing.spanId` 를 사용하거나 새 `SpanId`를 생성합니다.
`Span.SpanContext.TraceState`	X-Ray 트레이스 컨텍스트의 경우 해당 사항이 없습니다.
`Span.SpanContext.TraceFlags`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Sampled` 값을 사용합니다.
`Span.Attributes`	확장은 여기에 사용자 지정 값을 추가할 수 있습니다.

`*Report` 이벤트 매핑

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Name`	확장은 `type` 필드를 기반으로 값을 생성합니다.
`Span.StartTime`	일치하는 `*Start` 이벤트의 `e.time`을 사용합니다. 또는 `e.time - e.metrics.durationMs` 를 사용합니다.

OpenTelemetry	Lambda 텔레메트리 API 스키마
Span.EndTime	e.time를 사용합니다.
Span.Kind	Server로 설정합니다.
Span.Status	*RuntimeDone 이벤트와 동일한 값을 사용합니다.
Span.TraceId	e.tracing.value 에 있는 AWS X-Ray 헤더를 구문 분석한 다음 TraceId 값을 사용합니다.
Span.ParentId	e.tracing.value 에 있는 X-Ray 헤더를 구문 분석한 다음 Parent 값을 사용합니다.
Span.SpanId	*Start 이벤트와 동일한 SpanId를 사용합니다. 사용할 수 없는 경우 e.tracing.spanId 를 사용하거나 새 SpanId를 생성합니다.
Span.SpanContext.TraceState	X-Ray 트레이스 컨텍스트의 경우 해당 사항이 없습니다.
Span.SpanContext.TraceFlags	e.tracing.value 에 있는 X-Ray 헤더를 구문 분석한 다음 Sampled 값을 사용합니다.
Span.Attributes	확장은 여기에 사용자 지정 값을 추가할 수 있습니다.

OTel 범위를 하위 범위에 매핑

다음 표에서는 Lambda 텔레메트리 API 이벤트를 *RuntimeDone 범위의 하위(중첩) 범위가 포함된 OTel 범위로 변환하는 방법을 설명합니다. *Start 및 *Report 매핑의 경우 하위 범위와 동일하므로 the section called “OTel 범위를 범위 이벤트에 매핑”의 테이블을 참조하세요. 이 표에서 e는 텔레메트리 소스에서 발생하는 이벤트를 나타냅니다.

***RuntimeDone** 이벤트 매핑

OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Name`	확장은 `type` 필드를 기반으로 값을 생성합니다.
`Span.StartTime`	일치하는 `*Start` 이벤트의 `e.time`을 사용합니다. 또는 `e.time - e.metrics.durationMs` 를 사용합니다.
`Span.EndTime`	이벤트가 아직 완료되지 않았으므로 해당 사항이 없습니다.
`Span.Kind`	`Server`로 설정합니다.
`Span.Status`	`e.status`가 `success`와 같지 않으면 `Error`로 설정합니다. 그렇지 않으면 `Ok`(으)로 설정합니다.
`Span.TraceId`	`e.tracing.value` 에 있는 AWS X-Ray 헤더를 구문 분석한 다음 `TraceId` 값을 사용합니다.
`Span.ParentId`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Parent` 값을 사용합니다.
`Span.SpanId`	`*Start` 이벤트와 동일한 `SpanId`를 사용합니다. 사용할 수 없는 경우 `e.tracing.spanId` 를 사용하거나 새 `SpanId`를 생성합니다.
`Span.SpanContext.TraceState`	X-Ray 트레이스 컨텍스트의 경우 해당 사항이 없습니다.
`Span.SpanContext.TraceFlags`	`e.tracing.value` 에 있는 X-Ray 헤더를 구문 분석한 다음 `Sampled` 값을 사용합니다.

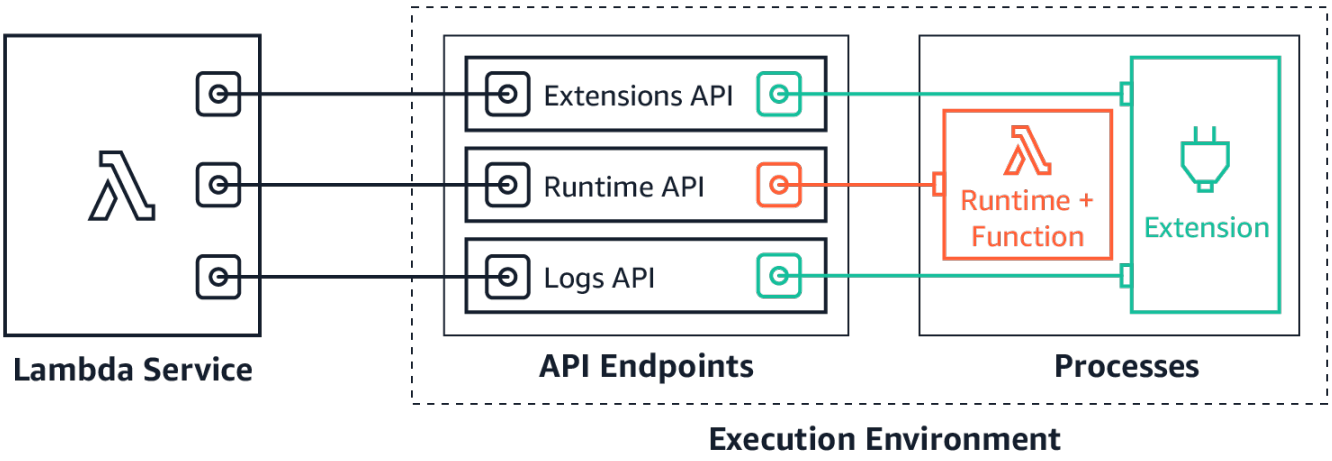
OpenTelemetry	Lambda 텔레메트리 API 스키마
`Span.Attributes`	확장은 여기에 사용자 지정 값을 추가할 수 있습니다.
`ChildSpan[i].Name`	`e.spans[i].name` 를 사용합니다.
`ChildSpan[i].StartTime`	`e.spans[i].start` 를 사용합니다.
`ChildSpan[i].EndTime`	`e.spans[i].start + e.spans[i].durations` 를 사용합니다.
`ChildSpan[i].Kind`	상위 `Span.Kind` 와 동일합니다.
`ChildSpan[i].Status`	상위 `Span.Status` 와 동일합니다.
`ChildSpan[i].TraceId`	상위 `Span.TraceId` 와 동일합니다.
`ChildSpan[i].ParentId`	상위 `Span.SpanId` 를 사용합니다.
`ChildSpan[i].SpanId`	새 `SpanId`를 생성합니다.
`ChildSpan[i].SpanContext.TraceState`	X-Ray 트레이스 컨텍스트의 경우 해당 사항이 없습니다.
`ChildSpan[i].SpanContext.TraceFlags`	상위 `Span.SpanContext.TraceFlags` 와 동일합니다.

Lambda 로그 API

> **Important**
>
> Lambda 텔레메트리 API는 Lambda 로그 API를 대체합니다. 로그 API도 계속 정상적으로 작동하지만, 앞으로는 텔레메트리 API만 사용하는 것이 좋습니다. 텔레메트리 API 또는 로그 API를 사용하여 확장에서 텔레메트리 스트림을 구독할 수 있습니다. 이러한 API 중 하나를 사용하여 구독한 후 다른 API를 사용하여 구독하려고 하면 오류가 반환됩니다.

Lambda는 런타임 로그를 자동으로 캡처하여 Amazon으로 스트리밍합니다. CloudWatch 이 로그 스트림에는 함수 코드 및 익스텐션이 생성하는 로그와 함수 호출의 일부로 Lambda가 생성하는 로그가 포함됩니다.

Lambda 익스텐션은 Lambda Runtime Logs API를 사용하여 Lambda 실행 환경 내에서 로그 스트림을 구독할 수 있습니다. Lambda가 로그를 익스텐션에 스트리밍하면 익스텐션은 로그를 처리하고 필터링하여 원하는 모든 대상에 전송할 수 있습니다.



Logs API를 사용하면 익스텐션이 세 가지 다른 로그 스트림을 구독할 수 있습니다.

- Lambda 함수가 생성하고 `stdout` 또는 `stderr`에 쓰는 함수 로그.
- 익스텐션 코드가 생성하는 익스텐션 로그.
- Lambda 플랫폼 로그는 호출 및 익스텐션과 관련된 이벤트 및 오류를 기록합니다.

> **Note**
>
> 확장 프로그램이 하나 이상의 로그 스트림을 CloudWatch 구독하는 경우에도 Lambda는 모든 로그를 로 전송합니다.

주제

수신 로그 구독

Lambda 익스텐션은 Logs API에 구독 요청을 전송하여 수신 로그를 구독할 수 있습니다.

수신 로그를 구독하려면 익스텐션 식별자(`Lambda-Extension-Identifier`)가 필요합니다. 먼저 익스텐션을 등록하여 익스텐션 식별자를 수신합니다. 그런 다음 초기화중에 Logs API를 구독합니다. 초기화 단계가 완료되면 Lambda가 구독 요청을 처리하지 않습니다.

> **Note**
>
> Logs API 구독은 멱등성이 있습니다. 중복 구독 요청으로 인해 구독이 중복되지 않습니다.

메모리 사용량

구독자 수가 증가함에 따라 메모리 사용량이 선형적으로 증가합니다. 각 구독은 새 메모리 버퍼를 열어 로그를 저장하기 때문에 메모리 리소스를 사용합니다. 메모리 사용량을 최적화하기 위해 버퍼링 구성을 조정할 수 있습니다. 버퍼 메모리 사용량은 실행 환경의 전체 메모리 소비에 포함됩니다.

대상 프로토콜

다음 프로토콜 중 하나를 선택하여 로그를 수신할 수 있습니다.

1. HTTP(권장) - Lambda가 로그를 로컬 HTTP 엔드포인트(`http://sandbox.localdomain:${PORT}/${PATH}`)에 JSON 형식의 레코드 배열로 전달합니다. `$PATH` 파라미터는 선택 항목입니다. HTTPS가 아닌 HTTP만 지원됩니다. PUT 또는 POST를 통해 로그를 수신하도록 선택할 수 있습니다.
2. TCP – Lambda가 로그를 새 줄 구분 JSON(NDJSON) 형식으로 TCP 포트에 전달합니다.

TCP 대신 HTTP를 사용하는 것이 좋습니다. TCP를 사용하면 Lambda 플랫폼에서는 로그를 애플리케이션 계층에 전송할 때 인식할 수 없습니다. 따라서 익스텐션이 충돌하는 경우 로그가 손실될 수 있습니다. HTTP에는 이러한 제한이 없습니다.

또한 수신 로그를 구독하기 전에 로컬 HTTP 수신기 또는 TCP 포트를 설정하는 것이 좋습니다. 설치하는 동안 다음 사항에 유의하세요.

- Lambda는 실행 환경 내에 있는 대상에만 로그를 보냅니다.
- Lambda는 리스너가 없거나 POST 또는 PUT 요청으로 인해 오류가 발생하는 경우 로그 전송을 다시 시도합니다(백오프 포함). 로그 구독자가 충돌하면 Lambda가 실행 환경을 다시 시작한 후 로그를 계속 수신합니다.
- Lambda는 포트 9001을 예약합니다. 다른 포트 번호 제한이나 권장 사항은 없습니다.

버퍼링 구성

Lambda는 로그를 버퍼링하여 구독자에게 전달할 수 있습니다. 다음과 같은 선택적 필드를 지정하여 구독 요청에서 이 동작을 구성할 수 있습니다. Lambda는 지정하지 않은 필드에 기본값을 사용합니다.

- timeoutMs – 배치를 버퍼링할 최대 시간(밀리초)입니다. 기본값: 1,000. 최소: 25 최대값: 30,000.
- maxBytes – 메모리에 버퍼링할 로그의 최대 크기(바이트)입니다. 기본값: 262,144. 최소값: 262,144. 최대값: 1,048,576.
- maxItems – 메모리에 버퍼링할 최대 이벤트 수입니다. 기본값: 10,000. 최소값: 1,000. 최대값: 10,000.

버퍼링 구성 중에 다음 사항에 유의하세요.

- Lambda는 런타임 충돌 등으로 인해 입력 스트림이 닫히면 로그를 플러시합니다.

- 각 구독자는 구독 요청에서 다른 버퍼링 구성을 지정할 수 있습니다.
- 데이터를 읽는 데 필요한 버퍼 크기를 고려하세요. 최대 `2*maxBytes+metadata` 크기의 페이로드를 수신할 수 있도록 준비하세요. 여기서, `maxBytes`는 구독 요청에서 구성됩니다. 예를 들어 Lambda는 각 레코드에 다음 메타데이터 바이트를 추가합니다.

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- 구독자가 들어오는 로그를 충분히 빠르게 처리할 수 없는 경우 Lambda는 메모리 사용률을 범위 내로 유지하기 위해 로그를 삭제할 수 있습니다. 삭제된 레코드의 수를 나타내기 위해 Lambda는 `platform.logsDropped` 로그를 전송합니다.

구독 예

다음 예제에서는 플랫폼 및 함수 로그를 구독하는 요청을 보여줍니다.

```
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"
  }
}
```

요청이 성공하면 구독자는 HTTP 200 성공 응답을 받습니다.

```
HTTP/1.1 200 OK
"OK"
```

Logs API용 샘플 코드

로그를 사용자 지정 대상으로 보내는 방법을 보여 주는 샘플 코드는 AWS Lambda 컴퓨팅 블로그의 AWS 익스텐션을 사용하여 사용자 지정 대상으로 로그 전송을 참조하세요.

기본 Lambda 확장 프로그램을 개발하고 Logs API를 구독하는 방법을 보여주는 Python 및 Go 코드 예제는 샘플 리포지토리의 확장 프로그램을 AWS Lambda참조하십시오. AWS GitHub Lambda 익스텐션 빌드에 대한 자세한 내용은 the section called “익스텐션 API” 단원을 참조하세요.

Logs API 참조

`AWS_LAMBDA_RUNTIME_API` 환경 변수에서 Logs API 엔드포인트를 검색할 수 있습니다. API 요청을 보내려면 API 경로 앞에 `2020-08-15/` 접두사를 사용합니다. 예:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

로그 API 버전 2020-08-15의 OpenAPI 사양은 다음에서 확인할 수 있습니다.zip logs-api-request

Subscribe

Lambda 실행 환경에서 사용할 수 있는 하나 이상의 로그 스트림을 구독하기 위해 익스텐션이 Subscribe API 요청을 전송합니다.

경로 – `/logs`

메서드 – PUT

본문 파라미터

`destination` 단원을 참조하세요.the section called “대상 프로토콜” 필수 항목 여부: 예. 유형: 문자열.

`buffering` 단원을 참조하세요.the section called “버퍼링 구성” 필수 항목 여부: 아니요 유형: 문자열.

`types` – 수신할 로그 유형의 배열입니다. 필수 항목 여부: 예. 유형: 문자열 배열 유효한 값: "platform", "function", "extension".

`schemaVersion` – 필수 항목 여부: 아니요 기본값: "2020-08-15". 익스텐션에서 `platform.runtimeDone` 메시지를 수신하려면 ‘2021-03-18’으로 설정합니다.

응답 파라미터

구독 응답에 대한 OpenAPI 사양(버전 2020-08-15)은 HTTP 및 TCP 프로토콜에 사용할 수 있습니다.

- logs-api-http-responseHTTP: .zip
- TCP: .zip logs-api-tcp-response

응답 코드

- 200 - 요청이 성공적으로 완료되었습니다.
- 202 - 요청이 수락되었습니다. 로컬 테스트 중 구독 요청에 대한 응답입니다.
- 4XX - 잘못된 요청
- 500 - 서비스 오류

요청이 성공하면 구독자는 HTTP 200 성공 응답을 받습니다.

```
HTTP/1.1 200 OK
"OK"
```

요청이 실패하면 구독자는 오류 응답을 받습니다. 예:

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

로그 메시지

Logs API를 사용하면 익스텐션이 세 가지 다른 로그 스트림을 구독할 수 있습니다.

- Function – Lambda 함수가 생성하고 `stdout` 또는 `stderr`에 쓰는 함수 로그.
- 익스텐션 - 익스텐션 코드가 생성하는 로그.
- 플랫폼 - 런타임 플랫폼이 생성하는 로그로 호출 및 익스텐션과 관련된 이벤트 및 오류 기록.

주제

- 함수 로그
- 익스텐션 로그
- 플랫폼 로그

함수 로그

Lambda 함수 및 내부 익스텐션은 함수 로그를 생성하여 `stdout` 또는 `stderr`에 작성합니다.

다음 예에서는 함수 로그 메시지의 형식을 보여 줍니다.{ "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

익스텐션 로그

익스텐션은 익스텐션 로그를 생성할 수 있습니다. 로그 형식은 함수 로그와 같습니다.

플랫폼 로그

Lambda는 `platform.start`, `platform.end`, `platform.fault`와 같은 플랫폼 이벤트에 대한 로그 메시지를 생성합니다

또는 `platform.runtimeDone` 로그 메시지를 포함하는 로그 API 스키마의 2021-03-18 버전을 구독할 수 있습니다.

플랫폼 로그 메시지 예

다음 예에서는 플랫폼 시작 로그와 플랫폼 종료 로그를 보여 줍니다. 이러한 로그는 requestID가 지정하는 호출에 대한 호출 시작 시간 및 호출 종료 시간을 나타냅니다.

```
{
    "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"}
}
```

플랫폼. initRuntimeDone로그 메시지는 Init 라이프사이클 `Runtime init` 단계의 일부인 하위 단계의 상태를 보여줍니다. `Runtime init`가 성공하면 런타임이 `/next` 런타임 API 요청(`on-demand` 및 `provisioned-concurrency` 초기화 유형의 경우) 또는 `restore/next`(`snap-start` 초기화 유형의 경우)를 전송합니다. 다음 예제는 성공적인 플랫폼을 보여줍니다. initRuntimeDone`snap-start`초기화 유형에 대한 로그 메시지.

```
{
  "time":"2022-07-17T18:41:57.083Z",
```

```
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

platform.initReport 로그 메시지에는 `Init` 단계가 지속된 시간과 이 단계에서 청구된 시간(밀리초)이 표시됩니다. 초기화 유형이 `provisioned-concurrency`인 경우 Lambda는 호출 중에 이 메시지를 보냅니다. 초기화 유형이 `snap-start`인 경우 Lambda는 스냅샷을 복원한 후에 이 메시지를 보냅니다. 다음 예에서는 `snap-start` 초기화 유형에 대한 platform.initReport 로그 메시지를 보여줍니다.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

플랫폼 보고서 로그에는 requestId가 지정하는 호출에 대한 지표가 포함됩니다. 이 `initDurationMs` 필드는 호출에 콜드 스타트가 포함된 경우에만 로그에 포함됩니다. AWS X-Ray 추적이 활성 상태인 경우 로그에 X-Ray 메타데이터가 포함됩니다. 다음 예에서는 콜드 스타트를 포함한 호출에 대한 플랫폼 보고서 로그를 보여 줍니다.

```
{
    "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
        }
    }
}
```

플랫폼 오류 로그는 런타임 또는 실행 환경 오류를 캡처합니다. 다음 예제에서는 플랫폼 장애 로그 메시지를 보여줍니다.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

Lambda는 익스텐션이 익스텐션 API에 등록할 때 플랫폼 익스텐션 로그를 생성합니다. 다음 예에서는 플랫폼 익스텐션 메시지를 보여줍니다.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

Lambda는 익스텐션이 로그 API를 구독할 때 플랫폼 로그 구독 로그를 생성합니다. 다음 예는 로그 구독 로그 메시지를 보여줍니다.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda는 익스텐션이 수신 중인 로그 수를 처리할 수 없는 경우 플랫폼 로그 삭제 로그를 생성합니다. 다음 예제에서는 `platform.logsDropped` 로그 메시지를 보여줍니다.

```
{
    "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 로그 메시지에는 Restore 단계가 시작된 시간이 표시됩니다(snap-start 초기화 유형만 해당). 예제

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

platform.restoreReport 로그 메시지에는 Restore 단계가 지속된 시간과 이 단계에서 청구된 시간(밀리초)이 표시됩니다(snap-start 초기화 유형만 해당). 예제

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

플랫폼 **runtimeDone** 메시지

구독 요청에서 스키마 버전을 '2021-03-18'로 설정하면 Lambda는 함수 호출이 성공적으로 완료되거나 오류가 발생한 경우 platform.runtimeDone 메시지를 보냅니다. 익스텐션은 이 메시지를 사용하여 이 함수 호출에 대한 모든 원격 분석 수집을 중지할 수 있습니다.

스키마 버전 2021-03-18의 로그 이벤트 유형에 대한 OpenAPI 사양은 schema-2021-03-18.zip에서 사용할 수 있습니다.

Lambda는 런타임에서 Next 또는 Error 런타임 API 요청을 보낼 때 platform.runtimeDone 로그 메시지를 생성합니다. platform.runtimeDone 로그는 함수 호출이 완료되었음을 로그 API 사용자

에게 알립니다. 익스텐션은 이 정보를 사용하여 해당 호출 중에 수집된 모든 원격 분석을 보낼 시간을 결정할 수 있습니다.

예제

Lambda는 함수 호출이 완료되면 런타임에서 NEXT 요청을 보낸 후 `platform.runtimeDone` 메시지를 전송합니다. 다음 예에서는 각 상태 값에 대한 메시지(성공, 실패 및 시간 초과)를 보여줍니다.

Example 성공 메시지 예

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example 오류 메시지 예

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example 시간 초과 메시지 예

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example 예제 플랫폼. restoreRuntimeDone 메시지 (**snap-start**초기화 유형만 해당)

플랫폼. restoreRuntimeDone로그 메시지에는 `Restore` 단계가 성공했는지 여부가 표시됩니다. Lambda는 런타임이 `restore/next` 런타임 API 요청을 보낼 때 이 로그 메시지를 보냅니다. 가능한 상태로는 성공, 실패 및 시간 초과가 있습니다. 다음 예는 성공적인 플랫폼을 보여줍니다. restoreRuntimeDone로그 메시지.

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Lambda 문제 해결

다음 항목에서는 Lambda API, 콘솔 또는 도구를 사용할 때 발생할 수 있는 오류 및 문제에 대한 문제 해결 조언을 제공합니다. 여기에 나열되지 않은 문제를 발견하는 경우 이 페이지의 [Feedback] 버튼을 사용하여 해당 문제를 보고할 수 있습니다.

문제 해결 조언과 일반적인 지원 질문에 대한 답변은 AWS 지식 센터를 참조하세요.

Lambda 애플리케이션 디버깅 및 문제 해결에 대한 자세한 내용은 Serverless Land의 Debugging을 참조하세요.

주제

Lambda의 배포 문제 해결

함수를 업데이트할 때 Lambda는 업데이트된 코드 또는 설정으로 함수의 새 인스턴스를 시작하여 변경 사항을 배포합니다. 배포 오류로 인해 새 버전이 사용되지 않으며, 이는 배포 패키지, 코드, 권한 또는 도구 문제 때문에 발생할 수 있습니다.

Lambda API를 사용하거나 AWS CLI와 같은 클라이언트를 사용하여 함수로 업데이트를 직접 배포하면 출력에서 직접 Lambda의 오류를 볼 수 있습니다. AWS CloudFormation, AWS CodeDeploy 또는 AWS CodePipeline 같은 서비스를 사용하는 경우에는 해당 서비스 로그 또는 이벤트 스트림에서 Lambda의 응답을 찾아보세요.

다음 항목에서는 Lambda API, 콘솔 또는 도구를 사용할 때 발생할 수 있는 오류 및 문제에 대한 문제 해결 조언을 제공합니다. 여기에 나열되지 않은 문제를 발견하는 경우 이 페이지의 [Feedback] 버튼을 사용하여 해당 문제를 보고할 수 있습니다.

문제 해결 조언과 일반적인 지원 질문에 대한 답변은 AWS 지식 센터를 참조하세요.

Lambda 애플리케이션 디버깅 및 문제 해결에 대한 자세한 내용은 Serverless Land의 Debugging을 참조하세요.

주제

일반: 권한이 거부됨 / 해당 파일을 로드할 수 없음

오류: EACCES: 권한 거부, '/var/task/index.js' 열기

오류: 해당 파일을 로드할 수 없음 -- 함수

오류: [Errno 13] 권한 거부: '/var/task/function.py'

Lambda 런타임은 배포 패키지의 파일을 읽을 수 있는 권한이 필요합니다. Linux 권한 8진수 표기법에서는 Lambda에 실행 불가능한 파일(rw-r--r--)에 대한 644개의 권한과 디렉터리 및 실행 파일에 대한 755개의 권한(rwxr-xr-x)이 필요합니다.

Linux 및 MacOS에서는 chmod 명령을 사용하여 배포 패키지의 파일 및 디렉터리에 대한 파일 권한을 변경합니다. 예를 들어, 실행 파일에 올바른 권한을 부여하려면 다음 명령을 실행합니다.

```
chmod 755 <filepath>
```

Windows에서 파일 권한을 변경하려면 Microsoft Windows 설명서의 Set, View, Change, or Remove Permissions on an Object를 참조하세요.

일반: UpdateFunctionCode 호출 시 오류 발생

오류: UpdateFunctionCode 작업을 호출할 때 오류 발생(RequestEntityTooLargeException)

배포 패키지 또는 계층 아카이브를 Lambda에 직접 업로드하면 ZIP 파일의 크기가 50MB로 제한됩니다. 더 큰 파일을 업로드하려면 Amazon S3에 저장하고 S3Bucket 및 S3Key 파라미터를 사용하세요.

> **Note**
>
> AWS CLI, AWS SDK 또는 다른 방법으로 파일을 직접 업로드하면 이진수 ZIP 파일이 base64로 변환되어 크기가 약 30% 증가합니다. 이를 허용하고 요청에서 다른 파라미터 크기를 허용하려면 Lambda가 적용하는 실제 요청 크기 제한이 더 큽니다. 이로 인해 50MB 제한은 근사값입니다.

Amazon S3: 오류 코드 PermanentRedirect.

오류: GetObject를 수행하는 동안 오류가 발생했습니다. S3 오류 코드: PermanentRedirect. S3 오류 메시지: 버킷이 us-east-2 리전에 있습니다. 이 리전을 사용하여 요청을 다시 시도하세요

Amazon S3 버킷에서 함수의 배포 패키지를 업로드할 때 버킷은 함수와 동일한 리전에 있어야 합니다. UpdateFunctionCode 호출 시 Amazon S3 객체를 지정하거나 AWS CLI 또는 AWS SAM CLI에서 패키지를 사용하고 명령을 배포할 때 이러한 문제가 발생할 수 있습니다. 애플리케이션을 개발하는 각 리전에 대한 배포 아티팩트 버킷을 생성합니다.

일반: 찾을 수 없음, 로드할 수 없음, 가져올 수 없음, 클래스를 찾을 수 없음, 해당 파일 또는 디렉터리가 없음

오류: 모듈 'function'을 찾을 수 없음

오류: 해당 파일을 로드할 수 없음 -- 함수

오류: 모듈 'function'을 가져올 수 없음

오류: 클래스를 찾을 수 없음: function.Handler

오류: fork/exec /var/task/function: 이러한 파일 또는 디렉터리 없음

오류: 어셈블리 'Function'에서 'Function.Handler' 유형을 로드할 수 없음.

함수의 핸들러 구성에 있는 파일 또는 클래스의 이름이 코드와 일치하지 않습니다. 자세한 내용은 다음 단원을 참조하십시오.

일반: 정의되지 않은 메서드 핸들러

오류: index.handler가 정의되지 않거나 내보내지지 않음

오류: 모듈 'function'에 핸들러 'handler'가 없음

오류: #<LambdaHandler:0x000055b76ccebf98>에 정의되지 않은 메서드 `handler'

오류: class function.Handler에서 적절한 메서드 서명이 있는 handleRequest라는 이름의 퍼블릭 메서드 없음

오류: 어셈블리 'Function'에서 'Function.Handler' 유형의 'handleRequest' 메서드를 찾을 수 없음

함수의 핸들러 구성에 있는 핸들러 메서드의 이름이 코드와 일치하지 않습니다. 각 런타임은 *filename.methodname* 같은 핸들러의 이름 지정 규칙을 정의합니다. 핸들러는 함수가 호출될 때 런타임이 실행하는 함수 코드의 메서드입니다.

일부 언어의 경우 Lambda는 핸들러 메서드에 특정 이름이 있어야 하는 인터페이스를 라이브러리에 제공합니다. 각 언어의 핸들러 이름 지정에 관한 자세한 내용은 다음 항목을 참조하세요.

- Node.js를 사용하여 Lambda 함수 빌드
- Python을 사용하여 Lambda 함수 빌드
- Ruby를 사용하여 Lambda 함수 빌드
- Java를 사용하여 Lambda 함수 빌드
- Go를 사용하여 Lambda 함수 빌드
- C#을 사용하여 Lambda 함수 빌드
- PowerShell을 사용하여 Lambda 함수 빌드

Lambda: 계층 변환 실패

오류: Lambda 계층 변환에 실패했습니다. 이 문제를 해결하는 방법에 대한 자세한 내용은 Lambda 사용 설명서의 Lambda의 배포 문제 해결 페이지를 참조하세요.

계층으로 Lambda 함수를 구성하면 Lambda는 계층을 함수 코드와 병합합니다. 이 프로세스가 완료되지 않으면 Lambda는 이 오류를 반환합니다. 이 오류가 발생한 경우 다음 단계를 수행합니다.

- 계층에서 사용하지 않는 파일 삭제
- 계층에서 심볼 링크 삭제
- 모든 함수 계층의 디렉터리와 이름이 같은 모든 파일의 이름 변경

Lambda: InvalidParameterValueException or RequestEntityTooLargeException

오류: InvalidParameterValueException: 제공된 환경 변수가 4KB 제한을 초과했기 때문에 Lambda에서 환경 변수를 구성할 수 없음. 측정된 문자열: {"A1":"uSFeY5cyPiPn7AtnX5BsM...

오류: RequestEntityTooLargeException: UpdateFunctionConfiguration 작업에 대한 요청은 5,120바이트보다 작아야 함

함수의 구성에 저장되는 변수 객체의 최대 크기는 4,096바이트를 초과할 수 없습니다. 여기에는 키 이름, 값, 따옴표, 쉼표 및 대괄호가 포함됩니다. HTTP 요청 본문의 총 크기도 제한됩니다.

```
{
    "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"
        }
    },
    ...
}
```

이 예제에서 객체는 39자이며 공백 없이 문자열 `{"BUCKET":"DOC-EXAMPLE-BUCKET","KEY":"file.txt"}`로 저장될 때 39바이트를 차지합니다. 환경 변수 값의 표준 ASCII 문자는 각각 1바이트를 사용합니다. 확장 ASCII 및 유니코드 문자는 문자당 2~4바이트를 사용할 수 있습니다.

Lambda: InvalidParameterValueException

오류: InvalidParameterValueException: 제공된 환경 변수에 현재 수정 불가능한 예약된 키가 포함되어 있기 때문에 Lambda에서 환경 변수를 구성할 수 없음.

Lambda는 내부 사용을 위해 일부 환경 변수 키를 예약합니다. 예를 들어 `AWS_REGION`는 런타임에 현재 리전을 결정하는 데 사용되며 재정의될 수 없습니다. `PATH`를 비롯한 다른 변수는 런타임에 사용되지만 함수 구성에서 확장될 수 있습니다. 전체 목록은 정의된 런타임 환경 변수 단원을 참조하세요.

Lambda: 동시성 및 메모리 할당량

Error: Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value

Error: 'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008

이러한 오류는 계정에 대한 동시성 또는 메모리 할당량이 초과될 때 발생합니다. 새 AWS 계정에 감소된 동시성 및 메모리 할당량이 적용되었습니다. 동시성과 관련한 오류를 해결하려면 할당량 증가를 요청하면 됩니다. 메모리 할당량 증가는 요청할 수 없습니다.

- 동시성: 예약되거나 프로비저닝된 동시성을 사용하여 함수를 만들려고 하거나 함수별 동시성 요청(PutFunctionConcurrency)이 계정의 동시성 할당량을 초과하는 경우 오류가 발생할 수 있습니다.
- 메모리: 함수에 할당된 메모리의 양이 계정의 메모리 할당량을 초과하면 오류가 발생합니다.

Lambda의 호출 문제 해결

Lambda 함수를 호출하면 Lambda는 요청을 검증하고 이벤트를 함수로 전송하거나 비동기 호출의 경우 이벤트 대기열로 보내기 전에 용량 확장을 확인합니다. 요청 파라미터, 이벤트 구조, 함수 설정, 사용자 권한, 리소스 권한 또는 한도와 관련된 문제로 인해 호출 오류가 발생할 수 있습니다.

함수를 직접 호출하면 Lambda에서 보낸 응답에 호출 오류가 표시됩니다. 이벤트 소스 매핑으로 또는 다른 서비스를 통해 함수를 비동기적으로 호출하는 경우 로그, 배달 못한 편지 대기열 또는 실패한 이벤트 대상에 오류가 있을 수 있습니다. 오류 처리 옵션 및 재시도 동작은 함수를 호출하는 방법과 오류 유형에 따라 다릅니다.

`Invoke` 작업이 반환할 수 있는 오류 유형 목록은 호출을 참조하세요.

IAM: lambda:InvokeFunction 권한 없음

오류: 사용자: arn:aws:iam::123456789012:사용자/개발자가 수행할 권한이 없음: 리소스의 lambda:InvokeFunction: my-function

사용자 또는 수임하고자 하는 역할에는 함수를 호출할 수 있는 권한이 있어야 합니다. 이 요구 사항은 Lambda 함수와 함수를 호출하는 다른 컴퓨팅 리소스에도 적용됩니다. AWS 관리형 정책 AWSLambdaRole을 사용자에게 추가하거나, 대상 함수에서 `lambda:InvokeFunction` 작업을 허용하는 사용자 정의 정책을 추가하세요.

> **Note**
>
> IAM 작업의 이름(`lambda:InvokeFunction`)은 `Invoke` Lambda API 작업을 나타냅니다.

자세한 정보는 AWS Lambda에서 권한 관리을 참조하십시오.

Lambda: 유효한 부트스트랩을 찾을 수 없음(Runtime.InvalidEntrypoint)

오류: Couldn't find valid bootstrap(s): [/var/task/bootstrap /opt/bootstrap]

이 오류는 일반적으로 배포 패키지의 루트에 `bootstrap` 실행 파일이 없을 때 발생합니다. 예를 들어, .zip 파일이 포함된 `provided.al2023` 함수를 배포하는 경우 `bootstrap` 파일은 디렉터리가 아닌 .zip 파일의 루트에 있어야 합니다.

Lambda: ResourceConflictException 작업을 수행할 수 없음

오류: ResourceConflictException: 지금은 작업을 수행할 수 없음. 함수가 현재 보류 중 상태

함수를 만들 때 Virtual Private Cloud(VPC)에 함수를 연결하면 Lambda에서 탄력적 네트워크 인터페이스를 생성하는 동안 함수가 `Pending` 상태가 됩니다. 이 시간 동안에는 함수를 호출하거나 수정할 수 없습니다. 함수를 만든 후 VPC에 함수를 연결하면 업데이트가 보류 중인 동안 함수를 호출할 수 있지만 해당 코드 또는 구성을 수정할 수는 없습니다.

자세한 정보는 Lambda 함수 상태을 참조하십시오.

Lambda: 함수가 보류 상태에서 멈춰 있음

오류: 함수가 몇 분 동안 *`Pending`* 상태에서 멈춰 있습니다.

함수가 6분 이상 `Pending` 상태에서 멈춰 있는 경우 다음 API 작업 중 하나를 호출하여 차단을 해제합니다.

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda는 보류 중인 작업을 취소하고 함수를 `Failed` 상태로 전환합니다. 그런 다음 다른 업데이트를 시도할 수 있습니다.

Lambda: 하나의 함수가 모든 동시성을 사용함

문제: 한 함수가 사용 가능한 모든 동시성을 사용하여 다른 함수에 병목을 유발합니다.

AWS 리전에서 AWS 계정의 사용 가능한 동시성을 풀로 나누려면 예약된 동시성을 사용합니다. 예약된 동시성은 함수가 항상 할당된 동시성으로 확장할 수 있으며 할당된 동시성을 초과하여 확장되지 않도록 합니다.

일반: 다른 계정 또는 서비스로 함수를 호출할 수 없음

문제: 함수를 직접 호출할 수 있지만, 다른 서비스 또는 계정에서 호출하는 경우 실행되지 않습니다.

기타 서비스 및 계정에 함수의 리소스 기반 정책에서 함수를 호출할 수 있는 권한을 부여합니다. 호출자가 다른 계정에 있는 경우, 해당 사용자도 함수를 호출할 수 있는 권한이 있어야 합니다.

일반: 함수 호출이 반복됨

문제: 함수가 루프에서 계속 호출됩니다.

이 문제는 일반적으로 함수가 해당 함수를 트리거하는 동일한 AWS 서비스의 리소스를 관리할 때 발생합니다. 예를 들어, 함수를 다시 호출하는 알림으로 구성된 Amazon Simple Storage Service(Amazon S3) 버킷에 객체를 저장하는 함수를 생성할 수 있습니다. 함수 실행을 중지하려면 사용 가능한 동시성을 0으로 줄입니다. 이렇게 하면 향후 모든 간접 호출이 제한됩니다. 그런 다음 재귀 호출을 일으킨 코드 경로 또는 구성 오류를 파악합니다. Lambda는 일부 AWS 서비스 및 SDK에 대한 재귀 루프를 자동으로 감지하고 중지합니다. 자세한 내용은 the section called “재귀 루프 감지” 단원을 참조하십시오.

Lambda: 프로비저닝된 동시성을 사용한 별칭 라우팅

문제: 별칭 라우팅 중에 프로비저닝된 동시성 초과 호출.

Lambda는 간단한 확률 모델을 사용하여 두 함수 버전 간에 트래픽을 분산시킵니다. 트래픽 수준이 낮으면 각 버전에서 구성된 트래픽과 실제 트래픽의 비율 간에 큰 차이가 나타날 수 있습니다. 함수가 프로비저닝된 동시성을 사용하는 경우 별칭 라우팅이 활성 상태인 동안 더 많은 수의 프로비전된 동시성 인스턴스를 구성하여 초과(spillover) 호출을 방지할 수 있습니다.

Lambda: 프로비저닝된 동시성으로 콜드 스타트

문제: 프로비저닝된 동시성을 활성화한 후 콜드 스타트가 표시됩니다.

함수에 대한 동시 실행 수가 프로비저닝된 동시성의 구성된 수준보다 작거나 이와 같으면 콜드 스타트가 없어야 합니다. 프로비저닝된 동시성이 정상적으로 작동하는지 확인하려면 다음을 수행하세요.

- 함수 버전 또는 별칭에서 프로비저닝된 동시성이 활성화되어 있는지 확인합니다.

> **Note**
>
> 게시되지 않은 버전의 함수($LATEST)에서는 프로비저닝된 동시성을 구성할 수 없습니다.

- 트리거가 올바른 함수 버전 또는 별칭을 호출하는지 확인합니다. 예를 들어, Amazon API Gateway를 사용하는 경우 API Gateway가 $LATEST가 아닌 프로비저닝된 동시성을 사용하여 함수 버전 또는 별칭을 호출하는지 확인합니다. 프로비저닝된 동시성이 사용되고 있는지 확인하려면 ProvisionedConcurrencyInvocations Amazon CloudWatch 지표를 살펴보면 됩니다. 0이 아닌 값은 함수가 초기화된 실행 환경에서 호출을 처리하고 있음을 나타냅니다.
- ProvisionedConcurrencySpilloverInvocations CloudWatch 지표를 살펴봐서 함수 동시성이 프로비저닝된 동시성의 구성된 수준을 초과하는지 확인합니다. 0이 아닌 값은 프로비저닝된 모든 동시성이 사용 중이고 일부 호출이 콜드 스타트로 발생했음을 나타냅니다.
- 호출 빈도(초당 요청 수)를 확인하세요. 프로비저닝된 동시성이 있는 함수의 최대 비율은 프로비저닝된 동시성별 초당 10개의 요청입니다. 예를 들어, 100 개의 프로비저닝된 동시성으로 구성된 함수는 초당 1,000개의 요청을 처리할 수 있습니다. 호출 비율이 초당 1,000개의 요청을 초과하면 일부 콜드 스타트가 발생할 수 있습니다.

Lambda: 새 버전으로 콜드 스타트

문제: 함수의 새 버전을 배포하는 동안 콜드 스타트가 나타납니다.

함수 별칭을 업데이트하면 Lambda는 별칭에 구성된 가중치를 기반으로 프로비저닝된 동시성을 새 버전으로 자동으로 이동시킵니다.

오류: KMSDisabledException: 사용된 KMS 키가 비활성화되어 있으므로 Lambda에서 환경 변수의 암호화를 해제할 수 없음. 해당 함수의 KMS 키 설정을 확인해야 함

AWS Key Management Service(AWS KMS) 키가 비활성화되어 있거나 Lambda에서 이 키를 사용할 수 있도록 하는 권한 부여가 취소된 경우 이 오류가 발생할 수 있습니다. 권한 부여가 누락된 경우 다른 키를 사용하도록 함수를 구성합니다. 그런 다음 사용자 지정 키를 다시 할당하여 권한 부여를 다시 만듭니다.

EFS: 함수가 EFS 파일 시스템을 마운트할 수 없음

오류: EFSMountFailureException: 이 함수는 액세스 포인트 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd를 사용하여 EFS 파일 시스템을 마운트할 수 없습니다.

함수의 파일 시스템에 대한 마운트 요청이 거부되었습니다. 함수의 권한을 확인하고 해당 파일 시스템 및 액세스 포인트가 존재하며 사용할 준비가 되었는지 확인합니다.

EFS: 함수가 EFS 파일 시스템에 연결할 수 없음

오류: EFSMountConnectivityException: 이 함수는 액세스 포인트 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd를 사용하여 Amazon EFS 파일 시스템에 연결할 수 없습니다. 네트워크 구성을 확인하고 다시 시도하세요.

함수가 NFS 프로토콜(TCP 포트 2049)을 사용하여 함수의 파일 시스템에 대한 연결을 설정할 수 없습니다. VPC 서브넷에 대한 보안 그룹 및 라우팅 구성을 확인합니다.

함수의 VPC 구성 설정을 업데이트한 후 이러한 오류가 발생하면 파일 시스템을 탑재 해제했다가 다시 탑재해 봅니다.

EFS: 시간 초과로 인해 함수가 EFS 파일 시스템을 마운트할 수 없음

오류: EFSMountTimeoutException: 이 함수는 마운트 시간 초과로 인해 액세스 포인트 {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd}를 사용하여 EFS 파일 시스템을 마운트할 수 없습니다.

함수가 함수의 파일 시스템에 연결할 수 있었지만 마운트 작업 시간이 초과되었습니다. 잠시 후 다시 시도를 하고, 함수의 동시성을 제한하여 파일 시스템의 부하를 줄이기 방법을 고려합니다.

Lambda: Lambda가 시간이 너무 오래 걸리는 IO 프로세스를 탐지함

EFSIOException: Lambda가 너무 오래 걸리는 IO 프로세스를 탐지했기 때문에 이 함수 인스턴스가 중지되었습니다.

이전 호출이 시간 초과되어 Lambda가 함수 핸들러를 종료할 수 없었습니다. 이 문제는 연결된 파일 시스템에 버스트 크레딧이 부족하여 기본 처리량이 충분하지 않은 경우에 발생할 수 있습니다. 처리량을 늘리려면 파일 시스템의 크기를 늘리거나 프로비저닝된 처리량을 사용할 수 있습니다. 자세한 정보는 처리량을 참조하십시오.

Lambda의 실행 문제 해결

Lambda 런타임에서 함수 코드를 실행하면 이벤트가 일정 시간 동안 처리 중인 함수의 인스턴스에서 처리되거나 새 인스턴스를 초기화해야 할 수 있습니다. 함수 초기화 동안 핸들러 코드가 이벤트를 처리할 때 또는 함수가 응답을 반환하거나 반환하지 않을 때 오류가 발생할 수 있습니다.

함수 실행 오류는 코드, 함수 구성, 다운스트림 리소스 또는 권한 문제로 인해 발생할 수 있습니다. 함수를 직접 호출하면 Lambda의 응답에 함수 오류가 표시됩니다. 이벤트 소스 매핑으로 또는 다른 서비스를 통해 함수를 비동기적으로 호출하는 경우 로그, 배달 못한 편지 대기열 또는 on-failure 대상에 오류가 있을 수 있습니다. 오류 처리 옵션 및 재시도 동작은 함수를 호출하는 방법과 오류 유형에 따라 다릅니다.

함수 코드 또는 Lambda 런타임에서 오류를 반환하면 Lambda의 응답에서 상태 코드는 200 OK입니다. 응답에 오류가 있으면 `X-Amz-Function-Error`라는 헤더로 표시됩니다. 400 및 500 시리즈 상태 코드는 호출 오류에 예약되어 있습니다.

Lambda: 실행 시간이 너무 오래 걸림

문제: 함수 실행에 너무 많은 시간이 소요됩니다.

코드가 로컬 시스템보다 Lambda에서 실행하는 데 더 오래 걸리는 경우, 함수에 사용할 수 있는 메모리 또는 처리 성능으로 인해 제한될 수 있습니다. 메모리와 CPU를 모두 관리하려면 추가 메모리를 사용하여 함수를 구성하세요.

Lambda: 로그 또는 추적이 나타나지 않음

문제: CloudWatch Logs에 로그가 나타나지 않습니다.

문제: 추적이 AWS X-Ray에 나타나지 않습니다.

함수는 CloudWatch Logs 및 X-Ray를 호출할 수 있는 권한이 필요합니다. 권한을 부여하려면 실행 역할을 업데이트하세요. 다음 관리형 정책을 추가하여 로그 및 추적을 활성화합니다.

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

함수에 권한을 추가할 때 코드나 구성에 대한 간단한 업데이트도 수행하세요. 그러면 자격 증명이 오래된 함수의 인스턴스 실행이 중지 및 대체됩니다.

> **Note**
>
> 함수 호출 후 로그가 표시되는 데 5~10분 정도 걸릴 수 있습니다.

Lambda: 일부 함수 로그가 표시되지 않음

문제: 권한이 올바른 경우에도 CloudWatch Logs에서 함수 로그가 누락됨

AWS 계정에서 CloudWatch Logs 할당량 한도에 도달하면 CloudWatch는 함수 로깅을 제한합니다. 이 경우 함수가 출력하는 일부 로그는 CloudWatch Logs에 표시되지 않을 수 있습니다.

함수의 로그 출력 속도가 높아 Lambda에서 처리할 수 없는 경우에도 CloudWatch Logs에 로그 출력이 나타나지 않을 수 있습니다. Lambda에서 함수의 로그 생성 속도에 맞춰 CloudWatch로 로그를 전송할 수 없는 경우 로그를 삭제하여 함수 실행 속도가 느려지는 것을 방지합니다.

함수가 JSON 형식의 로그를 사용하도록 구성된 경우 Lambda는 로그를 삭제하면 CloudWatch Logs에 `logsDropped` 이벤트를 보내려고 합니다. 하지만 CloudWatch가 함수의 로깅을 제한하면 해당 이벤트가 CloudWatch Logs에 도달하지 않을 수 있으므로 Lambda가 로그를 삭제할 때 레코드가 항상 표시되는 것은 아닙니다.

AWS 계정에서 CloudWatch Logs 할당량 한도에 도달했는지 확인하려면 다음을 수행합니다.

1. Service Quotas 콘솔을 엽니다.
2. 탐색 창에서 AWS 서비스를 선택합니다.
3. AWS 서비스 목록에서 Amazon CloudWatch Logs를 검색합니다.
4. 서비스 할당량 목록에서 `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second`, `PutLogEvents throttle limit in transactions per second` 할당량을 선택하여 사용률을 확인합니다.

또한 계정 사용률이 이러한 할당량에 지정한 한도를 초과할 경우 알리도록 CloudWatch 경보를 설정할 수 있습니다. 자세한 내용은 정적 임곗값을 기반으로 CloudWatch 경보 생성을 참조하세요.

CloudWatch Logs의 기본 할당량 한도가 사용 사례에 충분하지 않은 경우 할당량 증가를 요청할 수 있습니다.

Lambda: 실행이 완료되기 전에 함수가 반환됨

문제: (Node.js) 코드 실행을 완료하기 전에 함수가 반환됨

AWS SDK를 포함하는 많은 라이브러리가 비동기적으로 작동합니다. 네트워크를 호출하거나 응답을 기다려야 하는 다른 작업을 수행할 때 라이브러리는 백그라운드에서 작업의 진행 상황을 추적하는 promise라는 객체를 반환합니다.

promise가 응답으로 확인될 때까지 기다리려면 `await` 키워드를 사용하세요. 이렇게 하면 promise가 응답을 포함하는 객체로 확인될 때까지 핸들러 코드가 실행되지 않습니다. 코드에서 응답의 데이터를 사용할 필요가 없으면 promise를 런타임에 직접 반환 할 수 있습니다.

일부 라이브러리는 promise를 반환하지 않지만, 반환하는 코드로 래핑될 수 있습니다. 자세한 내용은 Node.js에서 Lambda 함수 핸들러 정의 섹션을 참조하세요.

AWS SDK: 버전 및 업데이트

문제: 런타임에 포함된 AWS SDK가 최신 버전이 아님

문제: 런타임에 포함된 AWS SDK가 자동으로 업데이트됨

스크립팅 언어의 런타임에는 AWS SDK가 포함되며 정기적으로 최신 버전으로 업데이트됩니다. 각 런타임의 현재 버전은 런타임 페이지에 나열됩니다. 최신 버전의 AWS SDK를 사용하거나 함수를 특정 버전으로 잠그려면 라이브러리를 함수 코드와 번들로 묶거나 Lambda 계층을 생성하면 됩니다. 종속성을 사용하여 배포 패키지를 만드는 방법에 대한 자세한 내용은 다음 항목을 참조하세요.

Node.js

- .zip 파일 아카이브를 사용하여 Node.js Lambda 함수 배포

Python

- Python Lambda 함수에 대한 .zip 파일 아카이브 작업

Ruby

- Ruby Lambda 함수에 대한 .zip 파일 아카이브 작업

Java

- .zip 또는 JAR 파일 아카이브를 사용하여 Java Lambda 함수 배포

Go

- .zip 파일 아카이브를 사용하여 Go Lambda 함수 배포

C#

- .zip 파일 아카이브를 사용하여 C# Lambda 함수를 빌드 및 배포

PowerShell

- .zip PowerShell 파일 아카이브와 함께 Lambda 함수 배포

Python: 라이브러리가 잘못 로드됨

문제: (Python) 일부 라이브러리는 배포 패키지에서 올바르게 로드되지 않음

C 또는 C++로 작성된 확장 모듈이 있는 라이브러리는 Lambda(Amazon Linux)와 동일한 프로세서 아키텍처를 사용하는 환경에서 컴파일해야 합니다. 자세한 내용은 Python Lambda 함수에 대한 .zip 파일 아카이브 작업 섹션을 참조하세요.

Lambda의 네트워킹 문제 해결

기본적으로 Lambda는 AWS 서비스 및 인터넷에 연결된 내부 가상 프라이빗 클라우드(VPC)에서 함수를 실행합니다. 로컬 네트워크 리소스에 액세스하려면 계정의 VPC에 연결하도록 함수를 구성할 수 있습니다. 이 기능을 사용하면 함수의 인터넷 액세스 및 Amazon Virtual Private Cloud(Amazon VPC) 리소스로의 네트워크 연결을 관리할 수 있습니다.

네트워크 연결 오류는 VPC의 라우팅 구성, 보안 그룹 규칙, AWS Identity and Access Management(IAM) 역할 권한 또는 네트워크 주소 변환(NAT) 문제로 야기되거나 IP 주소 또는 네트워크 인터페이스와 같은 리소스의 가용성 문제로 야기될 수 있습니다. 문제에 따라, 요청이 대상에 도달할 수 없는 경우 특정 오류나 시간 초과가 발생할 수 있습니다.

VPC: 함수가 인터넷 액세스를 잃거나 시간이 초과됨

문제: VPC 연결 후 Lambda 함수의 인터넷 액세스가 끊깁니다.

오류: 오류: connect ETIMEDOUT 176.32.98.189:443

오류: 오류: 10.00초 후 작업 시간 초과

오류: ReadTimuTerror: 읽기 시간이 초과되었습니다.(읽기 시간 초과=15)

VPC에 함수를 연결하면 모든 아웃바운드 요청이 VPC를 통과합니다. 인터넷에 연결하려면 함수의 서브넷에서 퍼블릭 서브넷의 NAT 게이트웨이로 아웃바운드 트래픽을 보내도록 VPC를 구성합니다. 자세한 내용과 샘플 VPC 구성은 을 참조하세요the section called “VPC 함수에 대한 인터넷 액세스”

일부 TCP 연결이 시간 초과되는 경우 패킷 조각화로 인한 것일 수 있습니다. Lambda는 TCP 또는 ICMP에 대한 IP 조각화를 지원하지 않으므로 Lambda 함수는 수신되는 조각화된 TCP 요청을 처리할 수 없습니다.

VPC: 함수에서 인터넷을 사용하지 않고 AWS 서비스에 액세스해야 함

문제: Lambda 함수에서 인터넷을 사용하지 않고 AWS 서비스에 액세스해야 합니다.

인터넷에 액세스할 수 없는 프라이빗 서브넷에서 AWS 서비스에 함수를 연결하려면 VPC 엔드포인트를 사용합니다.

VPC: 탄력적 네트워크 인터페이스 한도에 도달

오류: ENILimitReachedException: 함수의 VPC의 탄력적 네트워크 인터페이스 한도에 도달했습니다.

Lambda 함수를 VPC에 연결하면 Lambda는 함수에 연결된 서브넷과 보안 그룹의 각 조합에 대해 탄력적 네트워크 인터페이스를 생성합니다. 기본 서비스 할당량은 VPC당 250개의 네트워크 인터페이스입니다. 할당량 증가를 요청하려면 Service Quotas 콘솔을 사용합니다.

EC2: ‘lambda’ 유형 포함 탄력적 네트워크 인터페이스

오류 코드:: Client.OperationNotPermitted

오류 메시지: The security group can not be modified for this type of interface

Lambda에서 관리하는 탄력적 네트워크 인터페이스(ENI)를 수정하려고 하면 이 오류가 발생합니다. Lambda에서 생성한 탄력적 네트워크 인터페이스의 업데이트 작업을 위한 Lambda API에는 `ModifyNetworkInterfaceAttribute`가 포함되어 있지 않습니다.

AWS Lambda 애플리케이션

AWS Lambda 애플리케이션은 작업을 수행하는 데 함께 작동하는 Lambda 함수, 이벤트 소스 및 기타 리소스의 조합입니다. AWS CloudFormation 및 기타 도구를 사용해 애플리케이션의 구성 요소를 하나의 리소스로 배포해 관리할 수 있는 단일 패키지로 수집할 수 있습니다. 애플리케이션은 Lambda 프로젝트를 이동하기 쉽게 만들어 AWS CodePipeline, AWS CodeBuild 및 AWS Serverless Application Model 명령줄 인터페이스(AWS SAM CLI) 등과 같은 추가 개발자 도구와 통합이 가능하게 합니다.

AWS Serverless Application Repository는 몇 번만 클릭하여 간편하게 계정에 배포할 수 있는 Lambda 애플리케이션 모음을 제공합니다. 리포지토리에는 바로 사용할 수 있는 애플리케이션과 프로젝트의 시작 지점으로 사용할 수 있는 샘플이 들어 있습니다. 또한 자체 프로젝트를 제출해 포함할 수도 있습니다.

AWS CloudFormation에서는 애플리케이션의 리소스를 정의하는 템플릿을 생성할 수 있고 애플리케이션을 스택으로 관리할 수 있습니다. 애플리케이션 스택에서는 리소스를 더욱 안전하게 추가 또는 수정할 수 있습니다. 한 부분이라도 업데이트에 실패하면 AWS CloudFormation에서는 이전 구성으로 자동으로 롤백합니다. AWS CloudFormation 파라미터를 사용해 동일한 템플릿에서 애플리케이션을 위한 여러 환경을 생성할 수 있습니다. AWS SAM은 Lambda 애플리케이션 개발에 중점을 둔 간소화된 구문으로 AWS CloudFormation을 확장합니다.

AWS CLI 및 AWS SAM CLI는 Lambda 애플리케이션 스택을 관리하기 위한 명령줄 도구입니다. AWS CloudFormation API를 사용해 애플리케이션 스택을 관리하기 위한 명령 이외에 AWS CLI는 배포 패키지 업로드, 템플릿 업데이트 등의 작업을 간소화하는 상위 수준 명령을 지원합니다. AWS SAM CLI는 템플릿 확인, 로컬 테스트 및 CI/CD 시스템 통합을 포함하여 추가 기능을 제공합니다.

애플리케이션을 생성할 때 CodeCommit 또는 GitHub에 대한 AWS CodeStar 연결을 사용하여 Git 리포지토리를 만들 수 있습니다. CodeCommit을 사용하도록 설정하면 IAM 콘솔을 사용하여 사용자의 SSH 키와 HTTP 자격 증명을 관리할 수 있습니다. CodeConnections를 사용하면 GitHub 계정에 연결할 수 있습니다. 연결에 대한 자세한 내용은 개발자 도구 콘솔 사용자 가이드의 연결이란 무엇입니까?를 참조하세요.

Lambda 애플리케이션 설계에 대한 자세한 내용은 Serverless Land의 Application design을 참조하세요.

주제

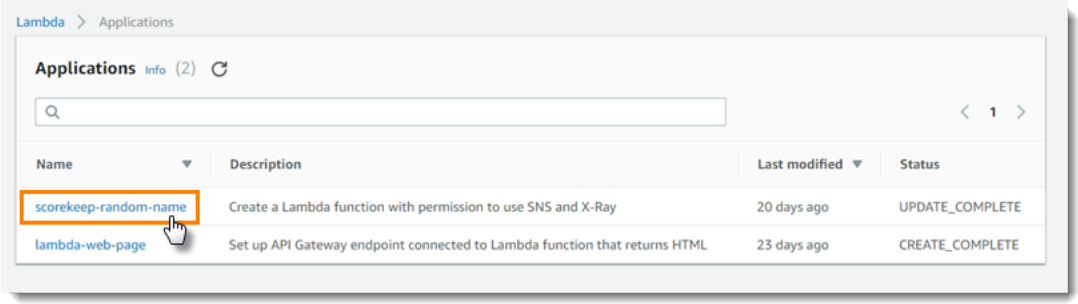
- AWS Lambda 콘솔에서 애플리케이션 관리
- Lambda 함수에 대한 롤링 배포 생성
- Kubernetes와 함께 Lambda 사용

AWS Lambda 콘솔에서 애플리케이션 관리

AWS Lambda 콘솔에서는 Lambda 애플리케이션을 모니터링 및 관리할 수 있습니다. 애플리케이션 메뉴에는 Lambda 함수와 함께 AWS CloudFormation 스택이 나열됩니다. 이 메뉴에는 AWS CloudFormation 콘솔, AWS CloudFormation, AWS Serverless Application Repository 또는 AWS CLI CLI를 사용하여 AWS SAM에서 실행한 스택이 포함되어 있습니다.

Lambda 애플리케이션을 보려면

1. Lambda 콘솔 애플리케이션 페이지를 엽니다.
2. 애플리케이션을 선택합니다.



개요에는 애플리케이션에 대한 다음 정보가 표시됩니다.

- AWS CloudFormation 템플릿 또는 SAM 템플릿 – 애플리케이션을 정의하는 템플릿.
- 리소스 – 애플리케이션 템플릿에서 정의하는 AWS 리소스. 애플리케이션의 Lambda 함수를 관리하려면 목록에서 함수 이름을 선택합니다.

애플리케이션 모니터링

Monitoring 탭에는 애플리케이션 내 리소스에 대한 집계 지표가 있는 Amazon CloudWatch 대시보드가 표시됩니다.

Lambda 애플리케이션을 모니터링하려면

1. Lambda 콘솔 애플리케이션 페이지를 엽니다.
2. 모니터링을 선택합니다.

기본적으로 Lambda 콘솔에는 기본 대시보드가 표시됩니다. 애플리케이션 템플릿에서 사용자 지정 대시보드를 정의해 이 페이지를 사용자 지정할 수 있습니다. 템플릿에 대시보드가 한 개 이상 포함되어

있는 경우 이 페이지에는 기본 대시보드가 아니라 사용자 지정 대시보드가 표시됩니다. 페이지 오른쪽 상단에 있는 드롭다운 메뉴를 사용해 대시보드 간에 전환할 수 있습니다.

사용자 지정 모니터링 대시보드

AWS::CloudWatch::Dashboard리소스 유형과 함께 애플리케이션 템플릿에 하나 이상의 Amazon CloudWatch 대시보드를 추가하여 애플리케이션 모니터링 페이지를 사용자 지정합니다. 다음 예에서는 `my-function` 함수의 호출 횟수를 그래프로 보여주는 단일 위젯이 있는 대시보드를 만듭니다.

Example 함수 대시보드 템플릿

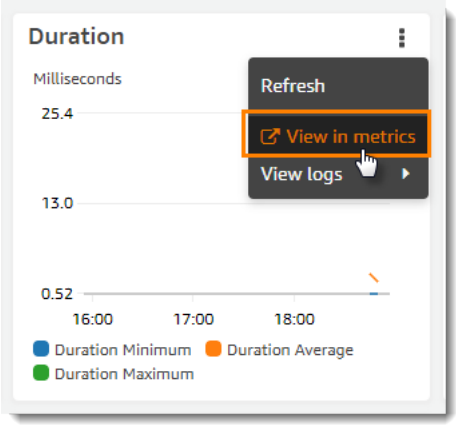
```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "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 콘솔의 기본 모니터링 대시보드에서 위젯에 대한 정의를 확인할 수 있습니다.

위젯 정의를 보려면

1. Lambda 콘솔 애플리케이션 페이지를 엽니다.
2. 표준 대시보드가 있는 애플리케이션을 선택합니다.
3. 모니터링을 선택합니다.
4. 위젯의 드롭다운 메뉴에서 지표에서 보기를 선택합니다.



5. [Source]를 선택합니다.

CloudWatch 대시보드 및 위젯 작성에 대한 자세한 내용은 Amazon CloudWatch API Reference의 대시보드 본문 구조 및 구문을 참조하십시오.

Lambda 함수에 대한 롤링 배포 생성

롤링 배포를 사용하여 Lambda 함수의 새 버전 도입과 관련된 위험을 제어할 수 있습니다. 롤링 배포에서 시스템은 자동으로 함수의 새 버전을 배포하고 새로운 버전으로 전송하는 트래픽의 양을 늘립니다. 트래픽 양과 증가 속도는 구성이 가능한 파라미터입니다.

롤링 배포는 AWS CodeDeploy 및 AWS SAM을 사용해 구성합니다. CodeDeploy는 Amazon EC2 및 AWS Lambda 같은 Amazon 컴퓨팅 플랫폼에 대한 애플리케이션 배포를 자동화하는 서비스입니다. 자세한 내용은 CodeDeploy란 무엇입니까?를 참조하세요. CodeDeploy를 사용하여 Lambda 함수를 배포하면 배포 상태를 쉽게 모니터링하고 문제가 발견되면 롤백을 시작할 수 있습니다.

AWS SAM은 서버리스 애플리케이션을 빌드하기 위한 오픈 소스 프레임워크입니다. AWS SAM 템플릿(YAML 형식)을 생성하여 롤링 배포에 필요한 구성 요소의 구성을 지정합니다. AWS SAM에서는 템플릿을 사용하여 구성 요소를 생성하고 구성합니다. 자세한 내용은 AWS SAM이란 무엇입니까?를 참조하세요.

롤링 배포에서 AWS SAM는 다음 작업을 수행합니다.

- Lambda 함수를 구성하고 별칭을 생성합니다.

  별칭 라우팅 구성은 롤링 배포 구현의 기반이 되는 기능으로,
- CodeDeploy 애플리케이션 및 배포 그룹을 생성합니다.

  배포 그룹은 롤링 배포 및 롤백(필요한 경우)을 관리합니다.
- Lambda 함수의 새 버전이 생성될 때 이를 감지합니다.
- 새 버전의 배포를 시작하도록 CodeDeploy를 트리거합니다.

AWS SAM Lambda 템플릿 예제

다음 예제는 간단한 롤링 배포를 위한 AWS SAM 템플릿을 보여줍니다.

```
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
```

이 템플릿은 다음 속성을 가진 `myDateTimeFunction`이라는 Lambda 함수를 정의합니다.

AutoPublishAlias

`AutoPublishAlias` 속성은 `live`라는 별칭을 생성합니다. 또한 AWS SAM 프레임워크는 함수에 대한 새 코드를 저장할 때 이를 자동으로 감지합니다. 그러면 프레임워크가 새 함수 버전을 게시하고 새 버전을 가리키도록 `live` 별칭을 업데이트합니다.

DeploymentPreference

`DeploymentPreference` 속성은 CodeDeploy 애플리케이션이 Lambda 함수의 원래 버전에서 새 버전으로 트래픽을 이동시키는 속도를 결정합니다. 값이 `Linear10PercentEvery2Minutes`면 트래픽의 10%가 2분마다 새 버전으로 새롭게 이동됩니다.

사전 정의된 배포 구성의 목록은 배포 구성을 참조하세요.

Lambda 함수와 함께 CodeDeploy를 사용하는 방법에 대한 자세한 자습서는 CodeDeploy를 사용하여 업데이트된 Lambda 함수 배포를 참조하세요.

Kubernetes와 함께 Lambda 사용

AWS Controllers for Kubernetes(ACK) 또는 Crossplane을 사용하여 Kubernetes API로 Lambda 함수를 배포하고 관리할 수 있습니다.

AWS Controllers for Kubernetes(ACK)

ACK를 사용하여 Kubernetes API에서 AWS 리소스를 배포하고 관리할 수 있습니다. ACK를 통해 Lambda, 아마존 엘라스틱 컨테이너 레지스트리 (Amazon ECR), 아마존 심플 스토리지 서비스 (Amazon S3), 아마존 등의 AWS 서비스를 위한 오픈 소스 사용자 지정 컨트롤러를 AWS 제공합니다. SageMaker 지원되는 각 AWS 서비스에는 자체 사용자 지정 컨트롤러가 있습니다. Kubernetes 클러스터에서 사용하려는 각 AWS 서비스에 대한 컨트롤러를 설치합니다. 그런 다음 사용자 지정 리소스 정의(CRD)를 생성하여 AWS 리소스를 정의합니다.

ACK 컨트롤러를 설치하려면 Helm 3.8 이상을 사용하는 것이 좋습니다. 모든 ACK 컨트롤러에는 컨트롤러, CRD 및 Kubernetes RBAC 규칙을 설치하는 자체 차트 Helm이 함께 제공됩니다. 자세한 내용은 ACK 설명서의 Install an ACK Controller를 참조하세요.

ACK 사용자 지정 리소스를 생성한 후에는 다른 기본 제공 Kubernetes 객체처럼 사용할 수 있습니다. 예를 들어, kubectl 등의 선호하는 Kubernetes 도구 체인을 사용하여 Lambda 함수를 배포하고 관리할 수 있습니다.

다음은 ACK를 통해 Lambda 함수를 프로비저닝하는 몇 가지 사용 사례입니다.

- 조직에서는 서비스 계정에 대한 IAM 역할과 역할 기반 액세스 제어(RBAC)를 사용하여 권한 경계를 생성합니다. ACK를 사용하면 새로운 사용자와 정책을 생성할 필요 없이 Lambda에 대해 이 보안 모델을 재사용할 수 있습니다.
- 조직에는 쿠버네티스 매니페스트를 사용하여 Amazon Elastic Kubernetes Service (Amazon EKS) 클러스터에 리소스를 배포하는 DevOps 프로세스가 있습니다. ACK를 사용하면 별도의 인프라를 코드 템플릿으로 생성하지 않고도 매니페스트를 사용하여 Lambda 함수를 프로비저닝할 수 있습니다.

ACK 사용에 대한 자세한 내용은 ACK 설명서의 Lambda 자습서를 참조하세요.

Crossplane

Crossplane은 Kubernetes를 사용하여 클라우드 인프라 리소스를 관리하는 오픈 소스 CNCF(Cloud Native Computing Foundation) 프로젝트입니다. Crossplane로 개발자는 인프라의 복잡성을 이해할 필요 없이 인프라를 요청할 수 있습니다. 플랫폼 팀이 인프라 프로비저닝 및 관리 방법에 대한 통제권을 갖습니다.

Crossplane을 사용하면 kubectl과 같은 선호하는 Kubernetes 도구 체인과 Kubernetes에 매니페스트를 배포할 수 있는 모든 CI/CD 파이프라인을 사용하여 Lambda 함수를 배포하고 관리할 수 있습니다. 다음은 Crossplane을 통해 Lambda 함수를 프로비저닝하는 몇 가지 사용 사례입니다.

- 조직에서 Lambda 함수에 올바른 태그가 있는지 확인하여 규정 준수를 시행하려고 합니다. 플랫폼 팀이 Crossplane Compositions를 사용하여 API 추상화를 통해 이 정책을 정의할 수 있습니다. 그러면 개발자가 이러한 추상화를 사용하여 태그가 있는 Lambda 함수를 배포할 수 있습니다.
- GitOps 프로젝트는 쿠버네티스와 함께 사용합니다. 이 모델에서 Kubernetes는 git 리포지토리(원하는 상태)를 클러스터 내에서 실행되는 리소스(현재 상태)와 지속적으로 조정합니다. 차이가 있는 경우 GitOps 프로세스가 자동으로 클러스터를 변경합니다. Crossplane을 통해 CRD 및 컨트롤러와 같은 친숙한 쿠버네티스 도구 및 개념을 사용하여 Lambda 함수를 배포하고 관리하는 데 Kubernetes와 GitOps 함께 사용할 수 있습니다.

Lambda와 함께 Crossplane을 사용하는 방법에 대해 자세히 알아보려면 다음을 참조하세요.

- AWS Blueprints for Crossplane: 이 리포지토리에는 Crossplane을 사용하여 Lambda 함수를 비롯한 AWS 리소스를 배포하는 방법에 대한 예제가 포함되어 있습니다.

  > **Note**
  >
  > AWS Blueprints for Crossplane는 현재 개발 중이므로 프로덕션에 사용해서는 안 됩니다.

- Deploying Lambda with Amazon EKS and Crossplane: 이 동영상에서는 개발자와 플랫폼 관점에서 설계를 살펴보면서 Crossplane을 사용하여 AWS 서버리스 아키텍처를 배포하는 고급 예제를 보여줍니다.

Lambda 샘플 애플리케이션

이 안내서의 GitHub 리포지토리에는 다양한 언어 및 AWS 서비스를 사용하는 방법을 보여주는 샘플 애플리케이션이 들어 있습니다. 각 샘플 애플리케이션에는 간편한 배포 및 정리를 위한 스크립트, AWS SAM 템플릿 및 지원 리소스가 포함되어 있습니다.

Node.js

Node.js의 샘플 Lambda 애플리케이션

- blank-nodejs – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Node.js 함수입니다.
- nodejs-apig - API Gateway의 이벤트를 처리하고 HTTP 응답을 반환하는 퍼블릭 API 엔드포인트가 있는 함수입니다.
- efs-nodejs – Amazon VPC에서 Amazon EFS 파일 시스템을 사용하는 함수입니다. 이 샘플에는 Lambda와 함께 사용하도록 구성된 VPC, 파일 시스템, 마운트 대상 및 액세스 포인트가 포함되어 있습니다.

Python

Python의 샘플 Lambda 애플리케이션

- blank-python – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Python 함수입니다.

Ruby

Ruby의 샘플 Lambda 애플리케이션

- blank-ruby – 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 Ruby 함수입니다.
- AWS Lambda용 Ruby 코드 샘플 – AWS Lambda와 상호 작용하는 방법을 보여주는 Ruby로 작성된 코드 샘플입니다.

Java

Java의 샘플 Lambda 애플리케이션

- java17-examples – Java 레코드를 사용하여 입력 이벤트 데이터 객체를 나타내는 방법을 보여주는 Java 함수입니다.
- java-basic – 단위 테스트 및 변수 로깅 구성을 사용하는 최소한의 Java 함수 모음입니다.
- java - Amazon API Gateway, Amazon SQS 및 Amazon Kinesis와 같은 다양한 서비스의 이벤트를 처리하는 방법에 대한 스켈레톤 코드가 포함된 Java 함수 모음입니다. 이러한 함수는 최신 버전의 aws-lambda-java-events 라이브러리(3.0.0 이상)를 사용합니다. 이러한 예는 AWS SDK를 종속 항목으로 요구하지 않습니다.
- s3-java – Amazon S3의 알림 이벤트를 처리하고 JCL(Java Class Library)을 사용하여 업로드된 이미지 파일의 썸네일을 생성하는 Java 함수입니다.
- API Gateway를 사용하여 Lambda 함수 호출 — 직원 정보가 포함된 Amazon DynamoDB 테이블을 스캔하는 Java 함수입니다. 이후 Amazon 간편 알림 서비스를 사용하여 직원들에게 근무 기념일을 축하하는 문자 메시지를 보냅니다. 이 예제에서는 API Gateway를 사용하여 함수를 호출합니다.

Lambda에서 인기 있는 Java 프레임워크 실행

- spring-cloud-function-samples – Spring Cloud 함수 프레임워크를 사용하여 AWS Lambda 함수를 생성하는 방법을 보여주는 Spring 예제입니다.
- 서버리스 Spring Boot 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 일반적인 Spring Boot 애플리케이션을 설정하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다.
- 서버리스 Micronaut 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 Micronaut을 사용하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다. Micronaut/Lambda 설명서에서 자세히 알아보세요.
- 서버리스 Quarkus 애플리케이션 데모 - SnapStart를 사용하거나 사용하지 않는 관리형 Java 런타임에서 Quarkus를 사용하거나 사용자 지정 런타임이 있는 GraalVM 네이티브 이미지로 설정하는 방법을 보여주는 예제입니다. Quarkus/Lambda 설명서 및 Quarkus/SnapStart 설명서에서 자세히 알아보세요.

Go

Lambda에서는 다음과 같은 Go 런타임용 샘플 애플리케이션을 제공합니다.

Go의 샘플 Lambda 애플리케이션

- go-al2 – 퍼블릭 IP 주소를 반환하는 hello world 함수입니다. 이 앱은 `provided.al2` 사용자 지정 런타임을 사용합니다.
- blank-go – Lambda의 Go 라이브러리, 로깅, 환경 변수 및 AWS SDK를 사용하는 방법을 보여주는 Go 함수입니다. 이 앱은 `go1.x` 런타임을 사용합니다.

C#

C#의 샘플 Lambda 애플리케이션

- blank-csharp – Lambda의 .NET 라이브러리, 로깅, 환경 변수, AWS X-Ray 추적, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여주는 C# 함수입니다.
- blank-csharp-with-layer – .NET CLI를 사용하여 함수의 종속 항목을 패키징하는 계층을 생성하는 C# 함수입니다.
- ec2-spot – Amazon EC2에서 스팟 인스턴스 요청을 관리하는 함수입니다.

PowerShell

Lambda는 PowerShell에 대해 다음과 같은 샘플 애플리케이션을 제공합니다.

- blank-powershell – 로깅, 환경 변수 및 AWS SDK를 사용하는 방법을 보여주는 PowerShell 함수입니다.

샘플 애플리케이션을 배포하려면 해당 README 파일의 지침을 따릅니다. 애플리케이션의 아키텍처 및 사용 사례에 대해 자세히 알아보려면 이 장의 주제를 참조하세요.

주제

- AWS Lambda용 빈 함수 샘플 애플리케이션

AWS Lambda용 빈 함수 샘플 애플리케이션

빈 함수 샘플 애플리케이션은 Lambda API를 호출하는 함수와 함께 Lambda의 일반적인 작업을 보여줍니다. 그리고 로깅, 환경 변수, AWS X-Ray 추적, 계층, 단위 테스트 및 AWS SDK를 사용하는 방법을 보여줍니다. 이 애플리케이션을 탐색하여 프로그래밍 언어로 Lambda 함수를 빌드하는 방법을 배우거나 애플리케이션을 자체 프로젝트의 시작점으로 사용하세요.

이 샘플 애플리케이션의 변형은 다음 언어에 사용할 수 있습니다.

변형

- Node.js – blank-nodejs.
- Python – blank-python.
- Ruby – blank-ruby.
- Java – blank-java.
- Go – blank-go.
- C# – blank-csharp.
- PowerShell – blank-powershell.

이 주제의 예제에서는 Node.js 버전의 코드를 강조하지만 세부 정보는 일반적으로 모든 변형에 적용됩니다.

AWS CLI와 AWS CloudFormation를 사용하여 단 몇 분 내에 샘플을 배포할 수 있습니다. README의 지침에 따라 샘플을 다운로드하고 구성하여 해당 계정에 배포하십시오.

단원

- 아키텍처 및 핸들러 코드
- AWS CloudFormation 및 AWS CLI를 사용한 배포 자동화
- AWS X-Ray를 사용한 계측
- 계층을 사용한 종속성 관리

아키텍처 및 핸들러 코드

샘플 애플리케이션은 함수 코드, AWS CloudFormation 템플릿 및 지원 리소스로 구성됩니다. 샘플을 배포할 때 다음 AWS 서비스를 사용합니다.

- AWS Lambda – 함수 코드를 실행하여 CloudWatch Logs로 전송하고 추적 데이터를 X-Ray로 전송합니다. 또한 이 함수는 Lambda API를 호출하여 현재 리전의 계정 할당량 및 사용량에 대한 세부 정보를 가져옵니다.
- AWS X-Ray – 추적 데이터를 수집하고, 검색을 위해 추적을 인덱스화하고, 서비스 맵을 생성합니다.
- Amazon CloudWatch – 로그 및 지표를 저장합니다.
- AWS Identity and Access Management(IAM) – 권한을 부여합니다.
- Amazon Simple Storage Service(Amazon S3) – 배포 중에 함수의 배포 패키지를 저장합니다.
- AWS CloudFormation – 애플리케이션 리소스를 생성하고 함수 코드를 배포합니다.

각 서비스마다 표준 요금이 적용됩니다. 자세한 내용은 AWS 요금을 참조하세요.

함수 코드는 이벤트를 처리하기 위한 기본 워크플로우를 보여줍니다. 핸들러는 Amazon Simple Queue Service(Amazon SQS) 이벤트를 입력으로 받아 해당 이벤트에 포함된 레코드를 반복하여 각 메시지의 내용을 기록합니다. 이때 이벤트 콘텐츠, 컨텍스트 객체 및 환경 변수를 기록합니다. 그런 다음 AWS SDK를 호출하고 응답을 Lambda 런타임에 다시 전달합니다.

Example blank-nodejs/function/index.js – 핸들러 코드

```
// 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);
};
```

샘플 애플리케이션에는 이벤트를 보내는 Amazon SQS 대기열이 포함되지 않지만 Amazon SQS(event.json)의 이벤트를 사용하여 이벤트가 처리되는 방법을 보여줍니다. 애플리케이션에 Amazon SQS 대기열을 추가하려면 Amazon SQS에서 Lambda 사용 단원을 참조하세요.

AWS CloudFormation 및 AWS CLI를 사용한 배포 자동화

샘플 애플리케이션의 리소스는 AWS CloudFormation 템플릿에 정의되고 AWS CLI와 함께 배포됩니다. 이 프로젝트에는 애플리케이션을 설정, 배포, 호출 및 삭제하는 프로세스를 자동화하는 간단한 셸 스크립트가 포함되어 있습니다.

이 애플리케이션 템플릿은 AWS Serverless Application Model(AWS SAM) 리소스 유형을 사용하여 모델을 정의합니다. AWS SAM은 실행 역할, API 및 기타 리소스의 정의를 자동화하여 서버리스 애플리케이션에 대한 템플릿 작성을 단순화합니다.

템플릿은 애플리케이션 스택 리소스를 정의합니다. 여기에는 함수, 실행 역할 및 함수의 라이브러리 종속성을 제공하는 Lambda 계층이 포함됩니다. 스택에는 배포 또는 CloudWatch Logs 로그 그룹 중에 AWS CLI에서 사용하는 버킷이 포함되지 않습니다.

Example blank-nodejs/template.yml – 서버리스 리소스

```
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
```

애플리케이션을 배포할 때 AWS CloudFormation은 AWS SAM 변환을 템플릿에 적용하여 AWS CloudFormation 및 `AWS::Lambda::Function`과 같은 표준 유형의 `AWS::IAM::Role` 템플릿을 생성합니다.

Example 처리된 템플릿

```
{
  "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"
          }
        ],
```

이 예제에서 `Code` 속성은 Amazon S3 버킷의 객체를 지정합니다. 이는 프로젝트 템플릿의 `CodeUri` 속성에 있는 로컬 경로에 해당합니다.

```
      CodeUri: function/.
```

프로젝트 파일을 Amazon S3에 업로드하기 위해 배포 스크립트는 AWS CLI의 명령을 사용합니다. `cloudformation package` 명령은 템플릿을 사전 처리하고, 아티팩트를 업로드하고, 로컬 경로를 Amazon S3 객체 위치로 바꿉니다. `cloudformation deploy` 명령은 처리된 템플릿을 AWS CloudFormation 변경 세트와 함께 배포합니다.

Example blank-nodejs/3-deploy.sh – 패키지 및 배포

```
#!/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
```

이 스크립트를 처음 실행하면 AWS CloudFormation라는 `blank-nodejs` 스택이 생성됩니다. 함수 코드 또는 템플릿을 변경한 경우 다시 실행하여 스택을 업데이트할 수 있습니다.

정리 스크립트(blank-nodejs/5-cleanup.sh)는 스택을 삭제하고 선택적으로 배포 버킷 및 함수 로그를 삭제합니다.

AWS X-Ray를 사용한 계측

샘플 함수는 AWS X-Ray를 사용한 추적용으로 구성되어 있습니다. 추적 모드가 활성인 상태에서 Lambda는 호출의 하위 집합에 대한 타이밍 정보를 기록하여 X-Ray에 전송합니다. X-Ray는 해당 데이터를 처리하여 하나의 클라이언트 노드와 두 개의 서비스 노드를 보여주는 서비스 맵을 생성합니다.

첫 번째 서비스 노드(`AWS::Lambda`)는 호출 요청을 검증하고 해당 요청을 함수에 전송하는 Lambda 서비스를 나타냅니다. 두 번째 노드 `AWS::Lambda::Function`은 함수 자체를 나타냅니다.

추가 세부 정보를 기록하기 위해 샘플 함수는 X-Ray SDK를 사용합니다. X-Ray SDK는 함수 코드를 최소한으로 변경하면서 AWS SDK에서 AWS 서비스에 대한 호출에 대한 세부 정보를 기록합니다.

Example blank-nodejs/function/index.js – 계측

```
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 클라이언트를 계측하면 서비스 맵에 노드가 추가되고 추적에 대한 자세한 내용이 추가됩니다. 이 예제에서 서비스 맵은 현재 리전의 스토리지 및 동시성 사용에 대한 세부 정보를 얻기 위해 Lambda API를 호출하는 샘플 함수를 보여줍니다.

추적은 함수 초기화, 호출 및 오버헤드에 대한 하위 세그먼트와 함께 호출에 대한 타이밍 세부 정보를 보여줍니다. 호출 하위 세그먼트에는 AWS API 작업에 대한 `GetAccountSettings` SDK 호출의 하위 세그먼트가 있습니다.

X-Ray SDK 및 기타 라이브러리를 함수의 배포 패키지에 포함하거나 Lambda 계층에 별도로 배포할 수 있습니다. Node.js, Ruby 및 Python의 경우 Lambda 런타임은 실행 환경의 AWS SDK를 포함합니다.

계층을 사용한 종속성 관리

라이브러리를 로컬로 설치하고 Lambda에 업로드하는 배포 패키지에 포함시킬 수 있지만 단점이 있습니다. 파일 크기가 클수록 배포 시간이 늘어나고 Lambda 콘솔에서 함수 코드의 변경 사항을 테스트하지 못할 수 있습니다. 배포 패키지를 작게 유지하고 변경되지 않은 종속성 업로드를 방지하기 위해 샘플 앱은 Lambda 계층을 생성하고 해당 계층을 함수와 연결합니다.

Example blank-nodejs/template.yml – 종속성 계층

```
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 스크립트는 npm과 함께 함수의 종속성을 설치하고 Lambda 런타임에 필요한 구조와 함께 폴더에 배치합니다.

Example 2-build-layer.sh – 계층 준비

```
#!/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/
```

샘플 애플리케이션을 처음 배포하면 AWS CLI가 함수 코드와 별도로 계층을 패키지하고 둘 다 배포합니다. 후속 배포의 경우 lib 폴더 콘텐츠가 변경된 경우에는 계층 아카이브만 업로드됩니다.

AWS SDK에서 Lambda 사용

다양한 프로그래밍 언어에 대해 AWS 소프트웨어 개발 키트(SDK)을 사용할 수 있습니다. 각 SDK는 개발자가 선호하는 언어로 애플리케이션을 쉽게 구축할 수 있도록 하는 API, 코드 예시 및 설명서를 제공합니다.

SDK 설명서	코드 예시
AWS SDK for C++	AWS SDK for C++ 코드 예시
AWS CLI	AWS CLI 코드 예시
AWS SDK for Go	AWS SDK for Go 코드 예시
AWS SDK for Java	AWS SDK for Java 코드 예시
AWS SDK for JavaScript	AWS SDK for JavaScript 코드 예시
AWS SDK for Kotlin	AWS SDK for Kotlin 코드 예시
AWS SDK for .NET	AWS SDK for .NET 코드 예시
AWS SDK for PHP	AWS SDK for PHP 코드 예시
AWS Tools for PowerShell	Tools for PowerShell 코드 예시
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) 코드 예시
AWS SDK for Ruby	AWS SDK for Ruby 코드 예시
AWS SDK for Rust	AWS SDK for Rust 코드 예시
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP 코드 예시
AWS SDK for Swift	AWS SDK for Swift 코드 예시

Lambda에 대한 구체적인 예는 AWS SDK를 사용한 Lambda용 코드 예제 단원을 참조하세요.

가용성 예제

필요한 예제를 찾을 수 없습니까? 이 페이지 하단의 피드백 제공 링크를 사용하여 코드 예시를 요청하세요.

AWS SDK를 사용한 Lambda용 코드 예제

다음 코드 예제에서는 Lambda를 AWS 소프트웨어 개발 키트(SDK)와 함께 사용하는 방법을 보여줍니다.

작업은 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 작업은 개별 서비스 함수를 호출하는 방법을 보여 주며 관련 시나리오와 교차 서비스 예시에서 컨텍스트에 맞는 작업을 볼 수 있습니다.

시나리오는 동일한 서비스 내에서 여러 함수를 호출하여 특정 태스크를 수행하는 방법을 보여주는 코드 예시입니다.

교차 서비스 예시는 여러 AWS 서비스 전반에서 작동하는 샘플 애플리케이션입니다.

AWS SDK 개발자 가이드 및 코드 예제의 전체 목록은 AWS SDK에서 Lambda 사용을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

시작하기

Hello Lambda

다음 코드 예제에서는 Lambda를 사용하여 시작하는 방법을 보여줍니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for .NET API 참조의 ListFunctions를 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

CMakeLists.txt CMake 파일의 코드입니다.

```
# 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 소스 파일의 코드입니다.

```
#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 세부 정보는 AWS SDK for C++ API 참조의 ListFunctions를 참조하십시오.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Go API 참조의 ListFunctions를 참조하십시오.

Java

SDK for Java 2.x

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Java 2.x API 참조의 ListFunctions를 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 ListFunctions를 참조하십시오.

코드 예시

- AWS SDK를 사용한 Lambda 작업
  - AWS SDK 또는 CLI와 함께 CreateAlias 사용
  - AWS SDK 또는 CLI와 함께 CreateFunction 사용
  - AWS SDK 또는 CLI와 함께 DeleteAlias 사용
  - AWS SDK 또는 CLI와 함께 DeleteFunction 사용
  - AWS SDK 또는 CLI와 함께 DeleteFunctionConcurrency 사용
  - AWS SDK 또는 CLI와 함께 DeleteProvisionedConcurrencyConfig 사용
  - AWS SDK 또는 CLI와 함께 GetAccountSettings 사용
  - AWS SDK 또는 CLI와 함께 GetAlias 사용
  - AWS SDK 또는 CLI와 함께 GetFunction 사용
  - AWS SDK 또는 CLI와 함께 GetFunctionConcurrency 사용
  - AWS SDK 또는 CLI와 함께 GetFunctionConfiguration 사용
  - AWS SDK 또는 CLI와 함께 GetPolicy 사용
  - AWS SDK 또는 CLI와 함께 GetProvisionedConcurrencyConfig 사용
  - AWS SDK 또는 CLI와 함께 Invoke 사용
  - AWS SDK 또는 CLI와 함께 ListFunctions 사용
  - AWS SDK 또는 CLI와 함께 ListProvisionedConcurrencyConfigs 사용
  - AWS SDK 또는 CLI와 함께 ListTags 사용
  - AWS SDK 또는 CLI와 함께 ListVersionsByFunction 사용
  - AWS SDK 또는 CLI와 함께 PublishVersion 사용
  - AWS SDK 또는 CLI와 함께 PutFunctionConcurrency 사용
  - AWS SDK 또는 CLI와 함께 PutProvisionedConcurrencyConfig 사용
  - AWS SDK 또는 CLI와 함께 RemovePermission 사용

- AWS SDK 또는 CLI와 함께 TagResource 사용
- AWS SDK 또는 CLI와 함께 UntagResource 사용
- AWS SDK 또는 CLI와 함께 UpdateAlias 사용
- AWS SDK 또는 CLI와 함께 UpdateFunctionCode 사용
- AWS SDK 또는 CLI와 함께 UpdateFunctionConfiguration 사용

- AWS SDK를 사용한 Lamda에 대한 시나리오
  - AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 확인
  - AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 마이그레이션
  - AWS SDK를 사용하여 Lamda 함수 생성 및 호출 시작하기
  - AWS SDK를 사용한 Amazon Cognito 사용자 인증 후 Lambda 함수를 사용하여 사용자 지정 활동 데이터 작성
- AWS SDK를 사용하는 Lambda의 서버리스 예제
  - Lambda 함수를 사용하여 Amazon RDS 데이터베이스에 연결
  - Kinesis 트리거에서 간접적으로 Lambda 함수 호출
  - DynamoDB 트리거에서 간접적으로 Lambda 함수 간접 호출
  - Amazon DocumentDB 트리거에서 간접적으로 Lambda 함수 호출
  - Amazon S3 트리거를 사용하여 Lambda 함수 호출
  - Amazon SNS 트리거를 사용하여 Lambda 함수 호출
  - Amazon SQS 트리거에서 간접적으로 Lambda 함수 호출
  - Kinesis 트리거로 Lambda 함수에 대한 배치 항목 실패 보고
  - DynamoDB 트리거로 Lambda 함수에 대한 배치 항목 실패 보고
  - Amazon SQS 트리거로 Lambda 함수에 대한 배치 항목 실패 보고
- AWS SDK를 사용한 Lambda용 교차 서비스 예제
  - COVID-19 데이터를 추적하는 API Gateway REST API 생성
  - 대출 라이브러리 REST API 생성
  - Step Functions를 사용하여 메신저 애플리케이션 생성
  - 사용자가 레이블을 사용하여 사진을 관리할 수 있는 사진 자산 관리 애플리케이션 만들기
  - API Gateway를 사용하여 WebSocket 채팅 애플리케이션 생성
  - 고객 피드백을 분석하고 오디오를 합성하는 애플리케이션 생성

- 브라우저에서 Lambda 함수 호출
- S3 객체 Lambda를 사용하여 애플리케이션의 데이터 변환
- API Gateway를 사용하여 Lambda 함수 호출
- Step Functions를 사용하여 Lambda 함수 호출
- 예약된 이벤트를 사용하여 Lambda 함수 호출

AWS SDK를 사용한 Lambda 작업

다음 코드 예제에서는 AWS SDK를 통해 개별 Lambda 작업을 수행하는 방법을 보여줍니다. 이들 발췌문은 Lambda API를 호출하며, 컨텍스트에서 실행되어야 하는 더 큰 프로그램에서 발췌한 코드입니다. 각 예제에는 GitHub에 대한 링크가 포함되어 있습니다. 여기에서 코드 설정 및 실행에 대한 지침을 찾을 수 있습니다.

다음 예제에는 가장 일반적으로 사용되는 작업만 포함되어 있습니다. 전체 목록은 AWS LambdaAPI 참조를 참조하세요.

예제

- AWS SDK 또는 CLI와 함께 CreateAlias 사용
- AWS SDK 또는 CLI와 함께 CreateFunction 사용
- AWS SDK 또는 CLI와 함께 DeleteAlias 사용
- AWS SDK 또는 CLI와 함께 DeleteFunction 사용
- AWS SDK 또는 CLI와 함께 DeleteFunctionConcurrency 사용
- AWS SDK 또는 CLI와 함께 DeleteProvisionedConcurrencyConfig 사용
- AWS SDK 또는 CLI와 함께 GetAccountSettings 사용
- AWS SDK 또는 CLI와 함께 GetAlias 사용
- AWS SDK 또는 CLI와 함께 GetFunction 사용
- AWS SDK 또는 CLI와 함께 GetFunctionConcurrency 사용
- AWS SDK 또는 CLI와 함께 GetFunctionConfiguration 사용
- AWS SDK 또는 CLI와 함께 GetPolicy 사용
- AWS SDK 또는 CLI와 함께 GetProvisionedConcurrencyConfig 사용
- AWS SDK 또는 CLI와 함께 Invoke 사용
- AWS SDK 또는 CLI와 함께 ListFunctions 사용

- AWS SDK 또는 CLI와 함께 ListProvisionedConcurrencyConfigs 사용
- AWS SDK 또는 CLI와 함께 ListTags 사용
- AWS SDK 또는 CLI와 함께 ListVersionsByFunction 사용
- AWS SDK 또는 CLI와 함께 PublishVersion 사용
- AWS SDK 또는 CLI와 함께 PutFunctionConcurrency 사용
- AWS SDK 또는 CLI와 함께 PutProvisionedConcurrencyConfig 사용
- AWS SDK 또는 CLI와 함께 RemovePermission 사용
- AWS SDK 또는 CLI와 함께 TagResource 사용
- AWS SDK 또는 CLI와 함께 UntagResource 사용
- AWS SDK 또는 CLI와 함께 UpdateAlias 사용
- AWS SDK 또는 CLI와 함께 UpdateFunctionCode 사용
- AWS SDK 또는 CLI와 함께 UpdateFunctionConfiguration 사용

AWS SDK 또는 CLI와 함께 **CreateAlias** 사용

다음 코드 예시는 `CreateAlias`의 사용 방법을 보여줍니다.

CLI

AWS CLI

Lambda 함수에 대한 별칭을 생성하려면 다음을 수행합니다.

다음 `create-alias` 예제에서는 `my-function` Lambda 함수의 버전 1을 가리키는 `LIVE`라는 별칭을 생성합니다.

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

출력:

```
{
```

```
    "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"
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 CreateAlias를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 지정된 버전 및 라우팅 구성에 대한 새 Lambda 별칭을 생성하여 수신하는 간접 호출 요청의 비율을 지정합니다.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 CreateAlias를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **CreateFunction** 사용

다음 코드 예시는 CreateFunction의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
/// <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 세부 정보는 AWS SDK for .NET API 참조의 CreateFunction을 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 CreateFunction을 참조하십시오.

CLI

AWS CLI

Lambda 함수를 생성하는 방법

다음 `create-function` 예제에서는 이름이 `my-function`인 Lambda 함수를 생성합니다.

```
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`의 콘텐츠:

```
This file is a deployment package that contains your function code and any
 dependencies.
```

출력:

```
{
    "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": ""
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 CreateFunction을 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 CreateFunction을 참조하십시오.

Java

SDK for Java 2.x

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Java 2.x API 참조의 CreateFunction을 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 CreateFunction을 참조하십시오.

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Kotlin API 참조의 CreateFunction를 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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 세부 정보는 AWS SDK for PHP API 참조의 CreateFunction을 참조하십시오.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 AWS Lambda에서 MyFunction이라는 이름의 새 C#(dotnetcore1.0 런타임) 함수를 생성하여 로컬 파일 시스템의 zip 파일(상대 경로 또는 절대 경로 사용 가능)에서 함수에 대한 컴파일된 바이너리를 제공합니다. C# Lambda 함수는 AssemblyName::Namespace.ClassName::MethodName 지정을 사용하여 함수에 대한 핸들러를 지정합니다. 핸들러 사양의 어셈블리 이름 (.dll 접미사 제외), 네임스페이스, 클래스 이름, 메

서드 이름 부분을 적절하게 바꿔야 합니다. 새 함수에는 제공된 값으로부터 환경 변수 'envvar1' 및 'envvar2'가 설정됩니다.

```
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" }
```

출력:

```
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        :
```

예제 2: 이 예제는 함수 바이너리가 먼저 Amazon S3 버킷(의도한 Lambda 함수와 동일한 리전에 있어야 함)에 업로드되고 함수 생성 시 결과 S3 객체가 참조된다는 점을 제외하면 이전 예제와 유사합니다.

```
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 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 CreateFunction을 참조하세요.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 CreateFunction를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 CreateFunction을 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /**
     * 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 세부 정보는 AWS SDK for Rust API 참조의 CreateFunction을 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
   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 세부 정보는 SAP ABAP용 AWS SDK API 참조의 CreateFunction을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **DeleteAlias** 사용

다음 코드 예시는 DeleteAlias의 사용 방법을 보여줍니다.

CLI

AWS CLI

Lambda 함수의 별칭을 삭제하려면 다음을 수행합니다.

다음 delete-alias 예제는 my-function Lambda 함수에서 LIVE라는 별칭을 삭제합니다.

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

이 명령은 출력을 생성하지 않습니다.

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 DeleteAlias를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 명령에 언급된 Lambda 함수 별칭을 삭제합니다.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 DeleteAlias를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **DeleteFunction** 사용

다음 코드 예시는 DeleteFunction의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /// <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 세부 정보는 AWS SDK for .NET API 참조의 DeleteFunction을 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 DeleteFunction을 참조하십시오.

CLI

AWS CLI

예제 1: 함수 이름을 기준으로 Lambda 함수를 삭제하는 방법

다음 `delete-function` 예제에서는 함수 이름을 지정하여 이름이 `my-function`인 Lambda 함수를 삭제합니다.

```
aws lambda delete-function \
    --function-name my-function
```

이 명령은 출력을 생성하지 않습니다.

예제 2: 함수 ARN을 기준으로 Lambda 함수를 삭제하는 방법

다음 `delete-function` 예제에서는 함수 ARN을 지정하여 이름이 `my-function`인 Lambda 함수를 삭제합니다.

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

이 명령은 출력을 생성하지 않습니다.

예제 3: 부분 함수 ARN을 기준으로 Lambda 함수를 삭제하는 방법

다음 `delete-function` 예제에서는 함수의 부분 ARN을 지정하여 이름이 `my-function`인 Lambda 함수를 삭제합니다.

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

이 명령은 출력을 생성하지 않습니다.

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 DeleteFunction을 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 DeleteFunction을 참조하십시오.

Java

SDK for Java 2.x

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Java 2.x API 참조의 DeleteFunction을 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
```

```
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- API 세부 정보는 AWS SDK for JavaScript API 참조의 DeleteFunction을 참조하십시오.

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Kotlin API 참조의 DeleteFunction를 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- API 세부 정보는 AWS SDK for PHP API 참조의 DeleteFunction을 참조하십시오.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 특정 버전의 Lambda 함수를 삭제합니다.

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 DeleteFunction을 참조하세요.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 DeleteFunction를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 DeleteFunction을 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
   /** 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 세부 정보는 AWS SDK for Rust API 참조의 DeleteFunction을 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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 세부 정보는 SAP ABAP용 AWS SDK API 참조의 DeleteFunction을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **DeleteFunctionConcurrency** 사용

다음 코드 예시는 DeleteFunctionConcurrency의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수에서 예약된 동시 실행 제한을 제거하려면 다음을 수행합니다.

다음 delete-function-concurrency 예제에서는 my-function 함수에서 예약된 동시 실행 제한을 삭제합니다.

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

이 명령은 출력을 생성하지 않습니다.

자세한 내용은 AWS Lambda 개발자 안내서의 Lambda 함수에 대한 동시성 예약을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 DeleteFunctionConcurrency를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수의 함수 동시성을 제거합니다.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 DeleteFunctionConcurrency를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **DeleteProvisionedConcurrencyConfig** 사용

다음 코드 예시는 DeleteProvisionedConcurrencyConfig의 사용 방법을 보여줍니다.

CLI

AWS CLI

프로비저닝된 동시성 구성을 삭제하려면 다음을 수행합니다.

다음 delete-provisioned-concurrency-config 예제에서는 지정된 함수의 GREEN 별칭에 대해 프로비저닝된 동시성 구성을 삭제합니다.

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- API 세부 정보는 AWS CLI 명령 참조의 DeleteProvisionedConcurrencyConfig를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 특정 별칭에 대한 프로비저닝된 동시성 구성을 제거합니다.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 DeleteProvisionedConcurrencyConfig를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetAccountSettings** 사용

다음 코드 예시는 GetAccountSettings의 사용 방법을 보여줍니다.

CLI

AWS CLI

AWS 리전에서 내 계정에 대한 세부 정보를 검색하려면 다음을 수행합니다.

다음 get-account-settings 예제는 계정에 대한 Lambda 한도 및 사용량 정보를 표시합니다.

```
aws lambda get-account-settings
```

출력:

```
{
    "AccountLimit": {
       "CodeSizeUnzipped": 262144000,
       "UnreservedConcurrentExecutions": 1000,
       "ConcurrentExecutions": 1000,
       "CodeSizeZipped": 52428800,
       "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
```

```
        "FunctionCount": 4,
        "TotalCodeSize": 9426
    }
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 제한을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 GetAccountSettings를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 샘플은 계정 제한과 계정 사용량을 비교하기 위해 표시됩니다.

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

출력:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetAccountSettings를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetAlias** 사용

다음 코드 예시는 GetAlias의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수 별칭에 대한 세부 정보를 검색하려면 다음을 수행합니다.

다음 get-alias 예제에서는 my-function Lambda 함수에서 LIVE라는 별칭에 대한 세부 정보를 표시합니다.

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

출력:

```
{
    "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"
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 GetAlias를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 특정 Lambda 함수 별칭에 대한 라우팅 구성 가중치를 검색합니다.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

출력:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetAlias를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetFunction** 사용

다음 코드 예시는 GetFunction의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /// <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 세부 정보는 AWS SDK for .NET API 참조의 GetFunction을 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 GetFunction을 참조하십시오.

CLI

AWS CLI

함수 정보를 검색하는 방법

다음 get-function 예제에서는 my-function 함수에 대한 정보를 표시합니다.

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

출력:

```
{
    "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": ""
    }
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 GetFunction을 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 GetFunction을 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- API 세부 정보는 AWS SDK for JavaScript API 참조의 GetFunction을 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
```

```
        ]);
    }
```

- API 세부 정보는 AWS SDK for PHP API 참조의 GetFunction을 참조하십시오.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 GetFunction를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 GetFunction을 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /** 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 세부 정보는 AWS SDK for Rust API 참조의 GetFunction을 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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 세부 정보는 SAP ABAP용 AWS SDK API 참조의 GetFunction을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetFunctionConcurrency** 사용

다음 코드 예시는 GetFunctionConcurrency의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수에 대한 예약된 동시성 설정을 보려면 다음을 수행합니다.

다음 get-function-concurrency 예제에서는 지정된 함수에 대한 예약된 동시성 설정을 검색합니다.

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

출력:

```
{
    "ReservedConcurrentExecutions": 250
}
```

- API 세부 정보는 AWS CLI 명령 참조의 GetFunctionConcurrency를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수에 대한 예약된 동시성을 가져옵니다.

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

출력:

```
ReservedConcurrentExecutions
----------------------------
100
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetFunctionConcurrency를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetFunctionConfiguration** 사용

다음 코드 예시는 GetFunctionConfiguration의 사용 방법을 보여줍니다.

CLI

AWS CLI

Lambda 함수의 버전별 설정을 검색하려면 다음을 수행합니다.

다음 get-function-configuration 예제에서는 my-function 함수의 버전 2에 대한 설정을 표시합니다.

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

출력:

```
{
    "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"
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 GetFunctionConfiguration을 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수의 버전별 구성을 반환합니다.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

출력:

```
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 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetFunctionConfiguration을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetPolicy** 사용

다음 코드 예시는 GetPolicy의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수, 버전 또는 별칭에 대한 리소스 기반 IAM 정책을 검색하려면 다음을 수행합니다.

다음 `get-policy` 예제에서는 `my-function` Lambda 함수에 대한 정책 정보를 표시합니다.

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

출력:

```
{
    "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"
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda에 리소스 기반 정책 사용을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 GetPolicy를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 샘플은 Lambda 함수의 함수 정책을 표시합니다.

```
Get-LMPolicy -FunctionName test -Select Policy
```

출력:

```
{"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 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetPolicy를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **GetProvisionedConcurrencyConfig** 사용

다음 코드 예시는 GetProvisionedConcurrencyConfig의 사용 방법을 보여줍니다.

CLI

AWS CLI

프로비저닝된 동시성 구성을 보려면 다음을 수행합니다.

다음 get-provisioned-concurrency-config 예제에서는 지정된 함수의 BLUE 별칭에 대해 프로비저닝된 동시성 구성의 세부 정보를 표시합니다.

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

출력:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
```

```
}
```

- API 세부 정보는 AWS CLI 명령 참조의 GetProvisionedConcurrencyConfig를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수의 지정된 별칭에 대해 프로비저닝된 동시성 구성을 가져옵니다.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

출력:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 GetProvisionedConcurrencyConfig를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **Invoke** 사용

다음 코드 예시는 Invoke의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /// <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 세부 정보는 AWS SDK for .NET API 참조의 호출을 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 호출을 참조하십시오.

CLI

AWS CLI

예제 1: Lambda 함수를 동기적으로 간접 호출하는 방법

다음 `invoke` 예제는 `my-function` 함수를 동기적으로 간접 호출합니다. `cli-binary-format` 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 자자세한 내용은 AWS Command Line Interface 사용 설명서에서 AWS CL에서 지원되는 전역 명령줄 옵션을 참조하세요.

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

출력:

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

자세한 내용은 AWS 개발자 안내서에서 동기식 간접 호출을 참조하세요.

예제 2: Lambda 함수를 비동기적으로 간접 호출하는 방법

다음 `invoke` 예제에서는 `my-function` 함수를 비동기적으로 간접 호출합니다. `cli-binary-format` 옵션은 AWS CLI 버전 2를 사용할 때 필요합니다. 자자세한 내용은 AWS Command Line Interface 사용 설명서에서 AWS CL에서 지원되는 전역 명령줄 옵션을 참조하세요.

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

출력:

```
{
    "StatusCode": 202
}
```

자세한 내용은 AWS 개발자 안내서에서 비동기 간접 호출을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 Invoke를 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 호출을 참조하십시오.

Java

SDK for Java 2.x

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Java 2.x API 참조의 호출을 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 호출을 참조하십시오.

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Kotlin API 참조의 Invoke를 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- API 세부 정보는 AWS SDK for PHP API 참조의 호출을 참조하십시오.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 Invoke를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 호출을 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /** 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 세부 정보는 AWS SDK for Rust API 참조의 Invoke를 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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.
```

- API에 대한 세부 정보는 SAP ABAP용 AWS SDK API 참조의 Invoke를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **ListFunctions** 사용

다음 코드 예시는 ListFunctions의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
/// <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 세부 정보는 AWS SDK for .NET API 참조의 ListFunctions를 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 ListFunctions를 참조하십시오.

CLI

AWS CLI

Lambda 함수 목록을 검색하는 방법

다음 `list-functions` 예제에서는 현재 사용자의 모든 함수 목록을 표시합니다.

```
aws lambda list-functions
```

출력:

```
{
    "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": ""
        }
    ]
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 ListFunctions를 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 ListFunctions를 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- API 세부 정보는 AWS SDK for JavaScript API 참조의 ListFunctions를 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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 세부 정보는 AWS SDK for PHP API 참조의 ListFunctions를 참조하십시오.

PowerShell

PowerShell용 도구

예제 1: 이 샘플은 정렬된 코드 크기로 모든 Lambda 함수를 표시합니다.

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

출력:

```
FunctionName                                                  Runtime   Timeout
 CodeSize
------------                                                  -------   -------
 --------
test                                                          python2.7       3
   243
MylambdaFunction123                                           python3.8     600
   659
myfuncpython1                                                 python3.8     303
   675
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 ListFunctions를 참조하세요.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 ListFunctions를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 ListFunctions를 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /** 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 세부 정보는 AWS SDK for Rust API 참조의 ListFunctions를 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    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 세부 정보는 SAP ABAP용 AWS SDK API 참조의 ListFunctions를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **ListProvisionedConcurrencyConfigs** 사용

다음 코드 예시는 `ListProvisionedConcurrencyConfigs`의 사용 방법을 보여줍니다.

CLI

AWS CLI

프로비저닝된 동시성 구성의 목록을 가져오려면 다음을 수행합니다.

다음 `list-provisioned-concurrency-configs` 예제에는 지정된 함수에 대한 프로비저닝된 동시성 구성이 나열되어 있습니다.

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

출력:

```
{
```

```
    "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 세부 정보는 AWS CLI 명령 참조의 ListProvisionedConcurrencyConfigs를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수에 대해 프로비저닝된 동시성 구성 목록을 검색합니다.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 ListProvisionedConcurrencyConfigs를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **ListTags** 사용

다음 코드 예시는 ListTags의 사용 방법을 보여줍니다.

CLI

AWS CLI

Lambda 함수에 대한 태그 목록을 검색하려면 다음을 수행합니다.

다음 `list-tags` 예제에서는 `my-function` Lambda 함수에 연결된 태그를 표시합니다.

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

출력:

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 Lambda 함수 태그 지정을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 ListTags를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 지정된 함수에 현재 설정된 태그와 해당 값을 검색합니다.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

출력:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 ListTags를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **ListVersionsByFunction** 사용

다음 코드 예시는 ListVersionsByFunction의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수 버전의 목록을 검색하려면 다음을 수행합니다.

다음 list-versions-by-function 예제에서는 my-function Lambda 함수의 버전 목록을 표시합니다.

```
aws lambda list-versions-by-function \
    --function-name my-function
```

출력:

```
{
    "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"
        }
    ]
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 ListVersionsByFunction을 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수의 각 버전에 대한 버전별 구성 목록을 반환합니다.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

출력:

```
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 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 ListVersionsByFunction을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **PublishVersion** 사용

다음 코드 예시는 PublishVersion의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수의 새 버전을 게시하려면 다음을 수행합니다.

다음 publish-version 예제에서는 my-function Lambda 함수의 새 버전을 게시합니다.

```
aws lambda publish-version \
    --function-name my-function
```

출력:

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

```
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 PublishVersion을 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수 코드의 기존 스냅샷에 대한 버전을 생성합니다.

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 PublishVersion을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **PutFunctionConcurrency** 사용

다음 코드 예시는 PutFunctionConcurrency의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수에 대해 예약된 동시성 한도를 구성하려면 다음을 수행합니다.

다음 put-function-concurrency 예제에서는 my-function 함수에 대해 100개의 예약된 동시 실행을 구성합니다.

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

출력:

```
{
    "ReservedConcurrentExecutions": 100
```

```
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 Lambda 함수에 대한 동시성 예약을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 PutFunctionConcurrency를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 함수 전체에 대해 동시성 설정을 적용합니다.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 PutFunctionConcurrency를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **PutProvisionedConcurrencyConfig** 사용

다음 코드 예시는 PutProvisionedConcurrencyConfig의 사용 방법을 보여줍니다.

CLI

AWS CLI

프로비저닝된 동시성을 할당하려면 다음을 수행합니다.

다음 put-provisioned-concurrency-config 예제에서는 지정된 함수의 BLUE 별칭에 대해 프로비저닝된 동시성 100개를 할당합니다.

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

출력:

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- API 세부 정보는 AWS CLI 명령 참조의 PutProvisionedConcurrencyConfig를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 함수의 별칭에 프로비저닝된 동시성 구성을 추가합니다.

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 PutProvisionedConcurrencyConfig를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **RemovePermission** 사용

다음 코드 예시는 RemovePermission의 사용 방법을 보여줍니다.

CLI

AWS CLI

기존 Lambda 함수에서 권한을 제거하려면 다음을 수행합니다.

다음 remove-permission 예제에서는 my-function이라는 함수를 간접적으로 호출할 수 있는 권한을 제거합니다.

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

이 명령은 출력을 생성하지 않습니다.

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda에 리소스 기반 정책 사용을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 RemovePermission을 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 Lambda 함수의 지정된 StatementId에 대한 함수 정책을 제거합니다.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 RemovePermission을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 `TagResource` 사용

다음 코드 예시는 `TagResource`의 사용 방법을 보여줍니다.

CLI

AWS CLI

기존 Lambda 함수에 태그를 추가하려면 다음을 수행합니다.

다음 `tag-resource` 예제는 지정된 Lambda 함수에 키 이름 `DEPARTMENT`와 값이 `Department A`인 태그를 추가합니다.

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

이 명령은 출력을 생성하지 않습니다.

자세한 내용은 AWS Lambda 개발자 안내서의 Lambda 함수 태그 지정을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 TagResource를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 3개의 태그(워싱턴, 오리건, 캘리포니아)와 관련 값을 ARN으로 식별되는 지정된 함수에 추가합니다.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 TagResource를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **UntagResource** 사용

다음 코드 예시는 UntagResource의 사용 방법을 보여줍니다.

CLI

AWS CLI

기존 Lambda 함수에서 태그를 제거하려면 다음을 수행합니다.

다음 untag-resource 예제에서는 my-function Lambda 함수에서 키 이름 DEPARTMENT 태그가 있는 태그를 제거합니다.

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

이 명령은 출력을 생성하지 않습니다.

자세한 내용은 AWS Lambda 개발자 안내서의 Lambda 함수 태그 지정을 참조하세요.

- API 세부 정보는 AWS CLI Command Reference의 UntagResource를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 함수에서 제공된 태그를 제거합니다. -Force 스위치가 지정되지 않은 경우 cmdlet은 진행하기 전에 확인 메시지를 표시합니다. 태그를 제거하려면 서비스를 한 번만 직접적으로 호출하면 됩니다.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

예제 2: 함수에서 제공된 태그를 제거합니다. -Force 스위치가 지정되지 않은 경우 cmdlet은 진행하기 전에 확인 메시지를 표시합니다. 제공된 태그별로 서비스에 대한 호출은 한 번만 이루어집니다.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 UntagResource를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **UpdateAlias** 사용

다음 코드 예시는 UpdateAlias의 사용 방법을 보여줍니다.

CLI

AWS CLI

함수 별칭을 업데이트하려면 다음을 수행합니다.

다음 update-alias 예제에서는 my-function Lambda 함수의 버전 3을 가리키도록 LIVE라는 별칭을 업데이트합니다.

```
aws lambda update-alias \
```

```
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

출력:

```
{
    "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"
}
```

자세한 내용은 AWS Lambda 개발자 안내서의 AWS Lambda 함수 별칭 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조의 UpdateAlias를 참조하세요.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 기존 Lambda 함수 별칭의 구성을 업데이트합니다. 트래픽의 60%(0.6)를 버전 1로 이동하도록 RoutingConfiguration 값을 업데이트합니다.

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 UpdateAlias를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **UpdateFunctionCode** 사용

다음 코드 예시는 UpdateFunctionCode의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /// <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 세부 정보는 AWS SDK for .NET API 참조의 UpdateFunctionCode를 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
        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 세부 정보는 AWS SDK for C++ API 참조의 UpdateFunctionCode를 참조하십시오.

CLI

AWS CLI

Lambda 함수 코드를 업데이트하는 방법

다음 `update-function-code` 예제에서는 `my-function` 함수의 게시되지 않은 ($LATEST) 버전 코드를 지정된 zip 파일의 콘텐츠로 바꿉니다.

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

출력:

```
{
    "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"
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 UpdateFunctionCode를 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 UpdateFunctionCode를 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 UpdateFunctionCode를 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
    {
```

```
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- API 세부 정보는 AWS SDK for PHP API 참조의 UpdateFunctionCode를 참조하십시오.

PowerShell

PowerShell용 도구

예제 1: 지정된 zip 파일에 포함된 새 콘텐츠로 'MyFunction'이라는 함수를 업데이트합니다. C# .NET Core Lambda 함수의 경우 zip 파일에는 컴파일된 어셈블리가 포함되어야 합니다.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

예제 2: 이 예제는 이전 예제와 유사하지만 업데이트된 코드가 포함된 Amazon S3 객체를 사용하여 함수를 업데이트합니다.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName mybucket -Key
 UpdatedCode.zip
```

- API 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 UpdateFunctionCode를 참조하세요.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 UpdateFunctionCode를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 UpdateFunctionCode를 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /** 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 세부 정보는 AWS SDK for Rust API 참조의 UpdateFunctionCode를 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
   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.
```

- API에 대한 세부 정보는 SAP ABAP용 AWS SDK API 참조의 UpdateFunctionCode를 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK 또는 CLI와 함께 **UpdateFunctionConfiguration** 사용

다음 코드 예시는 UpdateFunctionConfiguration의 사용 방법을 보여줍니다.

작업 예제는 대규모 프로그램에서 발췌한 코드이며 컨텍스트에 맞춰 실행해야 합니다. 다음 코드 예제에서는 컨텍스트 내에서 이 작업을 확인할 수 있습니다.

- 함수 시작하기

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /// <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 세부 정보는 AWS SDK for .NET API 참조의 UpdateFunctionConfiguration을 참조하십시오.

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
       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 세부 정보는 AWS SDK for C++ API 참조의 UpdateFunctionConfiguration을 참조하십시오.

CLI

AWS CLI

함수 구성을 수정하는 방법

다음 `update-function-configuration` 예제에서는 `my-function` 함수의 게시되지 않은 ($LATEST) 버전에서 메모리 크기를 256MB로 수정합니다.

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

출력:

```
{
    "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"
}
```

자세한 설명은 AWS 개발자 안내서에서 AWS Lambda 함수 구성을 참조하세요.

- API 세부 정보는 AWS CLI 명령 참조에서 UpdateFunctionConfiguration을 참조하세요.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 UpdateFunctionConfiguration을 참조하십시오.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 UpdateFunctionConfiguration을 참조하십시오.

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    public function updateFunctionConfiguration($functionName, $handler,
$environment = '')
    {
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- API 세부 정보는 AWS SDK for PHP API 참조의 UpdateFunctionConfiguration을 참조하십시오.

PowerShell

PowerShell용 도구

예제 1: 이 예제에서는 기존 Lambda 함수의 구성을 업데이트합니다.

```
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 세부 정보는 AWS Tools for PowerShell Cmdlet 참조의 UpdateFunctionConfiguration을 참조하세요.

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWSSDK for Python (Boto3) API 참조의 UpdateFunctionConfiguration를 참조하십시오.

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Ruby API 참조의 UpdateFunctionConfiguration을 참조하십시오.

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

```
    /** 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 세부 정보는 AWS SDK for Rust API 참조의 UpdateFunctionConfiguration을 참조하십시오.

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
   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.
```

- API에 대한 세부 정보는 SAP ABAP용 AWS SDK API 참조의 UpdateFunctionConfiguration을 참조하세요.

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용한 Lamda에 대한 시나리오

다음 코드 예제는 AWS SDK를 사용하여 Lambda에서 일반적인 시나리오를 구현하는 방법을 보여줍니다. 이러한 시나리오에서는 Lambda 내에서 여러 함수를 호출하여 특정 태스크를 수행하는 방법을 보여줍니다. 각 시나리오에는 GitHub에 대한 링크가 포함되어 있습니다. 여기에서 코드를 설정하고 실행하는 방법에 대한 지침을 찾을 수 있습니다.

예제

- AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 확인
- AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 마이그레이션
- AWS SDK를 사용하여 Lamda 함수 생성 및 호출 시작하기
- AWS SDK를 사용한 Amazon Cognito 사용자 인증 후 Lambda 함수를 사용하여 사용자 지정 활동 데이터 작성

AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 확인

다음 코드 예제는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 확인하는 방법을 보여줍니다.

- `PreSignUp` 트리거에 대해 Lambda 함수를 호출하도록 사용자 풀을 구성합니다.

- Amazon Cognito를 사용하여 사용자 가입시키기
- Lambda 함수는 DynamoDB 테이블을 스캔하고 알려진 사용자를 자동으로 확인합니다.
- 새 사용자로 로그인한 다음 리소스를 정리합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

명령 프롬프트에서 대화형 시나리오를 실행합니다.

```
// 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 함수를 사용하여 `PreSignUp` 트리거를 처리합니다.

```
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)
}
```

일반적인 작업을 수행하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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
}
```

리소스를 정리합니다.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 다음 주제를 참조하십시오.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용하는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 마이그레이션

다음 코드 예제는 Lambda 함수를 사용하여 알려진 Amazon Cognito 사용자를 자동으로 마이그레이션하는 방법을 보여줍니다.

- `MigrateUser` 트리거에 대해 Lambda 함수를 호출하도록 사용자 풀을 구성합니다.
- 사용자 풀에 없는 사용자 이름과 이메일을 사용하여 Amazon Cognito에 로그인합니다.
- Lambda 함수는 DynamoDB 테이블을 스캔하고 알려진 사용자를 사용자 풀로 자동으로 마이그레이션합니다.
- 암호 찾기 흐름을 수행하여 마이그레이션된 사용자의 암호를 재설정합니다.
- 새 사용자로 로그인한 다음 리소스를 정리합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

명령 프롬프트에서 대화형 시나리오를 실행합니다.

```
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 함수를 사용하여 `MigrateUser` 트리거를 처리합니다.

```
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)
}
```

일반적인 작업을 수행하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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
}
```

리소스를 정리합니다.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 다음 주제를 참조하십시오.
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용하여 Lamda 함수 생성 및 호출 시작하기

다음 코드 예제는 다음과 같은 작업을 수행하는 방법을 보여줍니다.

- IAM 역할과 Lambda 함수를 생성하고 핸들러 코드를 업로드합니다.

- 단일 파라미터로 함수를 간접적으로 호출하고 결과를 가져옵니다.
- 함수 코드를 업데이트하고 환경 변수로 구성합니다.
- 새 파라미터로 함수를 간접적으로 호출하고 결과를 가져옵니다. 반환된 실행 로그를 표시합니다.
- 계정의 함수를 나열합니다.

자세한 내용은 콘솔로 Lambda 함수 생성을 참조하십시오.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

Lambda 작업을 수행하는 메서드를 생성합니다.

```
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;
    }


}
```

시나리오를 실행하는 함수를 생성합니다.

```
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();
    }
```

```
}
```

숫자를 증가시키는 Lambda 핸들러를 정의합니다.

```
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;
        }
    }
}
```

산술 연산을 수행하는 두 번째 Lambda 핸들러를 정의합니다.

```
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 세부 정보는 AWS SDK for .NET API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK for C++

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
//! 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 세부 정보는 AWS SDK for C++ API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

Lambda 함수를 시작하는 방법을 보여주는 대화형 시나리오를 생성합니다.

```
// 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.")
 }
}
```

개별 Lambda 작업을 래핑하는 구조체를 생성합니다.

```
// 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"`
}
```

시나리오를 실행하는 데 도움이 되는 함수를 구현하는 구조체를 생성합니다.

```
// 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
```

```
}
```

숫자를 증가시키는 Lambda 핸들러를 정의합니다.

```
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
```

산술 연산을 수행하는 두 번째 Lambda 핸들러를 정의합니다.

```
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 세부 정보는 AWS SDK for Go API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK for Java 2.x

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
/*
 *  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 세부 정보는 AWS SDK for Java 2.x API 참조의 다음 항목을 참조하세요.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

로그에 쓸 수 있는 권한을 Lambda에 부여하는 AWS Identity and Access Management (IAM) 역할을 생성합니다.

```
    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 함수를 생성하고 핸들러 코드를 업로드합니다.

```
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);
};
```

단일 파라미터로 함수를 간접적으로 호출하고 결과를 가져옵니다.

```
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 };
};
```

함수 코드를 업데이트하고 환경 변수를 사용하여 Lambda 환경을 구성합니다.

```
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);
};
```

계정의 함수를 나열합니다.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

IAM 역할과 Lambda 함수를 삭제합니다.

```
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 세부 정보는 AWS SDK for JavaScript API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

Kotlin

SDK for Kotlin

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for Kotlin API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

SDK for PHP

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
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 세부 정보는 AWS SDK for PHP API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK for Python (Boto3)

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

숫자를 증가시키는 Lambda 핸들러를 정의합니다.

```
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
```

산술 연산을 수행하는 두 번째 Lambda 핸들러를 정의합니다.

```
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 작업을 래핑하는 함수를 생성합니다.

```
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
```

시나리오를 실행하는 함수를 생성합니다.

```
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 세부 정보는 AWS SDK for Python (Boto3) API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Ruby

SDK for Ruby

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

로그를 작성할 수 있는 Lambda 함수에 대한 사전 요구 IAM 권한을 설정합니다.

```
  # 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
```

호출 파라미터로 제공된 숫자를 증가 시키는 Lambda 핸들러를 정의합니다.

```
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 함수를 배포 패키지로 압축합니다.

```
 # 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
```

새 Lambda 함수를 생성합니다.

```
 # 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
```

선택적 런타임 파라미터를 사용하여 Lambda 함수를 호출합니다.

```
 # 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 함수의 구성을 업데이트하여 새 환경 변수를 삽입합니다.

```
 # 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 함수의 코드를 다른 코드가 포함된 다른 배포 패키지로 업데이트하십시오.

```
 # 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
```

내장 페이지네이터를 사용하여 기존의 모든 Lambda 함수를 나열합니다.

```
  # 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
```

특정 Lambda 함수를 삭제합니다.

```
  # 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 세부 정보는 AWS SDK for Ruby API 참조의 다음 주제를 참조하십시오.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Rust

SDK for Rust

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

이 시나리오에 사용된 종속 항목이 있는 Cargo.toml입니다.

```
[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"
```

이 시나리오에서 Lambda 직접 호출을 간소화하는 유틸리티 모음입니다. 이 파일은 크레이트에 있는 src/ations.rs입니다.

```
// 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(),
        );
    }
}
```

일부 동작을 제어하기 위해 명령줄 플래그를 사용하여 시나리오를 처음부터 끝까지 실행하는 바이너리입니다. 이 파일은 크레이트에 있는 src/bin/scenario.rs입니다.

```
// 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 세부 정보는 AWS SDK for Rust API 참조의 다음 주제를 참조하십시오.

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

SAP ABAP

SDK for SAP ABAP API

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS 코드 예제 리포지토리에서 전체 예제를 찾고 설정 및 실행하는 방법을 배워보세요.

```
   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 세부 정보는 AWS SDK for SAP ABAP API 참조의 다음 주제를 참조하세요.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용한 Amazon Cognito 사용자 인증 후 Lambda 함수를 사용하여 사용자 지정 활동 데이터 작성

다음 코드 예제는 Amazon Cognito 사용자 인증 후 Lambda 함수를 사용하여 사용자 지정 작업 데이터를 쓰는 방법을 보여줍니다.

- 관리자 함수를 사용하여 사용자 풀에 사용자를 추가합니다.
- `PostAuthentication` 트리거에 대해 Lambda 함수를 호출하도록 사용자 풀을 구성합니다.
- 새로운 사용자를 Amazon Cognito에 로그인시킵니다.
- Lambda 함수는 CloudWatch Logs와 DynamoDB 테이블에 사용자 지정 정보를 작성합니다.
- DynamoDB 테이블에서 사용자 지정 데이터를 가져오기 및 표시한 다음 리소스를 정리합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. AWS코드 예시 리포지토리에서 전체 예시를 찾고 설정 및 실행하는 방법을 배워보세요.

명령 프롬프트에서 대화형 시나리오를 실행합니다.

```
// 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 함수를 사용하여 PostAuthentication 트리거를 처리합니다.

```
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)
}
```

일반적인 작업을 수행하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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 작업을 래핑하는 구조체를 생성합니다.

```
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 작업을 래핑하는 구조체를 생성합니다.

```
// 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
}
```

리소스를 정리합니다.

```
// 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 세부 정보는 AWS SDK for Go API 참조의 다음 주제를 참조하십시오.
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용하는 Lambda의 서버리스 예제

다음 코드 예제에서는 Lambda를 AWS SDK와 함께 사용하는 방법을 보여줍니다.

예제

- Lambda 함수를 사용하여 Amazon RDS 데이터베이스에 연결
- Kinesis 트리거에서 간접적으로 Lambda 함수 호출
- DynamoDB 트리거에서 간접적으로 Lambda 함수 간접 호출
- Amazon DocumentDB 트리거에서 간접적으로 Lambda 함수 호출
- Amazon S3 트리거를 사용하여 Lambda 함수 호출
- Amazon SNS 트리거를 사용하여 Lambda 함수 호출
- Amazon SQS 트리거에서 간접적으로 Lambda 함수 호출
- Kinesis 트리거로 Lambda 함수에 대한 배치 항목 실패 보고
- DynamoDB 트리거로 Lambda 함수에 대한 배치 항목 실패 보고
- Amazon SQS 트리거로 Lambda 함수에 대한 배치 항목 실패 보고

Lambda 함수를 사용하여 Amazon RDS 데이터베이스에 연결

다음 코드 예시는 RDS 데이터베이스에 연결하는 Lambda 함수를 구현하는 방법을 보여줍니다. 이 함수는 간단한 데이터베이스 요청을 하고 결과를 반환합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda 함수에서 Amazon RDS 데이터베이스에 연결

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Javascript를 사용하여 Lambda 함수에서 Amazon RDS 데이터베이스에 연결

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Kinesis 트리거에서 간접적으로 Lambda 함수 호출

다음 코드 예제에서는 Kinesis 스트림에서 레코드를 받아 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 이 함수는 Kinesis 페이로드를 검색하고, Base64에서 디코딩하고, 레코드 콘텐츠를 로깅합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda에서 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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;
}
```

TypeScript를 사용하여 Lambda로 Kinesis 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 Kinesis 이벤트를 사용합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

DynamoDB 트리거에서 간접적으로 Lambda 함수 간접 호출

다음 코드 예시에서는 DynamoDB 스트림에서 레코드를 받아 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 이 함수는 DynamoDB 페이로드를 검색하고 레코드 콘텐츠를 로깅합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 DynamoDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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)}`);
};
```

TypeScript를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 DynamoDB 이벤트 사용.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Amazon DocumentDB 트리거에서 간접적으로 Lambda 함수 호출

다음 코드 예시에서는 DocumentDB 변경 스트림에서 레코드를 받아 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 이 함수는 DocumentDB 페이로드를 검색하고 레코드 콘텐츠를 로깅합니다.

Go

SDK for Go V2

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Amazon DocumentDB 이벤트 소비

```
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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Amazon S3 트리거를 사용하여 Lambda 함수 호출

다음 코드 예제에서는 S3 버킷에 객체를 업로드하여 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 해당 함수는 이벤트 파라미터에서 S3 버킷 이름과 객체 키를 검색하고 Amazon S3 API를 호출하여 객체의 콘텐츠 유형을 검색하고 로깅합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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);
    }
};
```

TypeScript를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 S3 이벤트 사용.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 S3 이벤트 사용.

```
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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 S3 이벤트를 사용합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Amazon SNS 트리거를 사용하여 Lambda 함수 호출

다음 코드 예제에서는 SNS 주제의 메시지를 받아 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 함수는 이벤트 파라미터에서 메시지를 검색하고 각 메시지의 내용을 로깅합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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;
  }
}
```

TypeScript를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SNS 이벤트를 사용합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Amazon SQS 트리거에서 간접적으로 Lambda 함수 호출

다음 코드 예제에서는 SQS 대기열에서 메시지를 받아 트리거된 이벤트를 수신하는 Lambda 함수를 구현하는 방법을 보여줍니다. 함수는 이벤트 파라미터에서 메시지를 검색하고 각 메시지의 내용을 로깅합니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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;
  }
}
```

TypeScript를 사용하여 Lambda로 SQS 이벤트 사용

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SQS 이벤트를 사용합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Kinesis 트리거로 Lambda 함수에 대한 배치 항목 실패 보고

다음 코드 예제에서는 Kinesis 스트림에서 이벤트를 수신하는 Lambda 함수에 대한 부분 배치 응답을 구현하는 방법을 보여줍니다. 이 함수는 응답으로 배치 항목 실패를 보고하고 나중에 해당 메시지를 다시 시도하도록 Lambda에 신호를 보냅니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Javascript를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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;
}
```

TypeScript를 사용하여 Lambda로 Kinesis 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 Kinesis 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 Kinesis 배치 항목 실패를 보고합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

DynamoDB 트리거로 Lambda 함수에 대한 배치 항목 실패 보고

다음 코드 예시에서는 DynamoDB 스트림에서 이벤트를 수신하는 Lambda 함수에 대한 부분 배치 응답을 구현하는 방법을 보여줍니다. 이 함수는 응답으로 배치 항목 실패를 보고하고 나중에 해당 메시지를 다시 시도하도록 Lambda에 신호를 보냅니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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: [] };
};
```

TypeScript를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 DynamoDB 배치 항목 실패 보고.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Amazon SQS 트리거로 Lambda 함수에 대한 배치 항목 실패 보고

다음 코드 예제에서는 SQS 대기열에서 이벤트를 수신하는 Lambda 함수에 대한 부분 배치 응답을 구현하는 방법을 보여줍니다. 이 함수는 응답으로 배치 항목 실패를 보고하고 나중에 해당 메시지를 다시 시도하도록 Lambda에 신호를 보냅니다.

.NET

AWS SDK for .NET

> **Note**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

.NET을 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Go를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Java를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

JavaScript를 사용하여 Lambda에서 SQS 배치 항목 실패를 보고합니다.

```
// 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}`);
}
```

TypeScript를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

PHP를 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
// 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Python을 사용하여 Lambda로 SQS 배치 항목 실패 보고

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Ruby를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
# 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**
>
> GitHub에 더 많은 내용이 있습니다. 서버리스 예제 리포지토리에서 전체 예제를 찾아보고 설정 및 실행 방법을 알아봅니다.

Rust를 사용하여 Lambda로 SQS 배치 항목 실패를 보고합니다.

```
// 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 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

AWS SDK를 사용한 Lambda용 교차 서비스 예제

다음 샘플 애플리케이션에서는 AWS SDK를 사용하여 Lambda를 다른 AWS 서비스와 결합합니다. 각 예시에는 애플리케이션을 설정하고 실행하는 방법에 대한 지침을 찾을 수 있는 GitHub 링크가 포함되어 있습니다.

예제

- COVID-19 데이터를 추적하는 API Gateway REST API 생성
- 대출 라이브러리 REST API 생성
- Step Functions를 사용하여 메신저 애플리케이션 생성
- 사용자가 레이블을 사용하여 사진을 관리할 수 있는 사진 자산 관리 애플리케이션 만들기
- API Gateway를 사용하여 WebSocket 채팅 애플리케이션 생성
- 고객 피드백을 분석하고 오디오를 합성하는 애플리케이션 생성
- 브라우저에서 Lambda 함수 호출
- S3 객체 Lambda를 사용하여 애플리케이션의 데이터 변환
- API Gateway를 사용하여 Lambda 함수 호출
- Step Functions를 사용하여 Lambda 함수 호출
- 예약된 이벤트를 사용하여 Lambda 함수 호출

COVID-19 데이터를 추적하는 API Gateway REST API 생성

다음 코드 예제에서는 가상 데이터를 사용하여 미국의 일별 COVID-19 발생 현황을 추적하는 시스템을 시뮬레이션하는 REST API를 생성하는 방법을 보여줍니다.

Python

SDK for Python(Boto3)

AWS SDK for Python (Boto3)와 함께 AWS Chalice를 사용하여 Amazon API Gateway, AWS Lambda 및 Amazon DynamoDB를 사용한 서버리스 REST API를 생성하는 방법을 보여줍니다. REST API로 가상 데이터를 사용하여 미국의 일별 COVID-19 발생 현황을 추적하는 시스템을 시뮬레이션합니다. 다음 작업을 수행하는 방법에 대해 알아보십시오.

- AWS Chalice를 사용하여 API Gateway를 통해 들어오는 REST 요청을 처리하도록 호출되는 Lambda 함수의 경로를 정의합니다.
- Lambda 함수로 데이터를 검색하고 DynamoDB 테이블에 저장하여 REST 요청을 처리합니다.
- AWS CloudFormation 템플릿에 테이블 구조 및 보안 역할 리소스를 정의합니다.
- AWS Chalice 및 CloudFormation을 사용하여 필요한 모든 리소스를 패키징하고 배포합니다.
- CloudFormation을 사용하여 생성된 모든 리소스를 정리합니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- API Gateway
- AWS CloudFormation
- DynamoDB
- Lambda

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

대출 라이브러리 REST API 생성

다음 코드 예제에서는 Amazon Aurora 데이터베이스가 지원하는 REST API를 사용하여 고객이 도서를 빌리고 반납할 수 있는 대출 라이브러리를 생성하는 방법을 보여줍니다.

Python

SDK for Python(Boto3)

Amazon Relational Database Service(Amazon RDS) API 및 AWS Chalice와 함께 AWS SDK for Python (Boto3)를 사용하여 Amazon Aurora 데이터베이스에서 지원하는 REST API를 생성하는 방법을 보여줍니다. 웹 서비스는 완전히 서버리스이며 고객이 책을 빌리고 반납할 수 있는 간단한 대출 라이브러리를 나타냅니다. 다음 작업을 수행하는 방법에 대해 알아보십시오.

- 서버리스 Aurora 데이터베이스 클러스터를 생성하고 관리합니다.
- AWS Secrets Manager를 사용하여 데이터베이스 자격 증명을 관리합니다.
- Amazon RDS를 사용하여 데이터를 데이터베이스 내부 및 외부로 이동하는 데이터 스토리지 계층을 구현합니다.
- AWS Chalice를 사용하여 서버리스 REST API를 Amazon API Gateway 및 AWS Lambda에 배포합니다.
- 요청 패키지를 사용하여 웹 서비스에 요청을 보냅니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- API Gateway
- Aurora
- Lambda
- Secrets Manager

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Step Functions를 사용하여 메신저 애플리케이션 생성

다음 코드 예제에서는 데이터베이스 테이블에서 메시지 레코드를 검색하는 AWS Step Functions 메신저 애플리케이션을 생성하는 방법을 보여줍니다.

Python

SDK for Python(Boto3)

AWS Step Functions를 AWS SDK for Python (Boto3)와 함께 사용하여 Amazon DynamoDB 테이블에서 메시지 레코드를 검색하고 Amazon Simple Queue Service(Amazon SQS)를 통해 전

송하는 메신저 애플리케이션을 생성하는 방법을 보여줍니다. 상태 머신은 AWS Lambda 함수와 통합되어 데이터베이스에서 전송되지 않은 메시지를 스캔합니다.

- Amazon DynamoDB 테이블에서 메시지 레코드를 검색하고 업데이트하는 상태 머신을 생성합니다.
- 상태 머신 정의를 업데이트하여 메시지를 Amazon Simple Queue Service(Amazon SQS)에도 전송합니다.
- 상태 머신의 실행을 시작하고 중지합니다.
- 서비스 통합을 사용하여 상태 머신에서 Lambda, DynamoDB 및 Amazon SQS에 연결합니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

사용자가 레이블을 사용하여 사진을 관리할 수 있는 사진 자산 관리 애플리케이션 만들기

다음 코드 예제는 사용자가 레이블을 사용하여 사진을 관리할 수 있는 서버리스 애플리케이션을 생성하는 방법을 보여 줍니다.

.NET

AWS SDK for .NET

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Amazon Rekognition을 사용하여 이미지에서 레이블을 감지하고 나중에 검색할 수 있도록 저장하는 사진 자산 관리 애플리케이션을 개발하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제의 출처에 대한 자세한 내용은 AWS 커뮤니티의 게시물을 참조하십시오.

이 예시에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

AWS SDK 개발자 가이드 및 코드 예제의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

API Gateway를 사용하여 WebSocket 채팅 애플리케이션 생성

다음 코드 예제에서는 Amazon API Gateway 기반의 WebSocket API에서 제공되는 채팅 애플리케이션을 생성하는 방법을 보여줍니다.

Python

SDK for Python(Boto3)

Amazon API Gateway V2와 함께 AWS SDK for Python (Boto3)를 사용하여 AWS Lambda 및 Amazon DynamoDB와 통합되는 WebSocket API를 생성하는 방법을 보여줍니다.

- API Gateway에서 제공되는 WebSocket API를 생성합니다.
- DynamoDB에 연결을 저장하고 다른 채팅 참가자에게 메시지를 게시하는 Lambda 핸들러를 정의합니다.
- WebSocket 채팅 애플리케이션에 연결하고 WebSocket 패키지를 사용하여 메시지를 전송합니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

고객 피드백을 분석하고 오디오를 합성하는 애플리케이션 생성

다음 코드 예제에서는 고객 의견 카드를 분석하고, 원어에서 번역하고, 감정을 파악하고, 번역된 텍스트에서 오디오 파일을 생성하는 애플리케이션을 생성하는 방법을 보여줍니다.

.NET

AWS SDK for .NET

이 예제 애플리케이션은 고객 피드백 카드를 분석하고 저장합니다. 특히 뉴욕시에 있는 가상 호텔의 필요를 충족합니다. 호텔은 다양한 언어의 고객들로부터 물리적인 의견 카드의 형태로 피드백을 받습니다. 피드백은 웹 클라이언트를 통해 앱에 업로드됩니다. 의견 카드의 이미지가 업로드된 후 다음 단계가 수행됩니다.

- Amazon Textract를 사용하여 이미지에서 텍스트가 추출됩니다.
- Amazon Comprehend가 추출된 텍스트와 해당 언어의 감정을 파악합니다.
- 추출된 텍스트는 Amazon Translate를 사용하여 영어로 번역됩니다.
- Amazon Polly가 추출된 텍스트에서 오디오 파일을 합성합니다.

전체 앱은 AWS CDK를 사용하여 배포할 수 있습니다. 소스 코드와 배포 지침은 GitHub의 프로젝트를 참조하십시오.

이 예시에서 사용되는 서비스

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK for Java 2.x

이 예제 애플리케이션은 고객 피드백 카드를 분석하고 저장합니다. 특히 뉴욕시에 있는 가상 호텔의 필요를 충족합니다. 호텔은 다양한 언어의 고객들로부터 물리적인 의견 카드의 형태로 피

드백을 받습니다. 피드백은 웹 클라이언트를 통해 앱에 업로드됩니다. 의견 카드의 이미지가 업로드된 후 다음 단계가 수행됩니다.

- Amazon Textract를 사용하여 이미지에서 텍스트가 추출됩니다.
- Amazon Comprehend가 추출된 텍스트와 해당 언어의 감정을 파악합니다.
- 추출된 텍스트는 Amazon Translate를 사용하여 영어로 번역됩니다.
- Amazon Polly가 추출된 텍스트에서 오디오 파일을 합성합니다.

전체 앱은 AWS CDK를 사용하여 배포할 수 있습니다. 소스 코드와 배포 지침은 GitHub의 프로젝트를 참조하십시오.

이 예시에서 사용되는 서비스

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK for JavaScript (v3)

이 예제 애플리케이션은 고객 피드백 카드를 분석하고 저장합니다. 특히 뉴욕시에 있는 가상 호텔의 필요를 충족합니다. 호텔은 다양한 언어의 고객들로부터 물리적인 의견 카드의 형태로 피드백을 받습니다. 피드백은 웹 클라이언트를 통해 앱에 업로드됩니다. 의견 카드의 이미지가 업로드된 후 다음 단계가 수행됩니다.

- Amazon Textract를 사용하여 이미지에서 텍스트가 추출됩니다.
- Amazon Comprehend가 추출된 텍스트와 해당 언어의 감정을 파악합니다.
- 추출된 텍스트는 Amazon Translate를 사용하여 영어로 번역됩니다.
- Amazon Polly가 추출된 텍스트에서 오디오 파일을 합성합니다.

전체 앱은 AWS CDK를 사용하여 배포할 수 있습니다. 소스 코드와 배포 지침은 GitHub의 프로젝트를 참조하십시오. 다음 발췌문은 Lambda 함수 내에서 AWS SDK for JavaScript가 사용되는 방식을 보여줍니다.

```
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 };
};
```

이 예시에서 사용되는 서비스

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK for Ruby

이 예제 애플리케이션은 고객 피드백 카드를 분석하고 저장합니다. 특히 뉴욕시에 있는 가상 호텔의 필요를 충족합니다. 호텔은 다양한 언어의 고객들로부터 물리적인 의견 카드의 형태로 피드백을 받습니다. 피드백은 웹 클라이언트를 통해 앱에 업로드됩니다. 의견 카드의 이미지가 업로드된 후 다음 단계가 수행됩니다.

- Amazon Textract를 사용하여 이미지에서 텍스트가 추출됩니다.
- Amazon Comprehend가 추출된 텍스트와 해당 언어의 감정을 파악합니다.
- 추출된 텍스트는 Amazon Translate를 사용하여 영어로 번역됩니다.
- Amazon Polly가 추출된 텍스트에서 오디오 파일을 합성합니다.

전체 앱은 AWS CDK를 사용하여 배포할 수 있습니다. 소스 코드와 배포 지침은 GitHub의 프로젝트를 참조하십시오.

이 예시에서 사용되는 서비스

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

브라우저에서 Lambda 함수 호출

다음 코드 예제에서는 브라우저에서 AWS Lambda 함수를 호출하는 방법을 보여줍니다.

JavaScript

JavaScript용 SDK(v2)

AWS Lambda 함수를 사용하여 사용자 선택 사항으로 Amazon DynamoDB 테이블을 업데이트하는 브라우저 기반 애플리케이션을 생성할 수 있습니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- DynamoDB
- Lambda

SDK for JavaScript (v3)

AWS Lambda 함수를 사용하여 사용자 선택 사항으로 Amazon DynamoDB 테이블을 업데이트하는 브라우저 기반 애플리케이션을 생성할 수 있습니다. 이 앱은 AWS SDK for JavaScript v3를 사용합니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- DynamoDB
- Lambda

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

S3 객체 Lambda를 사용하여 애플리케이션의 데이터 변환

다음 코드 예시는 S3 객체 Lambda를 사용하여 애플리케이션의 데이터를 변환하는 방법을 보여 줍니다.

.NET

AWS SDK for .NET

표준 S3 GET 요청에 사용자 지정 코드를 추가하여, 요청하는 클라이언트 또는 애플리케이션의 요구 사항에 맞게 S3에서 검색된 요청된 객체를 수정하는 방법을 보여 줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예시를 참조하세요.

이 예시에서 사용되는 서비스

- Lambda
- Amazon S3

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 섹션을 참조하십시오. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

API Gateway를 사용하여 Lambda 함수 호출

다음 코드 예제에서는 Amazon API Gateway에서 호출하여 AWS Lambda 함수를 생성하는 방법을 보여줍니다.

Java

SDK for Java 2.x

Lambda Java 런타임 API를 사용하여 AWS Lambda 함수를 생성하는 방법을 보여줍니다. 이 예제에서는 특정 사용 사례를 수행하는 서로 다른 AWS 서비스를 호출합니다. 이 예제에서는 Amazon API Gateway에서 호출한 Lambda 함수를 생성하여 작업 기념일에 대한 Amazon DynamoDB 테이블을 스캔하고 Amazon Simple Notification Service(Amazon SNS)를 사용하여 직원에게 1주년 기념일을 축하하는 문자 메시지를 전송하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- API Gateway

- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Lambda JavaScript 런타임 API를 사용하여 AWS Lambda 함수를 생성하는 방법을 보여줍니다. 이 예제에서는 특정 사용 사례를 수행하는 서로 다른 AWS 서비스를 호출합니다. 이 예제에서는 Amazon API Gateway에서 호출한 Lambda 함수를 생성하여 작업 기념일에 대한 Amazon DynamoDB 테이블을 스캔하고 Amazon Simple Notification Service(Amazon SNS)를 사용하여 직원에게 1주년 기념일을 축하하는 문자 메시지를 전송하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예시는 AWS SDK for JavaScript v3 개발자 안내서에서도 확인할 수 있습니다.

이 예제에서 사용되는 서비스

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

이 예제에서는 AWS Lambda 함수를 대상으로 하는 Amazon API Gateway REST API를 생성하고 사용하는 방법을 보여줍니다. Lambda 핸들러는 HTTP 메서드를 기반으로 라우팅하는 방법, 쿼리 문자열, 헤더 및 본문에서 데이터를 가져오는 방법, JSON 응답을 반환하는 방법을 보여줍니다.

- Lambda 함수를 배포합니다.
- API Gateway REST API를 생성합니다.
- Lambda 함수를 대상으로 하는 REST 리소스를 생성합니다.
- API Gateway가 Lambda 함수를 호출할 수 있는 권한을 부여합니다.

- 요청 패키지를 사용하여 REST API에 요청을 보냅니다.
- 데모 중에 생성된 모든 리소스를 정리합니다.

이 예제는 GitHub에서 가장 잘 볼 수 있습니다. 전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- API Gateway
- Lambda

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Step Functions를 사용하여 Lambda 함수 호출

다음 코드 예제에서는 AWS Lambda 함수 시퀀스를 호출하는 AWS Step Functions 상태 머신을 생성하는 방법을 보여줍니다.

Java

SDK for Java 2.x

AWS Step Functions 및 AWS SDK for Java 2.x을 사용하여 AWS 서버리스 워크플로를 생성하는 방법을 보여줍니다. 각 워크플로 단계는 AWS Lambda 함수를 사용하여 구현됩니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

JavaScript

SDK for JavaScript (v3)

AWS Step Functions 및 AWS SDK for JavaScript을 사용하여 AWS 서버리스 워크플로를 생성하는 방법을 보여줍니다. 각 워크플로 단계는 AWS Lambda 함수를 사용하여 구현됩니다.

Lambda는 서버를 프로비저닝하거나 관리하지 않고도 코드를 실행할 수 있게 하는 컴퓨팅 서비스입니다. Step Functions는 Lambda 함수와 기타 AWS 서비스를 결합할 수 있는 서버리스 오케스트레이션 서비스로, 비즈니스 크리티컬 애플리케이션을 구축합니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예시는 AWS SDK for JavaScript v3 개발자 안내서에서도 확인할 수 있습니다.

이 예제에서 사용되는 서비스

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

예약된 이벤트를 사용하여 Lambda 함수 호출

다음 코드 예제에서는 Amazon EventBridge의 예약된 이벤트에 의해 호출된 AWS Lambda 함수를 생성하는 방법을 보여줍니다.

Java

SDK for Java 2.x

AWS Lambda 함수를 호출하는 Amazon EventBridge 예약된 이벤트를 생성하는 방법을 보여줍니다. Lambda 함수가 호출될 때 cron 표현식을 사용하여 일정을 예약하도록 EventBridge를 구성합니다. 이 예제에서는 Lambda Java 런타임 API를 사용하여 Lambda 함수를 생성합니다. 이 예제에서는 특정 사용 사례를 수행하는 서로 다른 AWS 서비스를 호출합니다. 이 예제에서는 1주년 기념일에 직원에게 축하하는 모바일 문자 메시지를 전송하는 앱을 생성하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예제에서 사용되는 서비스

- DynamoDB
- EventBridge

- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

AWS Lambda 함수를 호출하는 Amazon EventBridge 예약된 이벤트를 생성하는 방법을 보여줍니다. Lambda 함수가 호출될 때 cron 표현식을 사용하여 일정을 예약하도록 EventBridge를 구성합니다. 이 예제에서는 Lambda JavaScript 런타임 API를 사용하여 Lambda 함수를 생성합니다. 이 예제에서는 특정 사용 사례를 수행하는 서로 다른 AWS 서비스를 호출합니다. 이 예제에서는 1주년 기념일에 직원에게 축하하는 모바일 문자 메시지를 전송하는 앱을 생성하는 방법을 보여줍니다.

전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예시는 AWS SDK for JavaScript v3 개발자 안내서에서도 확인할 수 있습니다.

이 예제에서 사용되는 서비스

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

이 예제에서는 AWS Lambda 함수를 예약된 Amazon EventBridge 이벤트의 대상으로 등록하는 방법을 보여줍니다. Lambda 핸들러는 나중에 검색할 수 있도록 알기 쉬운 메시지와 전체 이벤트 데이터를 Amazon CloudWatch Logs에 기록합니다.

- Lambda 함수를 배포합니다.
- EventBridge 예약된 이벤트를 생성하고 Lambda 함수를 대상으로 만듭니다.
- EventBridge에 Lambda 함수를 호출할 수 있는 권한을 부여합니다.
- CloudWatch Logs에서 최신 데이터를 인쇄하여 예약된 호출의 결과를 표시합니다.
- 데모 중에 생성된 모든 리소스를 정리합니다.

이 예제는 GitHub에서 가장 잘 볼 수 있습니다. 전체 소스 코드와 설정 및 실행 방법에 대한 지침은 GitHub에서 전체 예제를 참조하십시오.

이 예시에서 사용되는 서비스

- CloudWatch Logs
- EventBridge
- Lambda

AWS SDK 개발자 가이드 및 코드 예시의 전체 목록은 AWS SDK에서 Lambda 사용 단원을 참조하세요. 이 주제에는 시작하기에 대한 정보와 이전 SDK 버전에 대한 세부 정보도 포함되어 있습니다.

Lambda 할당량

> **Important**
>
> 새 AWS 계정에서는 동시성 및 메모리 할당량이 감소했습니다. AWS에서는 사용량에 따라 이런 할당량을 자동으로 늘립니다.

컴퓨팅 및 스토리지

Lambda는 함수를 실행하고 저장하는 데 사용할 수 있는 컴퓨팅 및 스토리지 리소스 양의 할당량을 설정합니다. 동시 실행 및 스토리지에 대한 할당량은 AWS 리전에 따라 적용됩니다. 탄력적 네트워크 인터페이스(ENI) 할당량은 지역에 관계없이 Virtual Private Cloud(VPC) 별로 적용됩니다. 다음 할당량은 기본값에서 늘릴 수 있습니다. 자세한 내용은 Service Quotas 사용 설명서의 할당량 증가 요청을 참조하세요.

리소스	기본 할당량	최대 한도 증가
동시 실행	1,000	수십만
업로드된 함수(.zip 파일 아카이브) 및 계층을 위한 스토리지. 각 함수 버전 및 계층 버전은 스토리지를 사용합니다. 코드 스토리지 관리 모범 사례는 Serverless Land의 Monitoring Lambda code storage를 참조하세요.	75GB	TB
컨테이너 이미지로 정의된 함수에 대한 스토리지. 이 이미지는 Amazon ECR에 저장됩니다.	Amazon ECR 서비스 할당량을 참조하세요.	
Virtual Private Cloud(VPC)별 탄력적 네트워크 인터페이스	250	수천

리소스	기본 할당량	최대 한도 증가
Note 이 할당량은 Amazon Elastic File System(Amazon EFS)과 같은 다른 서비스와 공유됩니다. Amazon VPC 할당량을 참조하세요.		

Lambda가 트래픽에 대한 응답으로 함수 동시성을 확장하는 방법과 동시성에 대한 자세한 내용은 Lambda 함수 규모 조정 이행 단원을 참조하세요.

함수 구성, 배포 및 실행

함수 구성, 배포 및 실행에는 다음 할당량이 적용됩니다. 미리 설명된 경우를 제외하고는 변경할 수 없습니다.

Note

Lambda 문서, 로그 메시지 및 콘솔은 약어 MB(MiB 대신)를 사용하여 1,024KB를 나타냅니다.

Resource	Quota
함수 메모리 할당	128MB~10,240MB, 1MB 단위 참고: Lambda는 구성된 메모리 크기에 비례하여 CPU 용량을 할당합니다. 메모리(MB) 설정을 사용하면 함수에 할당된 메모리 및 CPU 처리능력을 늘리거나 줄일 수 있습니다. 1,769MB에서, 함수는 하나의 vCPU와 동등한 값을 가집니다.
함수 제한 시간	900초(15분)

Resource	Quota
함수 환경 변수	4KB, 함수와 관련된 모든 환경 변수에 대한 합계
함수 리소스 기반 정책	20KB
함수 계층	5개 계층
함수 동시성 스케일링 제한	각 함수에 대해 10초 당 1,000개의 실행 환경
호출 페이로드(요청 및 응답)	요청 및 응답당 각각 6MB(동기식) 각 스트리밍된 응답에 대해 20MB(동기식. 스트리밍된 응답의 페이로드 크기는 기본값에서 늘릴 수 있습니다. 자세한 내용은 AWS Support에 문의하세요.) 256KB(비동기식) 요청 라인과 헤더 값의 총 합산 크기 1MB
스트리밍되는 응답의 대역폭	함수 응답의 처음 6MB에 대해 제한 없음 6MB보다 큰 응답의 경우 응답의 나머지 부분에 대해 2MBps
배포 패키지(.zip 파일 아카이브) 크기	50MB(직접 업로드용 압축 파일) 250MB(압축 해제됨) 이 할당량은 계층 및 사용자 지정 런타임 등 업로드하는 모든 파일에 적용됩니다. 3MB(콘솔 편집기)

Resource	Quota
컨테이너 이미지 설정 크기	16KB
컨테이너 이미지 코드 패키지 크기	10GB(모든 레이어를 포함한 최대 비압축 이미지 크기)
테스트 이벤트(콘솔 편집기)	10
/tmp 디렉터리 스토리지	512MB에서 10,240MB 사이, 1MB 단위로 증가
파일 설명자	1,024
실행 프로세스/스레드	1,024

Lambda API 요청

다음 할당량은 Lambda API 요청과 연관되어 있습니다.

리소스	할당량
리전별 함수당 호출 요청(동기)	실행 환경의 각 인스턴스가 초당 최대 10개의 요청을 처리할 수 있습니다. 즉, 총 호출 한도는 동시성 한도의 10배입니다. Lambda 함수 규모 조정 이행 섹션을 참조하세요.
리전별 함수당 호출 요청(비동기)	실행 환경의 각 인스턴스가 요청을 무제한으로 처리할 수 있습니다. 즉, 총 호출 한도는 함수에서 사용할 수 있는 동시성만을 기준으로 합니다. Lambda 함수 규모 조정 이행 섹션을 참조하세요.
함수 버전 또는 별칭당 호출 요청(초당 요청 수)	10배 할당된 프로비저닝된 동시성

리소스	할당량
	Note 이 할당량은 프로비저닝된 동시성을 사용하는 함수에만 적용됩니다.
GetFunction API 요청	초당 요청 100개입니다. 늘릴 수 없습니다.
GetPolicy API 요청	초당 요청 15개입니다. 늘릴 수 없습니다.
나머지 제어 플레인 API 요청(호출, GetFunction 및 GetPolicy 요청 제외)	모든 API에서 초당 요청 15개입니다(API별 초당 요청 15개 아님). 늘릴 수 없습니다.

기타 서비스

AWS Identity and Access Management(IAM), Amazon CloudFront(Lambda@Edge), Amazon Virtual Private Cloud(Amazon VPC) 등의 다른 서비스에 대한 할당량이 Lambda 함수에 영향을 줄 수 있습니다. 자세한 내용은 AWS 서비스 할당량을 Amazon Web Services 일반 참조에서, 그리고 다른 AWS 서비스의 이벤트로 Lambda 간접 호출를 참조하세요.

문서 이력

아래 표에 2018년 5월 이후 AWS Lambda 개발자 안내서의 중요한 변경 사항이 설명되어 있습니다. 이 설명서에 대한 업데이트 알림을 받으려면 RSS 피드를 구독하면 됩니다.

변경 사항	설명	날짜
새로운 리전의 SnapStart 지원	이제 유럽(스페인), 유럽(취리히), 아시아 태평양(멜버른), 아시아 태평양(하이데라바드) 및 중동(아랍에미리트) 리전에서 Lambda SnapStart를 사용할 수 있습니다.	2024년 1월 12일
AWS 관리형 정책 업데이트	Service Quotas에서 기존 AWS 관리형 정책(`AWSLambdaVPCAccessExecutionRole`)을 업데이트했습니다.	2024년 1월 5일
Python 3.12 런타임	Lambda는 이제 Python 3.12를 관리형 런타임 및 컨테이너 기본 이미지로 지원합니다. 자세한 내용은 AWS 컴퓨팅 블로그에서 Python 3.12 runtime now available in AWS Lambda을 참조하세요.	2023년 12월 14일
Java 21 런타임	Lambda는 이제 Java 21을 관리형 런타임 및 컨테이너 기본 이미지(`java21`)로 지원합니다.	2023년 11월 16일
Node.js 20.x 런타임	Lambda는 이제 Node.js 20을 관리형 런타임 및 컨테이너 기본 이미지(`nodejs20.x`)로 지원합니다. 자세한 내용은 현재 AWS Lambda에서 사용	2023년 11월 14일

	가능한 Node.js 20.x 런타임을 AWS Compute 블로그에서 참조하세요.	
provided.al2023 런타임	Lambda는 이제 Amazon Linux 2023을 관리형 런타임 및 컨테이너 기반 이미지로 지원합니다. 자세한 내용은 AWS Lambda을 위한 Amazon Linux 2023 런타임 소개를 AWS Compute 블로그에서 참조하세요.	2023년 11월 9일
듀얼 스택 서브넷에 대한 IPv6 지원	Lambda는 이제 듀얼 스택 서브넷으로의 아웃바운드 IPv6 트래픽을 지원합니다. 자세한 내용은 IPv6 지원을 참조하세요.	2023년 10월 12일
서버리스 함수 및 애플리케이션 테스트	클라우드에서 서버리스 함수 테스트를 디버깅하고 자동화하는 기법에 대해 알아보세요. 이제 Python 및 Typescript 언어 섹션에 포함된 테스트 장과 리소스가 있습니다. 자세한 내용은 서버리스 함수 및 애플리케이션 테스트를 참조하세요.	2023년 6월 16일
Ruby 3.2 런타임	이제 Lambda에서 Ruby 3.2에 대한 새 런타임을 지원합니다. 자세한 내용은 Ruby를 사용하여 Lambda 함수 빌드를 참조하세요.	2023년 6월 7일
응답 스트리밍	이제 Lambda에서 함수의 스트리밍 응답을 지원합니다. 자세한 내용은 응답을 스트리밍하도록 Lambda 함수 구성을 참조하세요.	2023년 4월 6일

비동기 호출 지표	Lambda는 비동기 호출 지표를 릴리스합니다. 자세한 정보는 비동기 호출 지표를 참조하세요.	2023년 2월 9일
런타임 버전 제어	Lambda는 보안 업데이트, 버그 수정 및 새로운 기능이 포함된 새로운 런타임 버전을 릴리스합니다. 이제 함수가 새 런타임 버전으로 업데이트되는 시점을 제어할 수 있습니다. 자세한 내용은 Lambda 런타임 업데이트를 참조하세요.	2023년 1월 23일
Lambda SnapStart	Lambda SnapStart를 사용하면 추가 리소스를 프로비저닝하거나 복잡한 성능 최적화 기능을 구현하지 않고도 Java 함수의 시작 시간을 줄일 수 있습니다. 자세한 내용은 Lambda SnapStart를 사용하여 시작 성능 개선을 참조하세요.	2022년 11월 28일
Node.js 18 런타임	이제 Lambda에서 Node.js 18에 대한 새 런타임을 지원합니다. Node.js 18은 Amazon Linux 2를 사용합니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드를 참조하세요.	2022년 11월 18일

lambda:SourceFunctionArn 조건 키	AWS 리소스에 대해 `lambda:SourceFunctionArn` 조건 키는 Lambda 함수의 ARN별로 리소스에 대한 액세스를 필터링합니다. 자세한 내용은 Working with Lambda execution environment credentials(Lambda 실행 환경 자격 증명 작업) 단원을 참조하십시오.	2022년 7월 1일
Node.js 16 런타임	이제 Lambda에서 Node.js 16에 대한 새 런타임을 지원합니다. Node.js 16은 Amazon Linux 2를 사용합니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드를 참조하세요.	2022년 5월 11일
Lambda 함수 URL	이제 Lambda에서 Lambda 함수의 전용 HTTP(S) 엔드포인트인 함수 URL을 지원합니다. 자세한 내용은 Lambda 함수 URL 섹션을 참조하세요.	2022년 4월 6일
AWS Lambda 콘솔의 공유 테스트 이벤트	Lambda는 이제 동일 AWS 계정에서 다른 사용자와 테스트 이벤트 공유를 지원합니다. 자세한 내용은 콘솔에서 Lambda 함수 테스트를 참조하세요.	2022년 3월 16일
리소스 기반 정책의 PrincipalOrgId	Lambda는 이제 AWS Organizations에서 조직에 권한을 부여할 수 있도록 지원합니다. 자세한 내용은 AWS Lambda에서 리소스 기반 정책 사용 섹션을 참조하세요.	2022년 3월 11일

.NET 6 런타임	이제 Lambda에서 .NET 6에 대한 새 런타임을 지원합니다. 자세한 내용은 Lambda 런타임을 참조하세요.	2022년 2월 23일
Kinesis, DynamoDB 및 Amazon SQS 이벤트 소스의 이벤트 필터링	이제 Lambda에서 Kinesis, DynamoDB 및 Amazon SQS 이벤트 소스의 이벤트 필터링을 지원합니다. 자세한 내용은 Lambda 이벤트 필터링 섹션을 참조하세요.	2021년 11월 24일
Amazon MSK 및 자체 관리형 Apache Kafka 이벤트 소스에 대한 mTLS 인증	Lambda는 이제 Amazon MSK 및 자체 관리형 Apache Kafka 이벤트 소스에 대한 mTLS 인증을 지원합니다. 자세한 내용은 Amazon MSK에서 Lambda 사용을 참조하세요.	2021년 11월 19일
Graviton2의 Lambda	이제 Lambda는 arm64 아키텍처를 사용하는 함수에 대해 Graviton2를 지원합니다. 자세한 내용은 Lambda 명령 세트 아키텍처를 참조하세요.	2021년 9월 29일
Python 3.9 런타임	이제 Lambda에서 Python 3.9에 대한 새 런타임을 지원합니다. 자세한 내용은 Lambda 런타임을 참조하세요.	2021년 8월 16일
Node.js, Python, Java에 대한 새 런타임 버전	Node.js, Python, Java에 대해 새 런타임 버전을 사용할 수 있습니다. 자세한 내용은 Lambda 런타임을 참조하세요.	2021년 7월 21일

Lambda에서 이벤트 소스로 RabbitMQ 지원	이제 Lambda에서 이벤트 소스로 RabbitMQ용 Amazon MQ를 지원합니다. Amazon MQ는 Apache ActiveMQ 및 RabbitMQ용 관리형 메시지 브로커 서비스로서, 클라우드에서 메시지 브로커를 쉽게 설치하고 운영할 수 있게 해줍니다. 자세한 내용은 Amazon MQ에서 Lambda 사용을 참조하세요.	2021년 7월 7일
Lambda의 자체 관리형 Kafka에 대한 SASL/PLAIN 인증	이제 Lambda의 자체 관리형 Kafka 소스에 SASL/PLAIN 인증 매커니즘이 지원됩니다. 자체 관리형 Kafka에 이미 SASL/PLAIN을 사용 중인 고객은 이제 Lambda를 사용해 인증 방식을 수정하지 않고 쉽게 고객 애플리케이션을 빌드할 수 있습니다. 자세한 내용은 자체 관리형 Apache Kafka에서 Lambda 사용 섹션을 참조하세요.	2021년 6월 29일
Lambda 익스텐션 API	Lambda 익스텐션이 일반에 제공됩니다. 익스텐션을 사용하면 Lambda 함수를 보강할 수 있습니다. Lambda 파트너가 제공하는 익스텐션을 사용하거나 자체적인 익스텐션을 만들 수 있습니다. 자세한 내용은 Lambda 익스텐션 API를 참조하세요.	2021년 5월 24일
새로운 Lambda 콘솔 환경	Lambda 콘솔은 재설계를 거쳐 성능과 일관성이 개선되었습니다.	2021년 3월 2일

Node.js 14 런타임	이제 Lambda에서 Node.js 14에 대한 새 런타임을 지원합니다. Node.js 14는 Amazon Linux 2를 사용합니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드를 참조하세요.	2021년 1월 27일
Lambda 컨테이너 이미지	이제 Lambda는 컨테이너 이미지로 정의된 함수를 지원합니다. 컨테이너 도구 모음의 유연성과 애플리케이션을 구축하는 Lambda의 민첩성 및 운영 간편성을 결합할 수 있습니다. 자세한 내용은 Lambda에서 컨테이너 이미지 사용을 참조하세요.	2020년 12월 1일
Lambda 함수에 대한 코드 서명	Lambda는 이제 코드 서명을 지원합니다. 관리자는 배포 시 서명된 코드만 허용하도록 Lambda 함수를 구성할 수 있습니다. Lambda는 서명을 검사하여 코드가 변경되거나 변조되지 않았는지 확인합니다. 또한 Lambda는 배포를 수락하기 전에 코드가 신뢰할 수 있는 개발자에 의해 서명되었는지 확인합니다. 자세한 내용은 Lambda 함수에 대한 코드 서명 구성을 참조하세요.	2020년 11월 23일

미리 보기: Lambda Runtime Logs API	이제 Lambda는 Runtime Logs API를 지원합니다. Lambda 익스텐션은 Logs API를 사용하여 실행 환경에서 로그 스트림을 구독할 수 있습니다. 자세한 내용은 Lambda Runtime Logs API를 참조하세요.	2020년 11월 12일
Amazon MQ에 대한 새로운 이벤트 소스	이제 Lambda에서 이벤트 소스로 RabbitMQ용 Amazon MQ를 지원합니다. Lambda 함수를 사용하여 Amazon MQ 메시지 브로커의 레코드를 처리할 수 있습니다. 자세한 내용은 Amazon MQ에서 Lambda 사용을 참조하세요.	2020년 11월 5일
미리 보기: Lambda 익스텐션 API	Lambda 익스텐션을 사용하면 Lambda 함수를 보강할 수 있습니다. Lambda 파트너가 제공하는 익스텐션을 사용하거나 자체적인 익스텐션을 만들 수 있습니다. 자세한 내용은 Lambda 익스텐션 API를 참조하세요.	2020년 10월 8일
AL2에서 Java 8 및 사용자 지정 런타임에 대한 지원	이제 Amazon Linux 2에서 Lambda가 Java 8과 사용자 지정 런타임을 지원합니다. 자세한 내용은 Lambda 런타임을 참조하세요.	2020년 8월 12일

Amazon Managed Streaming for Apache Kafka에 대한 새 이벤트 소스	이제 Lambda에서 이벤트 소스로 RabbitMQ용 Amazon MSK를 지원합니다. Amazon MSK에서 Lambda 함수를 사용하여 Kafka 주제의 레코드를 처리할 수 있습니다. 자세한 내용은 Amazon MSK에서 Lambda 사용을 참조하세요.	2020년 8월 11일
Amazon VPC 설정에 IAM 조건 키 사용	이제 VPC 설정에 Lambda 특정 조건 키를 사용할 수 있습니다. 예를 들어 조직의 모든 함수가 VPC에 연결되도록 요구할 수 있습니다. 또한 함수의 사용자가 사용할 수 있고 사용할 수 없는 서브넷 및 보안 그룹을 지정할 수도 있습니다. 자세한 내용은 IAM 함수에 맞게 VPC 구성을 참조하세요.	2020년 8월 10일
Kinesis HTTP/2 스트림 소비자를 위한 동시성 설정	이제 향상된 팬아웃(HTTP/2 스트림)을 사용하는 Kinesis 소비자에 대해 ParallelizationFactor, MaximumRetryAttempts, MaximumRecordAgeInSeconds, DestinationConfig, BisectBatchOnFunctionError와 같은 동시성 설정을 사용할 수 있습니다. . 자세한 내용은 Amazon Kinesis에서 AWS Lambda 사용을 참조하세요.	2020년 7월 7일

Kinesis HTTP/2 스트림 소비자를 위한 배치 기간	이제 HTTP/2 스트림에 대해 배치 기간(MaximumBatchingWindowInSeconds)을 구성할 수 있습니다. Lambda는 전체 배치를 수집하거나 배치 기간이 만료될 때까지 스트림에서 레코드를 읽습니다. 자세한 내용은 Amazon Kinesis에서 AWS Lambda 사용을 참조하세요.	2020년 6월 18일
Amazon EFS 파일 시스템 지원	이제 공유 네트워크 파일 액세스를 위해 Amazon EFS 파일 시스템을 Lambda 함수에 연결할 수 있습니다. 자세한 내용은 Lambda 함수에 대한 파일 시스템 액세스 구성을 참조하세요.	2020년 6월 16일
Lambda 콘솔의 AWS CDK 샘플 애플리케이션	이제 Lambda 콘솔에 TypeScript용 AWS Cloud Development Kit (AWS CDK)를 사용하는 샘플 애플리케이션이 포함됩니다. AWS CDK는 TypeScript, Python, Java 또는 .NET에서 애플리케이션 리소스를 정의할 수 있는 프레임워크입니다.	2020년 6월 1일
AWS Lambda에서 .NET Core 3.1.0 런타임 지원	AWS Lambda이제 에서 .NET Core 3.1.0 런타임을 지원합니다. 자세한 내용은 .NET Core CLI를 참조하세요.	2020년 3월 31일

API Gateway HTTP API 지원	HTTP API 지원을 포함해 API Gateway에서 Lambda 사용과 관련하여 문서를 업데이트하고 확장했습니다. AWS CloudFormation을 사용하여 API와 함수를 생성하는 샘플 애플리케이션을 추가했습니다. 자세한 내용은 Amazon API Gateway에서 Lambda 사용을 참조하세요.	2020년 3월 23일
Ruby 2.7	Ruby 2.7에 새로운 런타임 ruby2.7을 사용할 수 있습니다. 이는 Amazon Linux 2를 사용하는 최초의 Ruby 런타임입니다. 자세한 내용은 Ruby를 사용하여 Lambda 함수 빌드를 참조하세요.	2020년 2월 19일
동시성 지표	이제 Lambda는 모든 함수, 별칭 및 버전에 대한 `ConcurrentExecutions` 지표를 보고합니다. 함수의 모니터링 페이지에서 이 지표에 대한 그래프를 볼 수 있습니다. 이전에는 계정 수준과 예약된 동시성을 사용하는 함수에 대해서만 `ConcurrentExecutions` 가 보고되었습니다. 자세한 내용은 AWS Lambda 함수 지표를 참조하세요.	2020년 2월 18일

함수 상태 업데이트	이제 기본적으로 모든 함수에 함수 상태가 적용됩니다. 함수를 VPC에 연결하면 Lambda에서 공유 탄력적 네트워크 인터페이스를 생성합니다. 이를 통해 추가 네트워크 인터페이스를 생성하지 않고도 함수를 확장할 수 있습니다. 이 시간 동안에는 구성 업데이트 및 버전 게시를 포함하여 함수에 대한 추가 작업을 수행할 수 없습니다. 경우에 따라 호출도 영향을 받습니다. 함수의 현재 상태에 대한 세부 정보는 Lambda API에서 확인할 수 있습니다. 이 업데이트는 단계별로 릴리스됩니다. 자세한 내용은 AWS 컴퓨팅 블로그에서 VPC 네트워킹과 관련된 Lambda 상태 수명 주기 업데이트를 참조하세요. 함수 상태에 대한 자세한 내용은 AWS Lambda 함수 상태를 참조하세요.	2020년 1월 24일
함수 구성 API 출력 업데이트	VPC에 연결하는 함수의 StateReasonCode(InvalidSubnet, InvalidSecurityGroup) 및 LastUpdateStatusReasonCode(SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup)에 사유 코드를 추가했습니다. 함수 상태에 대한 자세한 내용은 AWS Lambda 함수 상태를 참조하세요.	2020년 1월 20일

프로비저닝된 동시성	이제 함수 버전이나 별칭에 대해 프로비저닝된 동시성을 할당할 수 있습니다. 프로비저닝된 동시성을 사용하면 지연 시간의 변동 없이 함수를 확장할 수 있습니다. 자세한 내용은 Lambda 함수의 동시성 관리를 참조하세요.	2019년 12월 3일
데이터베이스 프록시 생성	이제 Lambda 콘솔을 사용하여 Lambda 함수에 대한 데이터베이스 프록시를 생성할 수 있습니다. 데이터베이스 프록시를 사용하면 데이터베이스 연결을 소모하지 않고 함수가 높은 동시성 레벨에 도달할 수 있습니다. 자세한 내용은 Lambda 함수에 대한 데이터베이스 액세스 구성을 참조하세요.	2019년 12월 3일
기간 지표에 대한 백분위수 지원	이제 백분위수를 기준으로 기간 지표를 필터링할 수 있습니다. 자세한 내용은 AWS Lambda 지표를 참조하세요.	2019년 11월 26일
스트림 이벤트 소스의 동시성 증가	DynamoDB 스트림 및 Kinesis 스트림 이벤트 소스 매핑에 대한 새로운 옵션을 사용하면 각 샤드에서 한 번에 두 개 이상의 배치를 처리할 수 있습니다. 샤드당 동시 배치 수를 늘리면 함수의 동시성은 스트림에 있는 샤드 수의 최대 10배까지 증가할 수 있습니다. 자세한 내용은 Lambda 이벤트 소스 매핑을 참조하세요.	2019년 11월 25일

함수 상태	함수를 생성하거나 업데이트하면 Lambda가 해당 함수를 지원하는 리소스를 프로비저닝하는 동안 함수는 대기 중 상태로 전환됩니다. 함수를 VPC에 연결하면 Lambda는 함수가 호출될 때 네트워크 인터페이스를 생성하는 대신 즉시 공유 탄력적 네트워크 인터페이스를 생성할 수 있습니다. 이로 인해 VPC에 연결된 함수의 성능이 향상되지만 자동화를 업데이트해야 할 수 있습니다. 자세한 내용은 AWS Lambda 함수 상태를 참조하세요.	2019년 11월 25일
비동기 호출에 대한 오류 처리 옵션	비동기 호출에 새 구성 옵션을 사용할 수 있습니다. 재시도를 제한하고 최대 이벤트 기간을 설정하도록 Lambda를 구성할 수 있습니다. 자세한 내용은 비동기 호출에 대한 오류 처리 구성을 참조하세요.	2019년 11월 25일

스트림 이벤트 소스의 오류 처리	스트림에서 읽는 이벤트 소스 매핑에 새로운 구성 옵션을 사용할 수 있습니다. 재시도를 제한하고 최대 레코드 기간을 설정하도록 DynamoDB 스트림 및 Kinesis 스트림 이벤트 소스 매핑을 구성할 수 있습니다. 오류가 발생하면 재시도하기 전에 배치를 분할하고 실패한 배치의 호출 레코드를 대기열 또는 주제에 전송하도록 이벤트 소스 매핑을 구성할 수 있습니다. 자세한 내용은 Lambda 이벤트 소스 매핑을 참조하세요.	2019년 11월 25일
비동기 호출의 대상	이제 비동기 호출 레코드를 다른 서비스에 전송하도록 Lambda를 구성할 수 있습니다. 호출 레코드에는 이벤트, 컨텍스트 및 함수 응답에 대한 세부 정보가 포함되어 있습니다. SQS 대기열, SNS 주제, Lambda 함수 또는 EventBridge 이벤트 버스에 호출 레코드를 전송할 수 있습니다. 자세한 내용은 비동기 호출의 대상 구성을 참조하세요.	2019년 11월 25일
Node.js, Python 및 Java에 대한 새로운 런타임	Node.js 12, Python 3.8 및 Java 11에 대해 새로운 런타임을 사용할 수 있습니다. 자세한 내용은 Lambda 런타임을 참조하세요.	2019년 11월 18일

Amazon SQS 이벤트 소스에 대한 FIFO 대기열 지원	이제 FIFO(선입선출) 대기열에서 읽는 이벤트 소스 매핑을 만들 수 있습니다. 이전에는 표준 대기열만 지원되었습니다. 자세한 내용은 Amazon SQS에서 Lambda 사용을 참조하세요.	2019년 11월 18일
Lambda 콘솔에서 애플리케이션 생성	이제 Lambda 콘솔에서의 애플리케이션 생성을 일반적으로 사용할 수 있습니다. 지침은 Lambda 콘솔에서 애플리케이션 관리를 참조하세요.	2019년 10월 31일
Lambda 콘솔에서 애플리케이션 생성(베타)	이제 Lambda 콘솔에서 통합된 지속적 전달 파이프라인을 사용하여 Lambda 애플리케이션을 만들 수 있습니다. 이 콘솔은 자체 프로젝트의 시작점으로 사용할 수 있는 샘플 애플리케이션을 제공합니다. 소스 제어를 위해 AWS CodeCommit 및 GitHub 중에서 선택하세요. 리포지토리에 변경 사항을 푸시할 때마다 포함된 파이프라인이 자동으로 이들을 빌드 및 배포합니다. 지침은 Lambda 콘솔에서 애플리케이션 관리를 참조하세요.	2019년 10월 3일

VPC 연결 함수의 성능 개선	이제 Lambda는 Virtual Private Cloud(VPC) 서브넷의 모든 함수와 공유되는 새로운 유형의 탄력적 네트워크 인터페이스를 사용합니다. 함수를 VPC에 연결하면 Lambda는 선택한 보안 그룹 및 서브넷의 각 조합에 대해 네트워크 인터페이스를 생성합니다. 공유 네트워크 인터페이스를 사용할 수 있는 경우, 함수는 규모가 확장될 때 더 이상 추가 네트워크 인터페이스를 생성할 필요가 없습니다. 이렇게 하면 시작 시간이 대폭 단축됩니다. 자세한 정보는 VPC의 리소스에 액세스하도록 Lambda 함수 구성을 참조하세요.	2019년 9월 3일
스트림 배치 설정	이제 Amazon DynamoDB 및 Amazon Kinesis 이벤트 소스 매핑에 대한 배치 기간을 구성할 수 있습니다. 전체 배치를 사용할 수 있을 때까지 수신 레코드를 버퍼링하도록 최대 5분의 배치 기간을 구성합니다. 이렇게 하면 스트림의 사용 횟수가 적을 때 함수가 호출되는 횟수가 감소됩니다.	2019년 8월 29일
CloudWatch Logs Insights 통합	이제 Lambda 콘솔의 모니터링 페이지에 Amazon CloudWatch Logs Insights의 보고서가 포함됩니다. 자세한 내용은 AWS Lambda 콘솔에서의 모니터링 기능을 참조하세요.	2019년 6월 18일

Amazon Linux 2018.03	Lambda 실행 환경이 Amazon Linux 2018.03을 사용하도록 업데이트 중입니다. 자세한 내용은 실행 환경을 참조하세요.	2019년 5월 21일
Node.js 10	새 런타임은 Node.js 10, nodejs10.x에서 사용할 수 있습니다. 이 런타임은 Node.js 10.15를 사용하며 주기적으로 Node.js 10의 최신 포인트 릴리스로 업데이트됩니다. 또한 Node.js 10은 Amazon Linux 2를 사용하는 첫 번째 런타임입니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드를 참조하세요.	2019년 5월 13일
GetLayerVersionByArn API	GetLayerVersionByArn API를 사용하여 버전 ARN을 입력값으로 계층 버전 정보를 다운로드합니다. GetLayerVersion과는 달리, GetLayerVersionByArn을 이용하면 ARN을 분석하지 않고 바로 사용해 계층 이름과 버전 번호를 얻을 수 있습니다.	2019년 4월 25일
Ruby	이제 AWS Lambda은 새 런타임을 사용하여 Python 2.5를 지원합니다. 자세한 내용은 Ruby를 사용하여 Lambda 함수 빌드를 참조하세요.	2018년 11월 29일

계층	Lambda 계층을 사용하면 라이브러리, 사용자 지정 런타임 및 기타 종속 프로그램을 함수 코드와 별도로 패키징하고 배포할 수 있습니다. 사용 중인 계층을 귀하의 다른 계정 또는 전 세계 사용자들과 공유하세요. 자세한 내용은 Lambda 계층을 참조하세요.	2018년 11월 29일
사용자 지정 런타임	원하는 프로그래밍 언어로 Lambda 함수를 실행하는 사용자 지정 런타임을 빌드합니다. 자세한 내용은 사용자 지정 Lambda 런타임을 참조하세요.	2018년 11월 29일
Application Load Balancer 트리거	Elastic Load Balancing이 이제 Application Load Balancer의 대상으로 Lambda 함수를 지원합니다. 자세한 내용은 Application Load Balancer에서 Lambda 사용을 참조하세요.	2018년 11월 29일
Kinesis HTTP/2 스트림 소비자를 트리거로 사용	Kinesis HTTP/2 데이터 스트림 소비자를 사용하여 AWS Lambda로 이벤트를 보낼 수 있습니다. 스트림 소비자는 데이터 스트림의 각 샤드에서 전용 읽기 처리량이 있으며 HTTP/2를 사용하여 지연 시간을 최소화합니다. 자세한 내용은 Kinesis에서 Lambda 사용을 참조하세요.	2018년 11월 19일

Python 3.7	AWS Lambda가 이제 새 런타임을 사용하여 Python 3.7을 지원합니다. 자세한 내용은 Python을 사용하여 Lambda 함수 빌드를 참조하세요.	2018년 11월 19일
비동기 함수 호출에 대한 페이로드 한도 향상	Amazon SNS 트리거의 최대 메시지 크기에 맞추기 위해 비동기 호출에 대한 최대 페이로드 크기가 128KB에서 256KB로 향상되었습니다. 자세한 내용은 Lambda 할당량을 참조하세요.	2018년 11월 16일
AWS GovCloud(미국 동부) 리전	이제 AWS Lambda GovCloud(미국 동부) 리전에서 AWS을 사용할 수 있습니다.	2018년 11월 12일
별도의 개발자 안내서로 AWS SAM 항목 이전	많은 항목이 AWS Serverless Application Model(AWS SAM)을 사용하여 서버리스 애플리케이션을 빌드하는 내용을 중점적으로 다루고 있습니다. 이러한 주제가 AWS Serverless Application Model 개발자 안내서로 이전되었습니다.	2018년 10월 25일

콘솔에서 Lambda 애플리케이션 보기	Lambda 콘솔의 애플리케이션 페이지에서 Lambda 애플리케이션의 상태를 볼 수 있습니다. 이 페이지에는 AWS CloudFormation 스택의 상태가 표시되며, 스택의 리소스에 대한 자세한 정보를 확인할 수 있는 페이지 링크도 포함되어 있습니다. 또한 애플리케이션에 대한 집계 지표를 보고 사용자 지정 모니터링 대시보드를 생성할 수도 있습니다.	2018년 10월 11일
함수 실행 제한 시간	함수가 오래 실행될 수 있도록 구성 가능한 최대 실행 제한 시간이 5분에서 15분으로 늘어났습니다. 자세한 내용은 Lambda 제한을 참조하세요.	2018년 10월 10일
AWS Lambda에서 PowerShell Core에 대한 지원	이제 AWS Lambda에서 PowerShell Core 언어를 지원합니다. 자세한 내용은 PowerShell에서 Lambda 함수를 작성하기 위한 프로그래밍 모델을 참조하세요.	2018년 9월 11일
AWS Lambda에서 .NET Core 2.1.0 런타임 지원	이제 AWS Lambda에서 .NET Core 2.1.0 런타임을 지원합니다. 자세한 내용은 .NET Core CLI를 참조하세요.	2018년 7월 9일
RSS에서 현재 사용 가능한 업데이트	이제 RSS 피드를 구독하여 이 가이드의 릴리스를 따를 수 있습니다.	2018년 7월 5일

이벤트 소스로 Amazon SQS 지원	이제 AWS Lambda이 이벤트 소스로 Amazon Simple Queue Service(Amazon SQS)를 지원합니다. 자세한 내용은 Lambda 함수 호출을 참조하세요.	2018년 28월 6일
중국(닝샤) 리전	AWS Lambda를 이제 중국(닝샤) 지역에서 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2018년 6월 28일

이전 업데이트

다음 표에서는 2018년 6월 이전 AWS Lambda 개발자 안내서의 각 릴리스에서 변경된 중요 사항에 대해 설명합니다.

변경 사항	설명	날짜
Node.js 런타임 8.10에 대한 런타임 지원	AWS Lambda이제 에서 Node.js 런타임 버전 8.10을 지원합니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드 섹션을 참조하세요.	2018년 2월 4일
함수 및 별칭 개정 ID	AWS Lambda이제 에서 함수 버전 및 별칭에 대한 개정 ID를 지원합니다. 이들 ID를 사용하면 함수 버전 또는 별칭 리소스를 업데이트할 때 조건부 업데이트를 추적하고 적용할 수 있습니다.	2018년 1월 25일
Go 및 .NET 2.0에 대한 런타임 지원	AWS Lambda에서 Go 및 .NET 2.0에 대한 런타임 지원이 추가되었습니다. 자세한 내용은 Go를 사용하여 Lambda 함수 빌드 및 C#을 사용하여 Lambda 함수 빌드을(를) 참조하십시오.	2018년 1월 15일
콘솔 재설계	사용자 환경을 단순화하고 간편한 디버깅 및 함수 코드 수정 작업을 지원하는 Cloud9 코드 편집기가 추가된 새 Lambda 콘솔이 AWS Lambda에 도입되었습니다. 자세한 내용은	2017년 11월 30일

변경 사항	설명	날짜
	Lambda 콘솔 편집기를 사용하여 코드 편집 단원을 참조하십시오.	
개별 함수에 대한 동시성 한도 설정	AWS Lambda에서는 이제 개별 함수에 대한 동시성 한도 설정을 지원합니다. 자세한 내용은 함수에 대해 예약된 동시성 구성 섹션을 참조하세요.	2017년 11월 30일
별칭을 사용한 트래픽 이동	AWS Lambda에서는 이제 별칭을 사용한 트래픽 이동을 지원합니다. 자세한 내용은 Lambda 함수에 대한 롤링 배포 생성 섹션을 참조하세요.	2017년 11월 28일
점진적 코드 배포	이제 AWS Lambda는 코드 배포를 사용하여 Lambda 함수의 새 버전을 안전하게 배포할 수 있도록 지원합니다. 자세한 내용은 점진적 코드 배포를 참조하세요.	2017년 11월 28일
중국(베이징) 리전	이제 중국(베이징) 지역에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2017년 11월 9일
SAM Local 소개	AWS Lambda는 Lambda 런타임으로 업로드를 하기 전에 서버리스 애플리케이션을 로컬로 개발, 테스트 및 분석할 수 있는 환경을 제공하는 AWS CLI 도구인 SAM Local(지금은 SAM CLI라고도 함)을 도입했습니다. 자세한 내용은 서버리스 애플리케이션 테스트 및 디버그을 참조하세요.	2017년 8월 11일
캐나다(중부) 리전	AWS Lambda는 이제 캐나다(중부) 리전에서 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2017년 6월 22일
남아메리카(상파울루) 리전	이제 남아메리카(상파울루) 리전에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2017년 6월 6일

변경 사항	설명	날짜
AWS Lambda에 대한 지원AWS X-Ray	Lambda는 Lambda 애플리케이션에서 성능 문제를 감지, 분석 및 최적화할 수 있도록 허용하는 X-Ray에 대한 지원을 도입했습니다. 자세한 내용은 AWS X-Ray로 Lambda 함수 간접 호출 시각화 섹션을 참조하세요.	2017년 4월 19일
아시아 태평양(뭄바이) 리전	이제 아시아 태평양(뭄바이) 리전에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2017년 3월 28일
AWS Lambda는 Node.js 런타임 v6.10을 지원합니다.	AWS Lambda에서 Node.js 런타임 v6.10에 대한 지원이 추가되었습니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드 섹션을 참조하세요.	2017년 3월 22일
유럽(런던) 리전	이제 EU(런던) 리전에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2017년 2월 1일
AWS Lambda는 .NET 런타임, Lambda@Edge(프리뷰), 배달 못한 편지 대기열 및 서버리스 애플리케이션의 자동 배포를 지원합니다.	AWS Lambda에서 C#에 대한 지원이 추가되었습니다. 자세한 내용은 C#을 사용하여 Lambda 함수 빌드 섹션을 참조하세요. Lambda@Edge를 사용하면 CloudFront 이벤트에 대한 응답으로 AWS 엣지 로케이션에서 Lambda 함수를 실행할 수 있습니다. 자세한 내용은 CloudFront Lambda AWS Lambda @Edge 와 함께 사용 섹션을 참조하세요.	2016년 3월 12일
AWS Lambda가 지원되는 이벤트 소스로서 Amazon Lex를 추가합니다.	Lambda 및 Amazon Lex를 사용하여 Slack 및 Facebook 같은 다양한 서비스에서 채팅 봇을 신속하게 빌드할 수 있습니다. 자세한 내용은 Amazon Lex에서 AWS Lambda 사용 섹션을 참조하세요.	2016년 11월 30일
US West (N. California) Region	이제 미국 서부(캘리포니아 북부) 리전에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2016년 11월 21일

변경 사항	설명	날짜
Lambda 기반 애플리케이션을 생성 및 배포하고 Lambda 함수 구성 설정에서 환경 설정을 사용할 수 있도록 AWS SAM을 도입했습니다.	AWS SAM: 이제 AWS SAM을 사용하여 서버리스 애플리케이션 내에 리소스를 표현하기 위한 구문을 정의할 수 있습니다. 애플리케이션을 배포하려면 애플리케이션의 요소로서 필요한 리소스와 AWS CloudFormation 템플릿 파일(JSON 또는 YAML로 작성)에 연결된 권한 정책을 지정하고 배포 아티팩트를 패키징한 다음, 템플릿을 배포하기만 하면 됩니다. 자세한 내용은 AWS Lambda 애플리케이션 섹션을 참조하세요. 환경 변수: 환경 변수를 사용하여 함수 코드 밖에서 Lambda 함수에 대한 구성 설정을 지정할 수 있습니다. 자세한 내용은 Lambda 환경 변수를 사용하여 코드의 값 구성 섹션을 참조하세요.	2016년 11월 18일
아시아 태평양(서울) 리전	이제 아시아 태평양(서울) 리전에서 AWS Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2016년 8월 29일
아시아 태평양(시드니) 리전	이제 아시아 태평양(시드니) 리전에서 Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2016년 6월 23일
Lambda 콘솔에 대한 업데이트	Lambda 콘솔이 역할 생성 프로세스를 간소화하도록 업데이트되었습니다.	2016년 6월 23일
AWS Lambda이제에서 Node.js 런타임 v4.3을 지원합니다.	AWS Lambda에서 Node.js 런타임 v4.3에 대한 지원이 추가되었습니다. 자세한 내용은 Node.js를 사용하여 Lambda 함수 빌드 섹션을 참조하세요.	2016년 4월 07일
유럽(프랑크푸르트) 리전	이제 유럽(프랑크푸르트) 리전에서 Lambda를 사용할 수 있습니다. Lambda 리전 및 엔드포인트에 대한 자세한 내용은 AWS 일반 참조의 리전 및 엔드포인트를 참조하세요.	2016년 3월 14일
VPC 지원	이제 VPC의 리소스에 액세스하도록 Lambda 함수를 구성할 수 있습니다. 자세한 내용은 Lambda 함수에 Amazon VPC의 리소스에 대한 액세스 권한 부여 섹션을 참조하세요.	2016년 2월 11일

변경 사항	설명	날짜
Lambda 런타임이 업데이트되었습니다.	실행 환경이 업데이트되었습니다.	2015년 11월 4일
버전 관리 지원, Lambda 함수의 코드를 개발하기 위한 Python, 예약된 이벤트, 실행 시간 증가	이제 Python을 사용하여 Lambda 함수 코드를 배포할 수 있습니다. 자세한 내용은 Python을 사용하여 Lambda 함수 빌드 섹션을 참조하세요. 버전 관리: Lambda 함수에 대한 하나 이상의 버전을 유지할 수 있습니다. 버전 관리는 다른 환경(예를 들어 개발, 테스트 또는 프로덕션)에서 실행되는 Lambda 함수 버전을 관리할 수 있게 해줍니다. 자세한 내용은 Lambda 함수 버전 섹션을 참조하세요. 예정된 이벤트: Lambda 콘솔을 사용하여 코드를 정기적으로 호출할 수 있도록 Lambda를 설정할 수도 있습니다. 고정 비율(시간, 일 또는 주)을 지정하거나 cron 식을 지정할 수 있습니다. 자세한 내용은 Amazon EventBridge 스케줄러와 함께 Lambda 사용 단원을 참조하십시오. 실행 시간 증가: 대용량 데이터 수집 및 처리 작업 같은 함수를 더 오래 실행할 수 있도록 최대 5분 동안 Lambda 함수가 실행되도록 설정할 수 있습니다.	2015년 10월 08일
DynamoDB Streams 지원	이제 DynamoDB Streams는 상용 버전으로 사용할 수 있으며, DynamoDB가 출시된 모든 리전에서 사용이 가능합니다. 테이블에서 DynamoDB Streams를 활성화하고 테이블에 대한 트리거로서 Lambda 함수를 사용할 수 있습니다. 트리거는 DynamoDB 테이블에 대해 수행된 업데이트에 대한 응답에서 취할 수 있는 사용자 지정 작업입니다. 예제 연습은 섹션을 참조하세요자습서: Amazon DynamoDB Streams와 함께 AWS Lambda 사용	2015년 7월 14일

변경 사항	설명	날짜
이제 Lambda는 REST와 호환 가능한 클라이언트에서의 Lambda 함수 호출을 지원합니다.	지금까지는 웹, 모바일 또는 IoT 애플리케이션에서 Lambda 함수를 호출하려면 AWS SDK(예: AWS SDK for Java, AWS SDK for Android 또는 AWS SDK for iOS)가 필요했습니다. 이제는 Lambda가 Amazon API Gateway를 사용하여 생성할 수 있는 사용자 지정이 가능한 API를 통해 REST 호환 클라이언트에서 Lambda 함수를 호출할 수 있도록 지원합니다. Lambda 함수 엔드포인트 URL에 요청을 전송할 수 있습니다. 개방형 액세스를 허용하거나 AWS Identity and Access Management(IAM)를 활용하여 액세스를 인증하거나 API 키를 사용하여 Lambda 함수에 대한 다른 사람의 액세스를 측정할 수 있도록 엔드포인트에서 보안을 구성할 수 있습니다. 시작하기 연습의 예제는 Amazon API Gateway 엔드포인트를 사용하여 간접적으로 Lambda 함수 호출 섹션을 참조하세요. Amazon API Gateway에 대한 자세한 내용은 https://aws.amazon.com/api-gateway/를 참조하세요.	2015년 7월 09일
이제 Lambda 콘솔은 Lambda 함수를 손쉽게 생성하고 테스트할 수 있도록 블루프린트를 제공합니다.	Lambda 콘솔은 블루프린트 집합을 제공합니다. 각 블루프린트는 Lambda 기반 애플리케이션을 손쉽게 생성하는 데 사용할 수 있도록 Lambda 함수에 대한 샘플 이벤트 소스 구성과 샘플 코드를 제공합니다. 이제 모든 Lambda 시작하기 연습은 블루프린트를 사용합니다. 자세한 내용은 Lambda 시작하기 섹션을 참조하세요.	2015년 7월 09일
이제 Lambda는 Lambda 함수를 작성할 수 있도록 Java를 지원합니다.	이제 Java로 Lambda 코드를 작성할 수 있습니다. 자세한 내용은 Java를 사용하여 Lambda 함수 빌드 섹션을 참조하세요.	2015년 6월 15일

변경 사항	설명	날짜
이제 Lambda에서 Lambda 함수를 생성 또는 업데이트할 때 함수 .zip으로 Amazon S3 객체를 지정할 수 있습니다.	Lambda 함수 배포 패키지(.zip 파일)를 Lambda 함수를 생성하고 싶은 동일한 리전의 Amazon S3 버킷에 업로드할 수 있습니다. 그런 다음, Lambda 함수를 생성 또는 업데이트할 때 버킷 이름과 객체 키 이름을 지정할 수 있습니다.	2015년 5월 28일
이제 Lambda는 모바일 백엔드 지원을 추가한 상용 버전으로 사용할 수 있습니다.	Lambda는 프로덕션 용도의 상용 버전으로 사용할 수 있습니다. 이 릴리스에는 Lambda를 사용하여 모바일, 태블릿 및 사물 인터넷(IoT) 백엔드를 손쉽게 빌드함으로써 인프라 프로비저닝 또는 관리 없이도 자동으로 확장이 가능하게 해주는 기능들이 새로 추가되었습니다. Lambda는 실시간(동기식) 이벤트와 비동기식 이벤트를 모두 지원합니다. 추가 기능에는 보다 쉬워진 이벤트 소스 구성 및 관리 기능이 포함되어 있습니다. Lambda 함수에 대한 리소스 정책을 도입하여 권한 모델 및 프로그래밍 모델을 간소화했습니다. 이에 따라 설명서가 업데이트되었습니다. 자세한 내용은 다음 주제를 참조하세요. Lambda 시작하기 AWS Lambda	2015년 4월 9일
미리 보기 릴리스	AWS Lambda개발자 안내서 미리 보기 릴리스	2014년 11월 13일