開発者ガイド

AWS Cloud Development Kit (AWS CDK) v2

Version 2

AWS Cloud Development Kit (AWS CDK) v2: 開発者ガイド

Table of Contents

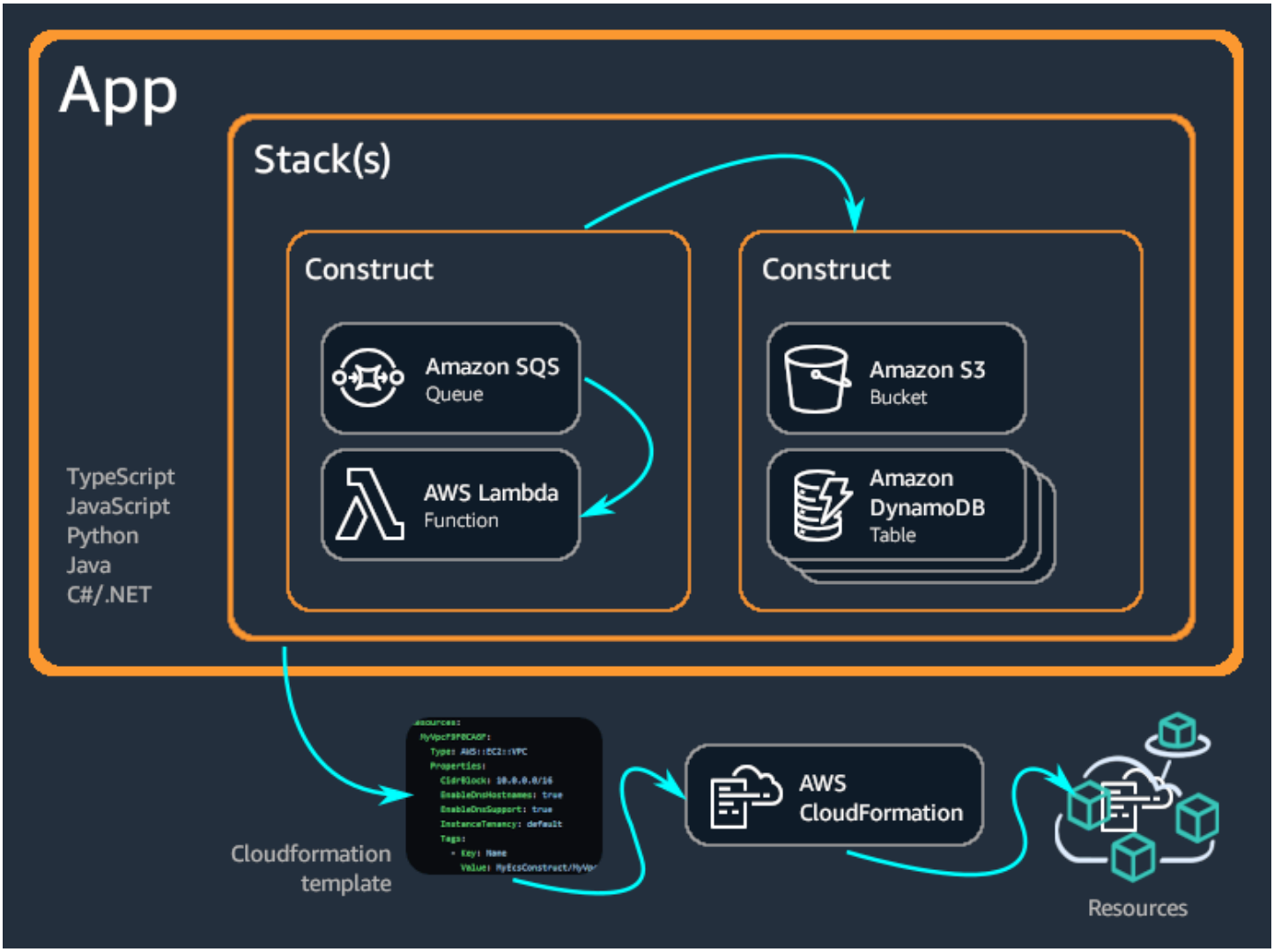
は何ですか AWS CDK?

AWS Cloud Development Kit (AWS CDK) は、 AWS CloudFormationクラウドインフラストラクチャをコードで定義し、それをプロビジョニングするためのオープンソースのソフトウェア開発フレームワークです。

は主に 2 AWS CDK つの部分で構成されています。

- AWS CDK Construct Library — コンストラクトと呼ばれる、あらかじめ作成されたモジュール式で再利用可能なコードの集まりで、使用、変更、統合することでインフラストラクチャを迅速に開発できます。 AWS CDK Construct Library の目標は、 AWS アプリケーションを構築する際にサービスを定義して統合する際に必要となる複雑さを軽減することです。 AWS
- AWS CDK Toolkit — CDK アプリを操作するためのコマンドラインツールです。 AWS CDK Toolkit を使用してプロジェクトを作成、管理、デプロイします。 AWS CDK

はTypeScript、、JavaScript、PythonJavaC#/.Net、 AWS CDK Goおよびをサポートします。サポートされているプログラミング言語のいずれかを使用して、コンストラクトと呼ばれる再利用可能なクラウドコンポーネントを定義できます。これらをまとめてスタックとアプリにします。次に、CDK AWS CloudFormation アプリケーションをにデプロイして、リソースをプロビジョニングまたは更新します。



トピック

- [の利点 AWS CDK]
- [その例 AWS CDK]
- [AWS CDK features]
- [次のステップ]
- [詳細はこちら]

の利点 AWS CDK

を使用すると、 AWS CDK プログラミング言語の優れた表現力を利用して、信頼性が高く、スケーラブルで費用対効果の高いアプリケーションをクラウドで開発できます。このアプローチには、次のような多くのメリットがあります。

インフラストラクチャー・アズ・コード (IaC) の開発と管理

インフラストラクチャー・アズ・コードを実践して、プログラム的、記述的、宣言的な方法でインフラストラクチャーを作成、デプロイ、保守しましょう。IaC では、開発者がコードを扱うのと同じようにインフラストラクチャーを扱うことができます。その結果、スケーラブルで構造化されたアプローチでインフラストラクチャーを管理できるようになります。IaC について詳しくは、 AWS ホワイトペーパーの「イントロダクション」の「コードとしてのインフラストラクチャ」を参照してください。 DevOps

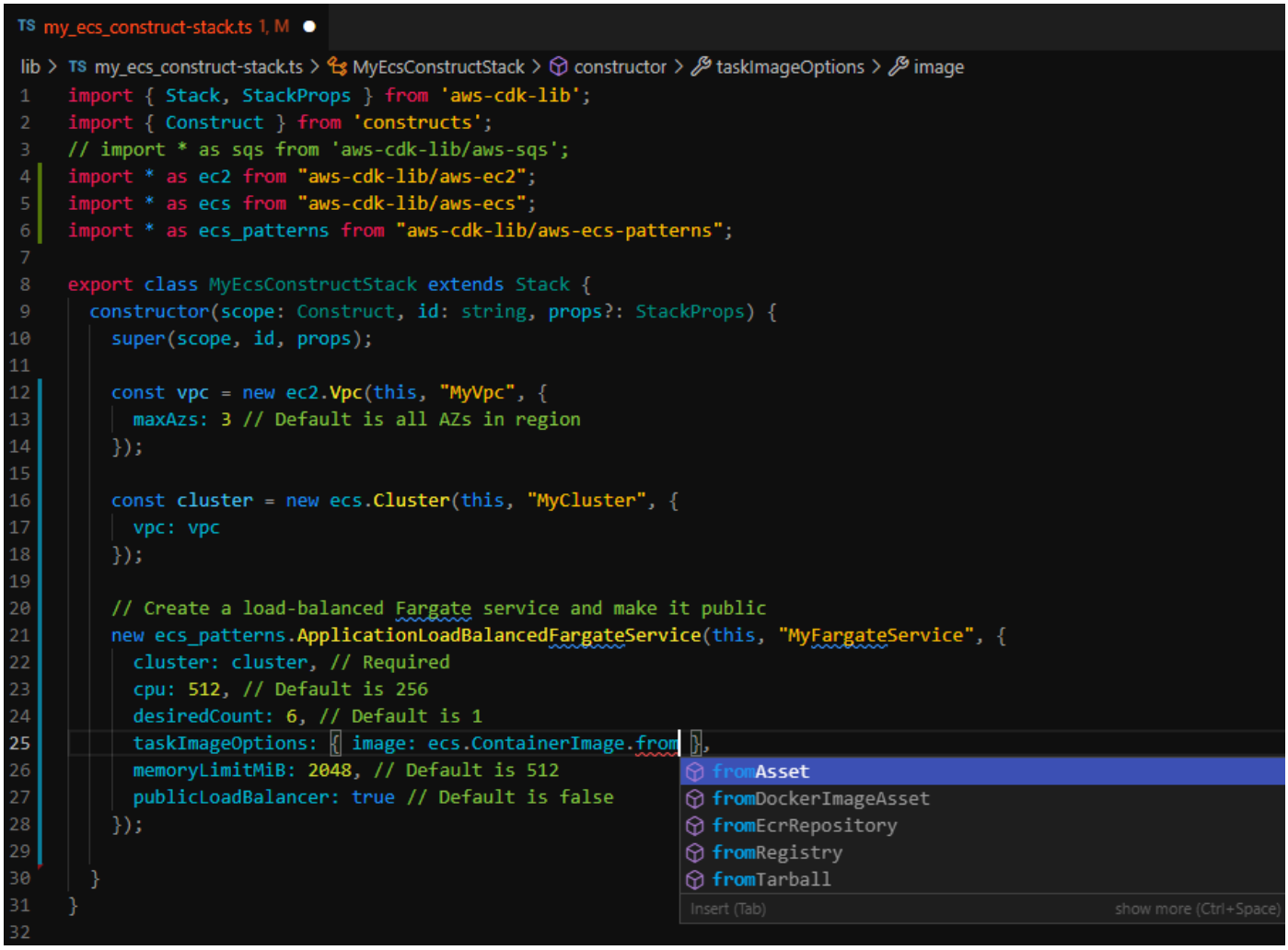
を使用すると AWS CDK、インフラストラクチャ、アプリケーションコード、および設定を 1 か所にまとめることができるため、あらゆるマイルストーンでクラウドに導入可能な完全なシステムを構築できます。コードレビュー、単体テスト、ソース管理などのソフトウェアエンジニアリングのベストプラクティスを採用して、インフラストラクチャーをより強固なものにしましょう。

汎用プログラミング言語を使用してクラウド・インフラストラクチャーを定義してください。

では AWS CDK、、、、、、のいずれかのプログラミング言語を使用してクラウドインフラストラクチャを定義できます。TypeScript JavaScript Python Java C#/.Net Go好みの言語を選択し、パラメーター、条件、ループ、構成、継承などのプログラミング要素を使用して、インフラストラクチャーの望ましい結果を定義します。

同じプログラミング言語を使用して、インフラストラクチャーとアプリケーションロジックを定義します。

シンタックスハイライトやインテリジェントなコード補完など、お好みの IDE (統合開発環境) でインフラストラクチャーを開発するメリットが得られます。



以下の方法でインフラストラクチャをデプロイできます。 AWS CloudFormation

AWS CDK AWS CloudFormation と統合して、インフラストラクチャをデプロイおよびプロビジョニングします。 AWS AWS CloudFormation は、 AWS のサービス にサービスをプロビジョニングするためのリソースとプロパティの設定を幅広くサポートするマネージャーです。 AWSを使用すると AWS CloudFormation、インフラストラクチャのデプロイを予測どおりに繰り返し実行でき、エラー発生時にロールバックできます。すでに使い慣れている場合は AWS CloudFormation、IaC 管理サービスを使い始めるときに新しい IaC 管理サービスを学ぶ必要はありません。 AWS CDK

コンストラクトを使えば、アプリケーションの開発をすぐに始められます。

コンストラクトと呼ばれる再利用可能なコンポーネントを使用したり共有したりすることで、開発をスピードアップできます。 AWS CloudFormation 低レベルの構成を使用して個々のリソースとそのプロパティを定義してください。高水準構造を使用すると、アプリケーションのより大き

なコンポーネントを迅速に定義できます。 AWS リソースには適切で安全なデフォルト設定を適用し、少ないコードでより多くのインフラストラクチャーを定義できます。

独自のユースケースに合わせてカスタマイズした独自の構成を作成して、組織全体で共有したり、一般に公開したりできます。

その例 AWS CDK

以下は、 AWS CDK AWS Fargate (Fargate) コンストラクトライブラリを使用して起動タイプの Amazon Elastic Container Service (Amazon ECS) サービスを作成する例です。この例の詳細については、「」を参照してください。the section called “ECS”

TypeScript

```
export class MyEcsConstructStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}
```

JavaScript

```
class MyEcsConstructStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}

module.exports = { MyEcsConstructStack }
```

Python

```
class MyEcsConstructStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
```

```
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is False
```

Java

```
public class MyEcsConstructStack extends Stack {

    public MyEcsConstructStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyEcsConstructStack(final Construct scope, final String id,
            StackProps props) {
        super(scope, id, props);

        Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                .vpc(vpc).build();

        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                .cluster(cluster)
                .cpu(512)
                .desiredCount(6)
                .taskImageOptions(
                       ApplicationLoadBalancedTaskImageOptions.builder()
                               .image(ContainerImage
                                       .fromRegistry("amazon/amazon-ecs-sample"))
                               .build()).memoryLimitMiB(2048)
                .publicLoadBalancer(true).build();
    }
}
```

C#

```
public class MyEcsConstructStack : Stack
{
```

```
    public MyEcsConstructStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        var vpc = new Vpc(this, "MyVpc", new VpcProps
        {
            MaxAzs = 3
        });

        var cluster = new Cluster(this, "MyCluster", new ClusterProps
        {
            Vpc = vpc
        });

        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
        {
            Cluster = cluster,
            Cpu = 512,
            DesiredCount = 6,
            TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
            {
                Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
            },
            MemoryLimitMiB = 2048,
            PublicLoadBalancer = true,
        });
    }
}
```

Go

```
func NewMyEcsConstructStack(scope constructs.Construct, id string, props
 *MyEcsConstructStackProps) awscdk.Stack {

 var sprops awscdk.StackProps

 if props != nil {
  sprops = props.StackProps
 }

 stack := awscdk.NewStack(scope, &id, &sprops)

 vpc := awsec2.NewVpc(stack, jsii.String("MyVpc"), &awsec2.VpcProps{
```

```
  MaxAzs: jsii.Number(3), // Default is all AZs in region
 })

 cluster := awsecs.NewCluster(stack, jsii.String("MyCluster"), &awsecs.ClusterProps{
  Vpc: vpc,
 })

 awsecspatterns.NewApplicationLoadBalancedFargateService(stack,
 jsii.String("MyFargateService"),
  &awsecspatterns.ApplicationLoadBalancedFargateServiceProps{
   Cluster:        cluster,           // required
   Cpu:            jsii.Number(512),  // default is 256
   DesiredCount:   jsii.Number(5),    // default is 1
   MemoryLimitMiB: jsii.Number(2048), // Default is 512
   TaskImageOptions: &awsecspatterns.ApplicationLoadBalancedTaskImageOptions{
    Image: awsecs.ContainerImage_FromRegistry(jsii.String("amazon/amazon-ecs-
sample"), nil),
   },
   PublicLoadBalancer: jsii.Bool(true), // Default is false
  })

 return stack

}
```

このクラスは 500 AWS CloudFormation 行を超えるテンプレートを生成します。 AWS CDK アプリをデプロイすると、以下の種類のリソースが 50 個以上生成されます。

- AWS::EC2::EIP
- AWS::EC2::InternetGateway
- AWS::EC2::NatGateway
- AWS::EC2::Route
- AWS::EC2::RouteTable
- AWS::EC2::SecurityGroup
- AWS::EC2::Subnet
- AWS::EC2::SubnetRouteTableAssociation
- AWS::EC2::VPCGatewayAttachment
- AWS::EC2::VPC

- AWS::ECS::Cluster
- AWS::ECS::Service
- AWS::ECS::TaskDefinition
- AWS::ElasticLoadBalancingV2::Listener
- AWS::ElasticLoadBalancingV2::LoadBalancer
- AWS::ElasticLoadBalancingV2::TargetGroup
- AWS::IAM::Policy
- AWS::IAM::Role
- AWS::Logs::LogGroup

AWS CDK features

AWS CDKGitHubレポジトリー

AWS CDK GitHub公式リポジトリについては、aws-cdk を参照してください。ここでは、課題の提出、ライセンスの表示、リリースの追跡などを行うことができます。

AWS CDK はオープンソースなので、より優れたツールになるよう皆さんに貢献していただきたいとチームでは奨励しています。詳細については、「への貢献」を参照してください。 AWS Cloud Development Kit (AWS CDK)

AWS CDK API リファレンス

AWS CDK コンストラクトライブラリには、CDK アプリケーションを定義し、アプリケーションに CDK コンストラクトを追加するための API が用意されています。詳細については、「 APIリファレンスAWS CDK」を参照してください。

コンストラクトプログラミングモデル

コンストラクト・プログラミング・モデル (CPM) は、 AWS CDK その背後にある概念を他の領域にも拡張します。CPM を使用するその他のツールには以下が含まれます。

- テラフォーム用 CDK (CDKTF)
- クベルネテス用 CDK (CDK8s)
- Projen、プロジェクト構成の構築用

コンストラクトハブ

Construct Hub は、 AWS CDK オープンソースのライブラリを検索、公開、共有できるオンラインレジストリです。

次のステップ

の使用を開始するには AWS CDK、を参照してくださいの開始方法 AWS CDK。

詳細はこちら

について引き続き学習するには AWS CDK、以下を参照してください。

- AWS CDK 主要概念を学ぶ— の重要な概念と用語 AWS CDK
- AWS CDK ワークショップ — を学び、使用するためのハンズオンワークショップ. AWS CDK
- AWS CDK パターン — 専門家がのために構築した、 AWS オープンソースのサーバーレスアーキテクチャパターン集。 AWS CDK AWS
- AWS CDK コード例 — GitHub サンプルプロジェクトのリポジトリ。 AWS CDK
- cdk.dev — コミュニティ主導のハブ。コミュニティワークスペースを含む。 AWS CDKSlack
- Awesome CDK — AWS CDK オープンソースプロジェクト、ガイド、ブログ、GitHubその他のリソースの厳選されたリストを含むリポジトリ。
- AWS ソリューション構築 — 検証済みのコードとしてのインフラストラクチャ (IaC) パターンで、本番環境ですぐに使えるアプリケーションに簡単に組み込めます。
- AWS デベロッパーツールブログ — ブログ投稿を絞り込んでいます。 AWS CDK
- AWS CDK オン Stack Overflow — aws-cdk on とタグ付けされた質問 Stack Overflow
- AWS CDK チュートリアル AWS Cloud9 — AWS CDK 開発環境でのの使用に関するチュートリアル。 AWS Cloud9

に関連するトピックの詳細については AWS CDK、以下を参照してください。

- AWS CloudFormation 概念 — AWS CDK はと連動するように構築されているので AWS CloudFormation、 AWS CloudFormation 重要な概念を学び、理解しておくことをおすすめします。
- AWS 用語集 — AWSさまざまな分野で使用される主要用語の定義

AWS CDK サーバーレスアプリケーションの開発とデプロイを簡素化するために使用できる関連ツールの詳細については、以下を参照してください。

- AWS Serverless Application Model— サーバーレスアプリケーションの構築と実行を簡素化および改善するオープンソースの開発者ツール。 AWS
- AWSChalice— でサーバーレスアプリを作成するためのフレームワーク。Python

AWS CDK 主要概念を学ぶ

の背後にある主要概念について説明します AWS Cloud Development Kit (AWS CDK)。

AWS CDK および IaC

AWS CDK は、コードを使用して AWS インフラストラクチャを管理するために使用できるオープンソースフレームワークです。このアプローチは、Infrastructure as Code (IaC) と呼ばれます。インフラストラクチャをコードとして管理およびプロビジョニングすることで、開発者がコードを処理するのと同じ方法でインフラストラクチャを扱います。これにより、バージョン管理やスケーラビリティなど、多くの利点が得られます。IaC の詳細については、「Infrastructure as Code とは」を参照してください。

AWS CDK および AWS CloudFormation

AWS CDK は と緊密に統合されています AWS CloudFormation。 AWS CloudFormation は、 でインフラストラクチャを管理およびプロビジョニングするために使用できるフルマネージドサービスです AWS。では AWS CloudFormation、 テンプレートでインフラストラクチャを定義し、 にデプロイします AWS CloudFormation。その後、 AWS CloudFormation サービスはテンプレートで定義された設定に従ってインフラストラクチャをプロビジョニングします。

AWS CloudFormation テンプレートは宣言型 です。つまり、インフラストラクチャの目的の状態または結果を宣言します。JSON または YAML を使用して、リソースとプロパティ を定義して AWS インフラストラクチャを AWS 宣言します。リソースは の AWS 多くのサービスを表し、プロパティはそれらのサービスの必要な設定を表します。テンプレートを にデプロイすると AWS CloudFormation、テンプレートの説明に従ってリソースと設定済みプロパティがプロビジョニングされます。

を使用すると AWS CDK、汎用プログラミング言語を使用してインフラストラクチャを必須に管理できます。目的の状態を宣言的に定義するだけでなく、目的の状態に到達するために必要なロジックまたはシーケンスを定義できます。例えば、 `if` ステートメントまたは条件ループを使用して、インフラストラクチャの目的の終了状態に到達する方法を決定できます。

で作成されたインフラストラクチャ AWS CDK は、最終的に変換されるか、テンプレートに AWS CloudFormation 合成され、 AWS CloudFormation サービスを使用してデプロイされます。そのため、 AWS CDK はインフラストラクチャの作成に異なるアプローチを提供しますが、 AWS リソー

ス設定の広範なサポートや堅牢なデプロイプロセスなど AWS CloudFormation、 の利点は引き続き得られます。

の詳細については AWS CloudFormation、「 AWS CloudFormation ユーザーガイド」の「 とは AWS CloudFormation」を参照してください。

AWS CDK および 抽象化

では AWS CloudFormation、リソースの設定方法の詳細をすべて定義する必要があります。これにより、インフラストラクチャを完全に制御できるという利点が得られます。ただし、これには、アクセス許可やイベント駆動型のインタラクションなど、リソース設定の詳細とリソース間の関係を含む堅牢なテンプレートを学習、理解、作成する必要があります。

を使用すると AWS CDK、リソース設定を同じように制御できます。ただし、 は強力な抽象化 AWS CDK も提供するため、インフラストラクチャ開発プロセスを高速化して簡素化できます。例えば、には AWS CDK 、適切なデフォルト設定を提供するコンストラクトと、定型コードを生成するヘルパーメソッドが含まれています。には、インフラストラクチャ管理アクションを実行する AWS CDK コマンドラインインターフェイス (AWS CDK CLI) などのツール AWS CDK も用意されています。

主要 AWS CDK 概念の詳細

とのやり取り AWS CDK

で を使用する場合 AWS CDK、主に AWS コンストラクトライブラリと を操作します AWS CDK CLI。

を使用した開発 AWS CDK

は、サポートされている任意のプログラミング言語 で記述 AWS CDK できます。まず、CDK プロジェクト から始めます。これには、アセット を含むフォルダとファイルの構造が含まれています。プロジェクト内で、CDK アプリケーション を作成します。アプリ内では、スタック を直接表す CloudFormation スタック を定義します。スタック内では、 コンストラクトを使用して AWS リソースとプロパティを定義します。

を使用したデプロイ AWS CDK

CDK アプリケーションを AWS 環境 にデプロイします。デプロイする前に、1 回限りのブートストラップを実行して環境を準備する必要があります。

詳細はこちら

AWS CDK 主要概念の詳細については、このセクションのトピックを参照してください。

サポートされているプログラミング言語

AWS Cloud Development Kit (AWS CDK) は、次の汎用プログラミング言語を最高水準でサポートしています。

- TypeScript
- JavaScript
- Python
- Java
- C#
- Go

JVM.NETCLR理論的には他の言語や言語も使用できますが、現時点では正式なサポートは提供していません。

> **Note**
>
> 現在、Goこのガイドには以外の説明やコード例は含まれていませんthe section called “Go で”。

AWS CDK は 1 TypeScript つの言語で開発されています。他の言語をサポートするために、 AWS CDK JSIIは言語バインディング生成というツールを利用しています。

AWS CDK 可能な限り自然で直感的な開発ができるよう、各言語の通常の規則を提供するよう努めています。たとえば、 AWS Construct Library モジュールはお好みの言語の標準リポジトリを使用して配布し、お客様はその言語の標準パッケージマネージャーを使用してインストールします。メソッドとプロパティの名前も、使用する言語で推奨されている命名パターンを使用して付けられます。

以下にコード例をいくつか示します。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
```

```
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

Python

```
bucket = s3.Bucket("MyBucket", bucket_name="my-bucket", versioned=True,
            website_redirect=s3.RedirectTarget(host_name="aws.amazon.com"))
```

Java

```
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket")
                      .versioned(true)
                      .websiteRedirect(new RedirectTarget.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true,
                      WebsiteRedirect = new RedirectTarget {
                              HostName = "aws.amazon.com"
                      }});
```

Go

```
bucket := awss3.NewBucket(scope, jsii.String("MyBucket"), &awss3.BucketProps {
 BucketName: jsii.String("my-bucket"),
 Versioned: jsii.Bool(true),
 WebsiteRedirect: &awss3.RedirectTarget {
  HostName: jsii.String("aws.amazon.com"),
 },
```

```
})
```

> **Note**
>
> これらのコードスニペットは説明のみを目的としています。これらは不完全で、そのままでは実行されません。

AWS Construct Library は、、NPMPyPiMaven、NuGetなどの各言語の標準パッケージ管理ツールを使用して配布されます。また、各言語に対応した AWS CDK API リファレンスのバージョンも提供しています。

AWS CDK お好みの言語でを使用できるように、このガイドには以下のサポート対象言語に関するトピックが含まれています。

- the section called “内 TypeScript”
- the section called “内 JavaScript”
- the section called “Python の場合”
- the section called “Java の場合”
- the section called “C# で”
- the section called “Go で”

TypeScript AWS CDKはが最初にサポートした言語で、 AWS CDK サンプルコードの多くはで記述されていますTypeScript。このガイドには、TypeScript AWS CDK サポートされている他の言語に合わせてコードを適合させる方法を具体的に示すトピックが含まれています。詳細については、「AWS CDKTypeScript と他の言語の比較」を参照してください。

AWS CDK プロジェクト

AWS Cloud Development Kit (AWS CDK) プロジェクトは、CDK コードを含むファイルとフォルダを表します。内容はプログラミング言語によって異なります。

AWS CDK プロジェクトは手動で作成することも、 AWS CDK コマンドラインインターフェイス (AWS CDK CLI) `cdk init` コマンドを使用して作成することもできます。このトピックでは、AWS CDK CLI によって作成されたファイルとフォルダのプロジェクト構造と命名規則について説明します。CDK プロジェクトは、ニーズに合わせてカスタマイズおよび整理できます。

> **Note**
>
> によって作成されたプロジェクト構造は、 AWS CDK CLI時間の経過とともにバージョンによって異なる場合があります。

トピック

ユニバーサルファイルとフォルダ

.git

`git` がインストールされている場合、 AWS CDK CLI はプロジェクトのGitリポジトリを自動的に初期化します。`.git` ディレクトリには、リポジトリに関する情報が含まれています。

.gitignore

が無視するファイルとフォルダを指定Gitするために使用するテキストファイル。

README.md

AWS CDK プロジェクトを管理するための基本的なガイダンスと重要な情報を提供するテキストファイル。必要に応じてこのファイルを変更し、CDK プロジェクトに関する重要な情報を文書化します。

cdk.json

の設定ファイル AWS CDK。このファイルは、アプリケーションの実行方法に関する の手順 AWS CDK CLIを提供します。

言語固有のファイルとフォルダ

以下のファイルとフォルダは、サポートされている各プログラミング言語に固有のものです。

TypeScript

以下は、 `cdk init --language typescript` コマンドを使用して `my-cdk-ts-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-ts-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-ts-project.ts
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-ts-project-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### my-cdk-ts-project.test.ts
### tsconfig.json
```

.npmignore

パッケージをnpmレジストリに公開するときに無視するファイルとフォルダを指定するファイル。このファイルは に似ていますが.gitignore、 npmパッケージに固有です。

ビン/my-cdk-ts-project.ts

アプリケーションファイルは CDK アプリを定義します。CDK プロジェクトには、1 つ以上のアプリケーションファイルを含めることができます。アプリケーションファイルは binフォルダに保存されます。

CDK アプリを定義する基本的なアプリケーションファイルの例を次に示します。

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MyCdkTsProjectStack } from '../lib/my-cdk-ts-project-stack';

const app = new cdk.App();
new MyCdkTsProjectStack(app, 'MyCdkTsProjectStack');
```

jest.config.js

の設定ファイルJest。 Jestは一般的なJavaScriptテストフレームワークです。

lib/my-cdk-ts-project-stack.ts

スタックファイルは CDK スタックを定義します。スタック内では、 コンストラクトを使用してリソースとプロパティを定義します AWS 。

CDK スタックを定義する基本的なスタックファイルの例を次に示します。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class MyCdkTsProjectStack extends cdk.Stack {
 constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

node_modules

Node.js プロジェクトの依存関係を含むプロジェクトの共通フォルダ。

package-lock.json

ファイルと連携して依存関係のバージョンを管理するメタデータpackage.jsonファイル。

package.json

Node.js プロジェクトで一般的に使用されるメタデータファイル。このファイルには、プロジェクト名、スクリプト定義、依存関係、その他のプロジェクトレベルのインポート情報など、CDK プロジェクトに関する情報が含まれています。

test/my-cdk-ts-project.test.ts

CDK プロジェクトのテストを整理するためのテストフォルダが作成されます。サンプルテストファイルも作成されます。

テストを実行する前に、 でテストを記述TypeScriptしJest、 を使用してTypeScriptコードをコンパイルできます。

tsconfig.json

コンパイラオプションとTypeScriptプロジェクト設定を指定するプロジェクトで使用される設定ファイル。

JavaScript

以下は、 `cdk init --language javascript` コマンドを使用して `my-cdk-js-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-js-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-js-project.js
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-js-project-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### my-cdk-js-project.test.js
```

.npmignore

パッケージをnpmレジストリに公開するときに無視するファイルとフォルダを指定するファイル。このファイルは に似ていますが `.gitignore`、 npmパッケージに固有です。

bin/my-cdk-js-project.js

アプリケーションファイルは CDK アプリを定義します。CDK プロジェクトには、1 つ以上のアプリケーションファイルを含めることができます。アプリケーションファイルは `bin`フォルダに保存されます。

CDK アプリを定義する基本的なアプリケーションファイルの例を次に示します。

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyCdkJsProjectStack } = require('../lib/my-cdk-js-project-stack');

const app = new cdk.App();
new MyCdkJsProjectStack(app, 'MyCdkJsProjectStack');
```

jest.config.js

の設定ファイルJest。 Jestは一般的なJavaScriptテストフレームワークです。

lib/my-cdk-js-project-stack.js

スタックファイルは CDK スタックを定義します。スタック内では、 コンストラクトを使用してリソースとプロパティを定義します AWS 。

CDK スタックを定義する基本的なスタックファイルの例を次に示します。

```
const { Stack, Duration } = require('aws-cdk-lib');

class MyCdkJsProjectStack extends Stack {
 constructor(scope, id, props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}

module.exports = { MyCdkJsProjectStack }
```

node_modules

Node.js プロジェクトの依存関係を含むプロジェクトの共通フォルダ。

package-lock.json

ファイルと連携して依存関係のバージョンを管理するメタデータpackage.jsonファイル。

package.json

Node.js プロジェクトで一般的に使用されるメタデータファイル。このファイルには、プロジェクト名、スクリプト定義、依存関係、その他のプロジェクトレベルのインポート情報など、CDK プロジェクトに関する情報が含まれています。

test/my-cdk-js-project.test.js

CDK プロジェクトのテストを整理するためのテストフォルダが作成されます。サンプルテストファイルも作成されます。

テストを実行する前に、 でテストJavaScriptを作成し、 Jest を使用してJavaScriptコードをコンパイルできます。

Python

以下は、 `cdk init --language python` コマンドを使用して `my-cdk-py-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-py-project
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### my_cdk_py_project
#   ### __init__.py
#   ### my_cdk_py_project_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
    ### __init__.py
    ### unit
```

.venv

CDK は、プロジェクトの仮想環境CLIを自動的に作成します。`.venv` ディレクトリは、この仮想環境を指します。

app.py

アプリケーションファイルは CDK アプリを定義します。CDK プロジェクトには、1 つ以上のアプリケーションファイルを含めることができます。

CDK アプリを定義する基本的なアプリケーションファイルの例を次に示します。

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

app = cdk.App()
MyCdkPyProjectStack(app, "MyCdkPyProjectStack")
```

```
app.synth()
```

my_cdk_py_project

スタックファイル を含むディレクトリ。CDK は、ここで以下CLIを作成します。

- `__init__.py` – 空のPythonパッケージ定義ファイル。
- `my_cdk_py_project` - CDK スタックを定義するファイル。次に、 コンストラクトを使用してスタック内の AWS リソースとプロパティを定義します。

スタックファイルの例を次に示します。

```
from aws_cdk import Stack

from constructs import Construct

class MyCdkPyProjectStack(Stack):
 def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
  super().__init__(scope, construct_id, **kwargs)

  # code that defines your resources and properties go here
```

requirements-dev.txt

に似ていますが`requirements.txt`、依存関係を本番環境ではなく開発目的で管理するために使用します。

requirements.txt

Python プロジェクトの依存関係を指定および管理するためにプロジェクトで使用される一般的な ファイル。

source.bat

Python 仮想環境を設定するWindowsために使用される のバッチファイル。

テスト

CDK プロジェクトのテストを含むディレクトリ。

ユニットテストの例を次に示します。

```
import aws_cdk as core
import aws_cdk.assertions as assertions
```

```
from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

def test_sqs_queue_created():
 app = core.App()
 stack = MyCdkPyProjectStack(app, "my-cdk-py-project")
 template = assertions.Template.from_stack(stack)

 template.has_resource_properties("AWS::SQS::Queue", {
  "VisibilityTimeout": 300
 })
```

Java

以下は、 `cdk init --language java` コマンドを使用して `my-cdk-java-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-java-project
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
    ### main
    ### test
```

pom.xml

CDK プロジェクトに関する設定情報とメタデータを含むファイル。このファイルは の一部ですMaven。

src/main

アプリケーションファイルとスタックファイルを含むディレクトリ。

アプリケーションファイルの例を次に示します。

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
```

```
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class MyCdkJavaProjectApp {
 public static void main(final String[] args) {
  App app = new App();

  new MyCdkJavaProjectStack(app, "MyCdkJavaProjectStack", StackProps.builder()
   .build());

  app.synth();
 }
}
```

スタックファイルの例を次に示します。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCdkJavaProjectStack extends Stack {
 public MyCdkJavaProjectStack(final Construct scope, final String id) {
  this(scope, id, null);
 }

 public MyCdkJavaProjectStack(final Construct scope, final String id, final
 StackProps props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

src/test

テストファイルを含むディレクトリ。以下に例を示します。

```
package com.myorg;

import software.amazon.awscdk.App;
```

```
import software.amazon.awscdk.assertions.Template;
import java.io.IOException;

import java.util.HashMap;

import org.junit.jupiter.api.Test;

public class MyCdkJavaProjectTest {

 @Test
 public void testStack() throws IOException {
  App app = new App();
  MyCdkJavaProjectStack stack = new MyCdkJavaProjectStack(app, "test");

  Template template = Template.fromStack(stack);

  template.hasResourceProperties("AWS::SQS::Queue", new HashMap<String, Number>()
 {{
   put("VisibilityTimeout", 300);
  }});
 }
}
```

C#

以下は、 `cdk init --language csharp` コマンドを使用して `my-cdk-csharp-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-csharp-project
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### MyCdkCsharpProject
    ### MyCdkCsharpProject.sln
```

src/MyCdkCsharpProject

アプリケーションファイルとスタックファイルを含むディレクトリ。

アプリケーションファイルの例を次に示します。

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyCdkCsharpProject
{
 sealed class Program
 {
  public static void Main(string[] args)
  {
   var app = new App();
   new MyCdkCsharpProjectStack(app, "MyCdkCsharpProjectStack", new StackProps{});
   app.Synth();
  }
 }
}
```

スタックファイルの例を次に示します。

```
using Amazon.CDK;
using Constructs;

namespace MyCdkCsharpProject
{
 public class MyCdkCsharpProjectStack : Stack
 {
  internal MyCdkCsharpProjectStack(Construct scope, string id, IStackProps props
 = null) : base(scope, id, props)
  {
   // code that defines your resources and properties go here
  }
 }
}
```

このディレクトリには、以下も含まれています。

- GlobalSuppressions.cs – プロジェクト全体で特定のコンパイラの警告またはエラーを抑制するために使用されるファイル。
- .csproj – プロジェクト設定、依存関係、ビルド設定を定義するために使用される XML ベースのファイル。

src/MyCdkCsharpProject.sln

Microsoft Visual Studio Solution File は、関連プロジェクトの整理と管理に使用されます。

Go

以下は、 `cdk init --language go` コマンドを使用して `my-cdk-go-project` ディレクトリに作成されたプロジェクトの例です。

```
my-cdk-go-project
### .git
### .gitignore
### README.md
### cdk.json
### go.mod
### my-cdk-go-project.go
### my-cdk-go-project_test.go
```

go.mod

モジュール情報を含み、Goプロジェクトの依存関係とバージョニングを管理するために使用されるファイル。

my-cdk-go-project.go

CDK アプリケーションとスタックを定義するファイル。

以下に例を示します。

```
package main
import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type MyCdkGoProjectStackProps struct {
 awscdk.StackProps
}

func NewMyCdkGoProjectStack(scope constructs.Construct, id string, props
 *MyCdkGoProjectStackProps) awscdk.Stack {
```

```
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)
 // The code that defines your resources and properties go here

  return stack
}

func main() {
 defer jsii.Close()
 app := awscdk.NewApp(nil)
 NewMyCdkGoProjectStack(app, "MyCdkGoProjectStack", &MyCdkGoProjectStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })
 app.Synth(nil)
}

func env() *awscdk.Environment {

 return nil
}
```

my-cdk-go-project_test.go

サンプルテストを定義するファイル。

以下に例を示します。

```
package main

import (
 "testing"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/assertions"
 "github.com/aws/jsii-runtime-go"
)

func TestMyCdkGoProjectStack(t *testing.T) {
```

```
 // GIVEN
 app := awscdk.NewApp(nil)

 // WHEN
 stack := NewMyCdkGoProjectStack(app, "MyStack", nil)

 // THEN
 template := assertions.Template_FromStack(stack, nil)
 template.HasResourceProperties(jsii.String("AWS::SQS::Queue"),
 map[string]interface{}{
  "VisibilityTimeout": 300,
 })
}
```

AWS CDK アプリ

AWS Cloud Development Kit (AWS CDK) アプリケーションは、1 つ以上の CDK スタックのコレクションです。スタックは、 AWS リソースとプロパティを定義する 1 つ以上のコンストラクトのコレクションです。したがって、スタックとコンストラクトの全体的なグループ化は CDK アプリと呼ばれます。

トピック

- アプリケーションの定義
- コンストラクトツリー
- アプリケーションのライフサイクル

アプリケーションの定義

プロジェクト のアプリケーションファイルでアプリケーションインスタンスを定義して、アプリケーションを作成します???。これを行うには、 Appコンストラクトライブラリから AWS コンストラクトをインポートして使用します。App コンストラクトには初期化引数は必要ありません。ルートとして使用できるコンストラクトはこれだけです。

Construct Library の AWS Appおよび Stack クラスは、一意のコンストラクトです。他のコンストラクトと比較すると、リソース AWS は単独で設定されません。代わりに、他のコンストラクトのコンテキストを提供するために使用されます。 AWS リソースを表すすべてのコンストラクトは、Stackコンストラクトの範囲内で直接または間接的に定義する必要があります。 Stackコンストラクトはコンストラクトの範囲内で定義されますApp。

その後、アプリケーションが合成され、スタックの AWS CloudFormation テンプレートが作成されます。以下に例を示します。

TypeScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

JavaScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

Python

```
app = App()
MyFirstStack(app, "hello-cdk")
app.synth()
```

Java

```
App app = new App();
new MyFirstStack(app, "hello-cdk");
app.synth();
```

C#

```
var app = new App();
new MyFirstStack(app, "hello-cdk");
app.Synth();
```

Go

```
app := awscdk.NewApp(nil)

MyFirstStack(app, "MyFirstStack", &MyFirstStackProps{
  awscdk.StackProps{
    Env: env(),
  },
})
```

```
app.Synth(nil)
```

1 つのアプリケーション内のスタックは、相互のリソースとプロパティを簡単に参照できます。は、スタック間の依存関係を AWS CDK 推測して、正しい順序でデプロイできるようにします。1 つのcdk deployコマンドで、アプリケーション内の一部またはすべてのスタックをデプロイできます。

コンストラクトツリー

コンストラクトは、すべてのコンストラクトに渡される scope引数を使用して他のコンストラクトの内部で定義され、 App クラスをルートとして使用します。このようにして、 AWS CDK アプリケーションはコンストラクトツリー と呼ばれるコンストラクトの階層を定義します。

このツリーのルートは、 Appクラスのインスタンスであるアプリです。アプリ内で、1 つ以上のスタックをインスタンス化します。スタック内では、 コンストラクトをインスタンス化します。それ自体がリソースやその他のコンストラクトをインスタンス化する場合などに、ツリーの下に置かれます。

コンストラクトは、常に別のコンストラクトの範囲内で明示的に定義され、コンストラクト間の関係を作成します。ほとんどの場合、スコープとして this (Python では self) を渡し、新しいコンストラクトが現在のコンストラクトの子であることを示します。意図したパターンは、 からコンストラクトを導き出しConstruct、コンストラクターで使用するコンストラクトをインスタンス化することです。

スコープを明示的に渡すと、各コンストラクトはツリーにそれ自体を追加でき、この動作はConstruct基本クラス に完全に含まれています。これは、 でサポートされているすべての言語で同じように機能 AWS CDK し、追加のカスタマイズは必要ありません。

> **Important**
>
> 厳密には、コンストラクトをインスタンス化thisするときに 以外のスコープを渡すことができます。ツリー内の任意の場所、または同じアプリ内の別のスタックにコンストラクトを追加できます。例えば、引数として渡されたスコープにコンストラクトを追加する混合形式の関数を記述できます。ここでの実用的な難点は、コンストラクトに選択した IDs が他のユーザーのスコープ内で一意であることを簡単に保証できないことです。また、このプラクティスにより、コードの理解、保守、再利用が難しくなります。引scope数を悪用することなく、インテントを表現する方法を見つけるのがほとんどの場合です。

AWS CDK は、ツリーのルートから各子コンストラクトへのパス内のすべてのコンストラクトの IDs を使用して、 に必要な一意の IDs を生成します AWS CloudFormation。このアプローチでは、コンストラクト IDsは、ネイティブ のようにスタック全体ではなく、スコープ内で一意である必要があるだけです AWS CloudFormation。ただし、コンストラクトを別のスコープに移動すると、生成されたスタック固有の ID が変更され、同じリソースと見な AWS CloudFormation されません。

コンストラクトツリーは、 AWS CDK コードで定義したコンストラクトとは別のものです。ただし、ツリー内のそのコンストラクトを表すノードへの参照である任意のコンストラクトの node 属性からアクセスできます。各ノードは、ツリーのルートとノードの親スコープおよび子へのアクセスを提供する の属性であるNodeインスタンスです。

1. node.children – コンストラクトの直接の子。
2. node.id – スコープ内のコンストラクトの識別子。
3. node.path – すべての親の IDs を含むコンストラクトのフルパス。
4. node.root – コンストラクトツリー (アプリ) のルート。
5. node.scope – コンストラクトのスコープ (親) 。ノードがルートの場合は未定義。
6. node.scopes - ルートまでのコンストラクトのすべての親。
7. node.uniqueId – ツリー内のこのコンストラクトの一意の英数字識別子 (デフォルトでは、node.pathとハッシュから生成されます) 。

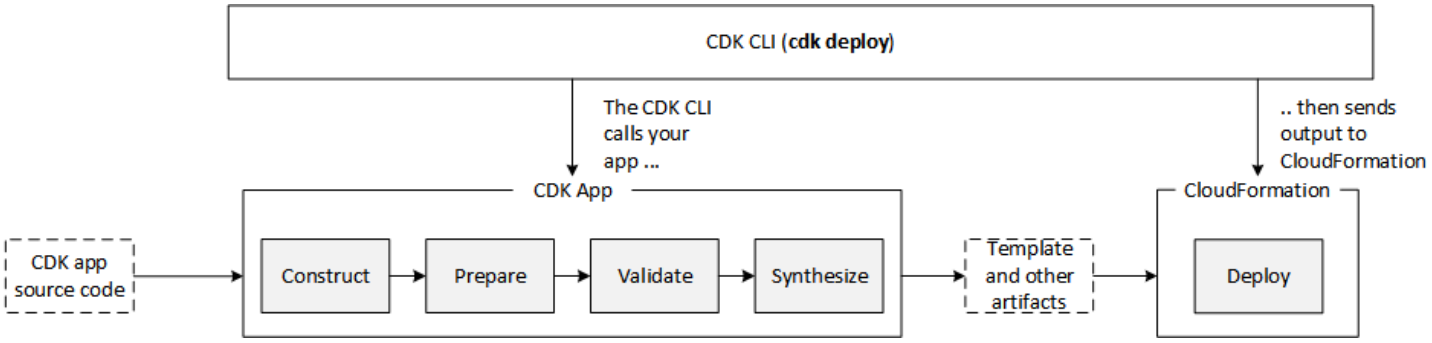
コンストラクトツリーは、コンストラクトが最終的な AWS CloudFormation テンプレートのリソースに合成される暗黙的な順序を定義します。あるリソースを別のリソースの前に作成する必要がある場合、 AWS CloudFormation または Construct Library AWS は一般的に依存関係を推測します。次に、リソースが正しい順序で作成されていることを確認します。

を使用して、2 つのノード間に明示的な依存関係を追加することもできますnode.addDependency()。詳細については、 AWS CDK API リファレンスの「依存関係」を参照してください。

AWS CDK は、コンストラクトツリー内のすべてのノードにアクセスし、各ノードに対してオペレーションを実行する簡単な方法を提供します。詳細については、「the section called “側面”」を参照してください。

アプリケーションのライフサイクル

CDK アプリをデプロイすると、次のフェーズが実行されます。これは、アプリケーションライフサイクル と呼ばれます。



AWS CDK アプリケーションは、ライフサイクルの次のフェーズを実行します。

- 構成 (または初期化) — コードは、定義されたすべての構成をインスタンス化し、それらをリンクします。この段階では、すべてのコンストラクト (アプリケーション、スタック、子コンストラクト) がインスタンス化され、コンストラクターチェーンが実行されます。アプリコードのほとんどはこの段階で実行されます。
- 準備 — `prepare`メソッドを実装したすべてのコンストラクトは、最終状態をセットアップするための最終変更ラウンドに参加します。準備フェーズは自動的に行われます。ユーザーとして、このフェーズからのフィードバックは表示されません。「prepare」フックを使用する必要はほとんどなく、通常はお勧めしません。オペレーションの順序が動作に影響を与える可能性があるため、このフェーズでコンストラクトツリーをミューテーションする場合は注意が必要です。
- 検証 — `validate`メソッドを実装したすべてのコンストラクトは、それらが正しくデプロイされる状態であることを確認するために、自分自身を検証できます。このフェーズで検証に失敗した場合は、通知を受け取ります。通常、できるだけ早く (通常は入力を取得するとすぐに) 検証を行い、できるだけ早く例外をスローすることをお勧めします。検証を早期に実行すると、スタックトレースがより正確になり、コードを安全に実行し続けることができるため、信頼性が向上します。
- Synthesis – これは、 AWS CDK アプリケーションの実行の最終段階です。これは への呼び出しによってトリガーされ`app.synth()`、コンストラクトツリーを横断して、すべてのコンストラクトで `synthesize`メソッドを呼び出します。を実装するコンストラクト`synthesize`は合成に参加し、結果のクラウドアセンブリにデプロイアーティファクトを出力できます。これらのアーティファクトには、 AWS CloudFormation テンプレート、 AWS Lambda アプリケーションバンドル、ファイルとDockerイメージアセット、およびその他のデプロイアーティファクトが含まれます。はこのフェーズの出力the section called “クラウドアセンブリ”について説明します。ほとんどの場合、 `synthesize`メソッドを実装する必要はありません。
- デプロイ — このフェーズでは AWS CDK CLI、 は合成フェーズによって生成されたデプロイアーティファクトクラウドアセンブリを取得し、 AWS 環境にデプロイします。アセットを Amazon S3 と Amazon ECR にアップロードするか、必要な場所にアップロードします。次に、 AWS CloudFormation デプロイを開始してアプリケーションをデプロイし、リソースを作成します。

AWS CloudFormation デプロイフェーズが開始されるまでに、 AWS CDK アプリケーションはすでに終了し、終了しています。これにより、以下の影響があります。

- AWS CDK アプリケーションは、作成中のリソースやデプロイ全体の終了など、デプロイ中に発生するイベントに対応できません。デプロイフェーズでコードを実行するには、カスタムリソース として AWS CloudFormation テンプレートにコードを挿入する必要があります。アプリケーションへのカスタムリソースの追加の詳細については、AWS CloudFormation モジュール または custom-resource の例を参照してください。
- AWS CDK アプリは、実行時に認識できない値を操作する必要がある場合があります。例えば、アプリケーションが AWS CDK 自動生成された名前で Amazon S3 バケットを定義し、`bucket.bucketName` (Python: `bucket_name`) 属性を取得した場合、その値はデプロイされたバケットの名前ではありません。代わりに、 `Token`値を取得します。特定の値が使用可能かどうかを判断するには、 `cdk.isUnresolved(value)` (Python:) を呼び出しますis_unresolved。詳細については、「the section called “トークン”」を参照してください。

クラウドアセンブリ

への呼び出しapp.synth()は、アプリケーションからクラウドアセンブリを合成 AWS CDK するように に指示するものです。通常、クラウドアセンブリと直接やり取りすることはありません。これらは、アプリケーションをクラウド環境にデプロイするために必要なすべてを含むファイルです。例えば、アプリケーション内の各スタックの AWS CloudFormation テンプレートが含まれています。また、アプリで参照するファイルアセットや Docker イメージのコピーも含まれます。

クラウドアセンブリのフォーマット方法の詳細については、クラウドアセンブリ仕様を参照してください。

AWS CDK アプリケーションが作成するクラウドアセンブリを操作するには、通常 を使用します AWS CDK CLI。ただし、クラウドアセンブリ形式を読み取ることができる任意のツールを使用してアプリケーションをデプロイできます。

アプリケーションの実行

CDK は、 AWS CDK アプリの実行方法を知るCLI必要があります。`cdk init` コマンドを使用してテンプレートからプロジェクトを作成した場合、アプリケーションの `cdk.json` ファイルには appキーが含まれます。このキーは、アプリケーションが書き込まれる言語に必要なコマンドを指定します。言語にコンパイルが必要な場合、コマンドラインはアプリケーションを実行する前にこのステップを実行するため、忘れることはありません。

TypeScript

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-app.ts"
}
```

JavaScript

```
{
  "app": "node bin/my-app.js"
}
```

Python

```
{
    "app": "python app.py"
}
```

Java

```
{
  "app": "mvn -e -q compile exec:java"
}
```

C#

```
{
  "app": "dotnet run -p src/MyApp/MyApp.csproj"
}
```

Go

```
{
  "app": "go mod download && go run my-app.go"
}
```

CDK を使用してプロジェクトを作成しなかった場合CLI、または で指定されたコマンドラインを上書きする場合はcdk.json、cdkコマンドを発行するときに --appオプションを使用できます。

```
$ cdk --app 'executable' cdk-command ...
```

コマンドの実行###部分は、CDK アプリケーションを実行するために実行する必要があるコマンドを示します。示されているように引用符を使用します。このようなコマンドにはスペースが含まれているためです。*cdk-command* は、アプリでCLI何をするかを CDK に指示deployする synthまたは のようなサブコマンドです。これに従い、そのサブコマンドに必要な追加オプションを指定します。

AWS CDK CLI は、既に合成されたクラウドアセンブリと直接やり取りすることもできます。そのためには、クラウドアセンブリが保存されているディレクトリを に渡します--app。次の例では、 に保存されているクラウドアセンブリで定義されたスタックを一覧表示します`./my-cloud-assembly`。

```
$ cdk --app ./my-cloud-assembly ls
```

スタック

AWS Cloud Development Kit (AWS CDK) スタックは、 AWS リソースを定義する 1 つ以上のコンストラクトのコレクションです。各 CDK スタックは CDK アプリの AWS CloudFormation スタックを表します。デプロイ時に、スタック内のコンストラクトは、 AWS CloudFormation スタックと呼ばれる単一のユニットとしてプロビジョニングされます。 AWS CloudFormation スタックの詳細については、「 AWS CloudFormation ユーザーガイド」の「スタックの使用」を参照してください。

CDK スタックは AWS CloudFormation スタックを通じて実装されるため、 AWS CloudFormation クォータと制限が適用されます。詳細については、AWS CloudFormation 「 クォータ」を参照してください。

トピック

- スタックの定義
- スタックの操作

スタックの定義

スタックはアプリケーションのコンテキスト内で定義されます。スタックは、 AWS コンストラクトライブラリの `Stack` クラスを使用して定義します。スタックは、次のいずれかの方法で定義できます。

- アプリのスコープ内で直接。
- ツリー内の任意のコンストラクトによって間接的に。

次の例では、2 つのスタックを含む CDK アプリケーションを定義します。

TypeScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

JavaScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

Python

```
app = App()

MyFirstStack(app, 'stack1')
MySecondStack(app, 'stack2')

app.synth()
```

Java

```
App app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.synth();
```

C#

```
var app = new App();
```

```
new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.Synth();
```

次の例は、別のファイルでスタックを定義するための一般的なパターンです。ここでは、 Stack クラスを拡張または継承し、scope、、idおよび を受け入れるコンストラクターを定義しますprops。次に、受信した 、、scopeidおよび superで を使用してベースStackクラスコンストラクタを呼び出しますprops。

TypeScript

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    //...
  }
}
```

JavaScript

```
class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    //...
  }
}
```

Python

```
class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # ...
```

Java

```
public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // ...
    }
}
```

C#

```
public class HelloCdkStack : Stack
{
    public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        //...
    }
}
```

Go

```
func HelloCdkStack(scope constructs.Construct, id string, props *HelloCdkStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

  return stack
}
```

次の例では、単一の Amazon S3 バケットを含む という名前のスタッククラスを宣言`MyFirstStack`します。

TypeScript

```
class MyFirstStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

JavaScript

```
class MyFirstStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket');
  }
}
```

Python

```
class MyFirstStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket")
```

Java

```
public class MyFirstStack extends Stack {
    public MyFirstStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyFirstStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        new Bucket(this, "MyFirstBucket");
    }
```

```
}
```

C#

```
public class MyFirstStack : Stack
{
    public MyFirstStack(Stack scope, string id, StackProps props = null) :
 base(scope, id, props)
    {
        new Bucket(this, "MyFirstBucket");
    }
}
```

Go

```
func MyFirstStack(scope constructs.Construct, id string, props *MyFirstStackProps)
 awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  s3.NewBucket(stack, jsii.String("MyFirstBucket"), &s3.BucketProps{})
  return stack
}
```

ただし、このコードではスタックのみが宣言されています。スタックを実際に AWS CloudFormation テンプレートに合成してデプロイするには、インスタンス化する必要があります。また、すべての CDK コンストラクトと同様に、何らかのコンテキストでインスタンス化する必要があります。はそのコンテキストAppです。

標準 AWS CDK の開発テンプレートを使用している場合、スタックはAppオブジェクトをインスタンス化するのと同じファイルにインスタンス化されます。

TypeScript

プロジェクトのbinフォルダ内のプロジェクトにちなんで名付けられたファイル (例: `hello-cdk.ts`) 。

JavaScript

プロジェクトのbinフォルダ内のプロジェクトにちなんで名付けられたファイル (例: `hello-cdk.js`) 。

Python

プロジェクトのメインディレクトリapp.pyにある ファイル。

Java

という名前のファイル*ProjectName*App.java、例えば はHelloCdkApp.java、 src/main ディレクトリの下に深くネストされています。

C#

Program.cs という名前のファイル。例えばsrc*ProjectName*、src\HelloCdk \Program.cs。

スタック API

スタックオブジェクトは、以下を含むリッチ API を提供します。

- `Stack.of(construct)` – コンストラクトが定義されている スタックを返す静的メソッド。これは、再利用可能なコンストラクト内からスタックを操作する必要がある場合に便利です。スタックがスコープ内で見つからない場合、呼び出しは失敗します。
- `stack.stackName` (Python: `stack_name`) — スタックの物理名を返します。前述のように、すべての AWS CDK スタックには合成中に が解決 AWS CDK できる物理名があります。
- `stack.region` および `stack.account` - このスタックをデプロイする AWS リージョンとアカウントをそれぞれ返します。これらのプロパティは、次のいずれかを返します。
  - スタックが定義されたときに明示的に指定されたアカウントまたはリージョン
  - アカウントとリージョンの AWS CloudFormation 擬似パラメータに解決され、このスタックが環境に依存しないことを示す文字列エンコードされたトークン

  スタックの環境の決定方法については、「」を参照してくださいthe section called “環境”。
- `stack.addDependency(stack)` (Python: `stack.add_dependency(stack)`) – 2 つのスタック間で依存関係の順序を明示的に定義するために使用できます。この順序は、複数のスタックを一度にデプロイするときに cdk deploy コマンドによって尊重されます。

- `stack.tags` – スタックレベルのタグを追加または削除するためにTagManager使用できる を返します。このタグマネージャーは、スタック内のすべてのリソースにタグを付け、 を通じてスタックを作成するときにスタック自体にタグを付けます AWS CloudFormation。
- `stack.partition`、 `stack.urlSuffix` (Python: `url_suffix`) `stack.stackId`、 (Python: `stack_id`) 、および `stack.notificationArn` (Python: `notification_arn`) – などのそれぞれの AWS CloudFormation 擬似パラメータに解決されるトークンを返します`{ "Ref": "AWS::Partition" }`。これらのトークンは、 AWS CDK フレームワークがクロススタック参照を識別できるように、特定のスタックオブジェクトに関連付けられます。
- `stack.availabilityZones` (Python: `availability_zones`) — このスタックがデプロイされている環境で使用可能なアベイラビリティーゾーンのセットを返します。環境に依存しないスタックの場合、これは常に 2 つのアベイラビリティーゾーンを持つ配列を返します。環境固有のスタックの場合、 は環境を AWS CDK クエリし、指定したリージョンで使用可能なアベイラビリティーゾーンの正確なセットを返します。
- `stack.parseArn(arn)` および `stack.formatArn(comps)` (Python: `parse_arn`、`format_arn`) — Amazon リソースネーム (ARNsの操作に使用できます。
- `stack.toJsonString(obj)` (Python: `to_json_string`) – 任意のオブジェクトを AWS CloudFormation テンプレートに埋め込むことができる JSON 文字列としてフォーマットするために使用できます。オブジェクトには、デプロイ中にのみ解決されるトークン、属性、参照を含めることができます。
- `stack.templateOptions` (Python: `template_options`) — スタックの変換、説明、メタデータなどのテンプレートオプションを指定 AWS CloudFormation するために使用します。

スタックの操作

CDK アプリ内のすべてのスタックを一覧表示するには、 cdk ls コマンドを使用します。前の例では、以下を出力します。

```
stack1
stack2
```

スタックは、 AWS CloudFormation スタックの一部として AWS 環境 にデプロイされます。環境は、特定の AWS アカウント と をカバーします AWS リージョン。

複数のスタックを持つアプリケーションに対して cdk synth コマンドを実行すると、クラウドアセンブリにはスタックインスタンスごとに個別のテンプレートが含まれます。2 つのスタックが同じク

ラスのインスタンスであっても、 はそれらを 2 つの個別のテンプレートとして AWS CDK 出力します。

cdk synth コマンドでスタック名を指定することで、各テンプレートを合成できます。次の例では、Stack1 の テンプレートを合成します。

```
$ cdk synth stack1
```

このアプローチは、 AWS CloudFormation テンプレートを複数回デプロイし、AWS CloudFormation パラメータ を使用してパラメータ化できる、テンプレートの通常の使用方法とは概念的に異なります。 AWS CloudFormation パラメータは で定義できますが AWS CDK、 AWS CloudFormation パラメータはデプロイ時にのみ解決されるため、通常は推奨されません。つまり、コード内の値を決定することはできません。

例えば、パラメータ値に基づいて条件付きでアプリケーションにリソースを含めるには、 AWS CloudFormation 条件を設定し、リソースにタグを付ける必要があります。 AWS CDK は、合成時に具体的なテンプレートが解決されるアプローチを採用しています。したがって、if ステートメントを使用して値をチェックし、リソースを定義するか、何らかの動作を適用するかを決定できます。

> **Note**
>
> AWS CDK は、合成時に可能な限り多くの解像度を提供し、プログラミング言語のイディオマティックで自然な使用を可能にします。

他のコンストラクトと同様に、スタックはグループで構成できます。次のコードは、コントロールプレーン、データプレーン、モニタリングスタックの 3 つのスタックで構成されるサービスの例を示しています。サービスコンストラクトは 2 回定義されます。1 つはベータ環境用、もう 1 つは本番環境用です。

TypeScript

```
import { App, Stack } from 'aws-cdk-lib';
import { Construct } from 'constructs';

interface EnvProps {
  prod: boolean;
}

// imagine these stacks declare a bunch of related resources
```

```
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope: Construct, id: string, props?: EnvProps) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

JavaScript

```
const { App, Stack } = require('aws-cdk-lib');
const { Construct } = require('constructs');

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope, id, props) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");
  }
```

```
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

Python

```
from aws_cdk import App, Stack
from constructs import Construct

# imagine these stacks declare a bunch of related resources
class ControlPlane(Stack): pass
class DataPlane(Stack): pass
class Monitoring(Stack): pass

class MyService(Construct):

  def __init__(self, scope: Construct, id: str, *, prod=False):

    super().__init__(scope, id)

    # we might use the prod argument to change how the service is configured
    ControlPlane(self, "cp")
    DataPlane(self, "data")
    Monitoring(self, "mon")

app = App();
MyService(app, "beta")
MyService(app, "prod", prod=True)

app.synth()
```

Java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.constructs.Construct;
```

```
public class MyApp {

    // imagine these stacks declare a bunch of related resources
    static class ControlPlane extends Stack {
        ControlPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class DataPlane extends Stack {
        DataPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class Monitoring extends Stack {
        Monitoring(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class MyService extends Construct {
        MyService(Construct scope, String id) {
            this(scope, id, false);
        }

        MyService(Construct scope, String id, boolean prod) {
            super(scope, id);

            // we might use the prod argument to change how the service is
 configured
            new ControlPlane(this, "cp");
            new DataPlane(this, "data");
            new Monitoring(this, "mon");
        }
    }

    public static void main(final String argv[]) {
        App app = new App();

        new MyService(app, "beta");
        new MyService(app, "prod", true);

        app.synth();
```

```
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;

// imagine these stacks declare a bunch of related resources
public class ControlPlane : Stack {
    public ControlPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class DataPlane : Stack {
    public DataPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class Monitoring : Stack
{
    public Monitoring(Construct scope, string id=null) : base(scope, id) { }
}

public class MyService : Construct
{
    public MyService(Construct scope, string id, Boolean prod=false) : base(scope,
 id)
    {
        // we might use the prod argument to change how the service is configured
        new ControlPlane(this, "cp");
        new DataPlane(this, "data");
        new Monitoring(this, "mon");
    }
}

class Program
{
    static void Main(string[] args)
    {

        var app = new App();
        new MyService(app, "beta");
        new MyService(app, "prod", prod: true);
        app.Synth();
```

```
    }
}
```

この AWS CDK アプリは最終的に 6 つのスタックで構成され、環境ごとに 3 つになります。

```
$ cdk ls

betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

AWS CloudFormation スタックの物理名は、ツリー内のスタックのコンストラクトパス AWS CDK に基づいて によって自動的に決定されます。デフォルトでは、スタックの名前は `Stack` オブジェクトのコンストラクト ID から取得されます。ただし、次のように prop (Python では `stackName` `stack_name`) を使用して明示的な名前を指定できます。

TypeScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

JavaScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

Python

```
MyStack(self, "not:a:stack:name", stack_name="this-is-stack-name")
```

Java

```
new MyStack(this, "not:a:stack:name", StackProps.builder()
    .StackName("this-is-stack-name").build());
```

C#

```
new MyStack(this, "not:a:stack:name", new StackProps
```

```
{
    StackName = "this-is-stack-name"
});
```

ネストされたスタック

NestedStack コンストラクトは、スタックの AWS CloudFormation 500 リソース制限を回避する方法を提供します。ネストされたスタックは、それを含むスタック内の 1 つのリソースとしてカウントされます。ただし、追加のネストされたスタックを含め、最大 500 個のリソースを含めることができます。

ネストされたスタックの範囲は、 `Stack`または `NestedStack`コンストラクトである必要があります。ネストされたスタックは、親スタック内で辞書的に宣言する必要はありません。ネストされたスタックをインスタンス化するときには、親スタックを最初のパラメータ (`scope`) として渡すだけで済みます。この制限を除いて、ネストされたスタックのコンストラクトの定義は、通常のスタックとまったく同じように機能します。

合成時に、ネストされたスタックは独自の AWS CloudFormation テンプレートに合成され、デプロイ時にステージングバケットにアップロード AWS CDK されます。ネストされたスタックは親スタックにバインドされ、独立したデプロイアーティファクトとして扱われません。これらは によってリストされておらず`cdk list`、 によってデプロイすることはできません`cdk deploy`。

親スタックとネストされたスタック間のリファレンスは、クロススタックリファレンス と同様に、生成された AWS CloudFormation テンプレート内のスタックパラメータと出力に自動的に変換されます。

> **Warning**
>
> セキュリティ体制の変更は、ネストされたスタックのデプロイ前には表示されません。この情報は、最上位スタックに対してのみ表示されます。

構築

コンストラクトはアプリケーションの基本的な構成要素です AWS Cloud Development Kit (AWS CDK) 。コンストラクトは、1 つ以上の AWS CloudFormation リソースとその設定を表すアプリケーション内のコンポーネントです。コンストラクトをインポートして設定することで、アプリケーションを個別に構築します。

コンストラクトは、CDK アプリケーションにインポートするクラスです。コンストラクトは、 AWS コンストラクトライブラリから入手できます。また、独自のコンストラクトを作成して配布したり、サードパーティーのデベロッパーが作成したコンストラクトを使用したりできます。

コンストラクトは、コンストラクトプログラミングモデル (CPM) の一部です。CDK for Terraform (CDKtf)、CDK for (CDK8s)、 などの他のツールで使用できますProjen。 Kubernetes CDK8s

トピック

- AWS ライブラリの構築
- コンストラクトの定義
- コンストラクトの使用
- サードパーティーのコンストラクトの使用
- 詳細はこちら

AWS ライブラリの構築

AWS コンストラクトライブラリには、 によって開発および保守されるコンストラクトのコレクションが含まれています AWS。これは、 で利用可能なすべてのリソースを表すコンストラクトを含むさまざまなモジュールに編成されています AWS。リファレンス情報については、AWS CDK 「 API リファレンス」を参照してください。

メイン CDK パッケージは と呼ばれaws-cdk-lib、 AWS コンストラクトライブラリの大部分が含まれています。また、 Stackや などの基本クラスも含まれていますApp。

メイン CDK パッケージの実際のパッケージ名は言語によって異なります。

TypeScript

インストール	`npm install aws-cdk-lib`
Import	`import * as cdk from 'aws-cdk-lib';`

JavaScript

インストール	`npm install aws-cdk-lib`

Import	`const cdk = require('aws-cdk-lib');`

Python

インストール	python -m pip install aws-cdk-lib
Import	`import aws_cdk as cdk`

Java

でpom.xml、 を追加します。	グループ `software.amazon.awscdk` ; アーティファクト `aws-cdk-lib`
Import	`import software.amazon.awscdk.App;`

C#

インストール	dotnet add package Amazon.CDK.Lib
Import	`using Amazon.CDK;`

Go

インストール	go get github.com/aws/aws-cdk-go/awscdk/v2
Import	`import ( "github.com/aws/aws-cdk-go/awscdk/v2" )`

> **Note**
>
> を使用して CDK プロジェクトを作成した場合はcdk init、 を手動でインストールする必要はありませんaws-cdk-lib。

AWS コンストラクトライブラリには、Constructベースクラスを含む constructsパッケージも含まれています。CDK for Terraform や CDK for Kubernetes など AWS CDK、 に加えて他のコンストラクトベースのツールによって使用されるため、独自のパッケージに含まれています。

多数のサードパーティーが と互換性のあるコンストラクトも公開しています AWS CDK。Construct Hub にアクセスして、 AWS CDK コンストラクトパートナーエコシステムをご覧ください。

ビルドレベル

コンストラクトライブラリからのコンストラクトは 3 AWS つのレベルに分類されます。各レベルは、抽象化のレベルを高めます。抽象化が高いほど、設定が容易になり、専門知識が少なくなります。抽象化が低いほど、より多くのカスタマイズが可能となり、より多くの専門知識が必要になります。

レベル 1 (L1) コンストラクト

L1 コンストラクトは、CFN リソース とも呼ばれ、最も低いレベルのコンストラクトであり、抽象化を提供しません。各 L1 コンストラクトは 1 つの AWS CloudFormation リソースに直接マッピングされます。L1 コンストラクトでは、特定の AWS CloudFormation リソースを表すコンストラクトをインポートします。次に、コンストラクトインスタンス内でリソースのプロパティを定義します。

L1 コンストラクトは、 AWS リソースプロパティの定義に慣れており AWS CloudFormation 、完全に制御する必要がある場合に最適です。

AWS コンストラクトライブラリでは、L1 コンストラクトの名前は で始まりCfn、その後にそれが表す AWS CloudFormation リソースの識別子が続きます。例えば、 CfnBucketコンストラクトは AWS::S3::Bucket AWS CloudFormation リソースを表す L1 コンストラクトです。

L1 コンストラクトはAWS CloudFormation リソース仕様 から生成されます。リソースが に存在する場合 AWS CloudFormation、L1 コンストラクト AWS CDK として で使用できます。新しいリソースまたはプロパティがコンストラクトライブラリで利用可能になるまでに最大 1 AWS 週間かかる場合があります。詳細については、「 ユーザーガイド」の「 AWS リソースタイプとプロパティタイプのリファレンスAWS CloudFormation 」を参照してください。

レベル 2 (L2) コンストラクト

キュレーションされたコンストラクトとも呼ばれる L2 コンストラクトは CDK チームによって慎重に開発され、通常は最も広く使用されているコンストラクトタイプです。L2 コンストラクトは、L1 コンストラクトと同様に、単一の AWS CloudFormation リソースに直接マッピングされます。L1 コンストラクトと比較して、L2 コンストラクトは直感的なインテントベースの API を通じてより高度な抽象化を提供します。L2 コンストラクトには、適切なデフォルトプロパティ設定、ベストプラクティスのセキュリティポリシー、および多くの定型コードとグルーロジックの生成が含まれます。

L2 コンストラクトは、ほとんどのリソースのヘルパーメソッドも提供しており、プロパティ、アクセス許可、リソース間のイベントベースのインタラクションなどをより簡単かつ迅速に定義できます。

`s3.Bucket` クラスは、Amazon Simple Storage Service (Amazon S3) バケットリソースの L2 コンストラクトの例です。Amazon S3

AWS コンストラクトライブラリには、安定しており、本番環境で使用できるように指定された L2 コンストラクトが含まれています。開発中の L2 コンストラクトの場合、実験的なものとして指定され、別のモジュールで提供されます。

レベル 3 (L3) コンストラクト

パターン とも呼ばれる L3 コンストラクトは、抽象化の最高レベルです。 各 L3 コンストラクトには、アプリケーション内の特定のタスクまたはサービスを達成するために連携するように設定されたリソースのコレクションを含めることができます。L3 コンストラクトは、アプリケーションの特定のユースケースの AWS アーキテクチャ全体を作成するために使用されます。

完全なシステム設計、または大規模なシステムのかなりの部分を提供するために、L3 コンストラクトはオリジン化されたデフォルトのプロパティ設定を提供します。これらは、問題の解決とソリューションの提供に対する特定のアプローチに基づいて構築されています。L3 コンストラクトを使用すると、入力とコードの量を最小限に抑えながら、複数のリソースをすばやく作成して設定できます。

`ecsPatterns.ApplicationLoadBalancedFargateService` クラスは、Amazon Elastic Container Service (Amazon ECS) クラスターで実行され、Application Load Balancer によってフロントされる AWS Fargate サービスを表す L3 コンストラクトの例です。

L2 コンストラクトと同様に、本番環境で使用できる L3 AWS コンストラクトは、 コンストラクトライブラリに含まれています。開発中のものは別々のモジュールで提供されます。

コンストラクトの定義

コンポジション

コンポジションは、コンストラクトを通じて高レベルの抽象化を定義するための主要なパターンです。高レベルのコンストラクトは、任意の数の下位レベルのコンストラクトから構成できます。ボトムアップの観点からは、 コンストラクトを使用して、デプロイする個々の AWS リソースを整理します。目的に便利な任意の抽象化を必要な数だけ使用します。

コンポジションでは、再利用可能なコンポーネントを定義し、他のコードと同様に共有します。例えば、チームは、バックアップ、グローバルレプリケーション、自動スケーリング、モニタリングなど、Amazon DynamoDB テーブルに対する会社のベストプラクティスを実装するコンストラクトを定義できます。チームは、コンストラクトを内部で他のチームと共有することも、パブリックに共有することもできます。

チームは、他のライブラリパッケージと同様にコンストラクトを使用できます。ライブラリが更新されると、デベロッパーは他のコードライブラリと同様に、新しいバージョンの改善とバグ修正にアクセスできます。

初期化

コンストラクトは、`Construct` 基本クラスを拡張するクラスで実装されます。コンストラクトを定義するには、 クラスをインスタンス化します。すべてのコンストラクトは、初期化時に次の 3 つのパラメータを取ります。

- scope – コンストラクトの親または所有者。これはスタックでも別のコンストラクトでもかまいません。スコープは、コンストラクトツリー 内のコンストラクトの位置を決定します。通常、スコープの現在のオブジェクトを表す `this` (`self` の Python) を渡す必要があります。
- id – スコープ内で一意???である必要があります。識別子は、 コンストラクト内で定義されているすべての名前空間として機能します。リソース名や AWS CloudFormation 論理 IDs などの一意の識別子を生成するために使用されます。

  識別子はスコープ内でのみ一意である必要があります。これにより、含まれる可能性のあるコンストラクトと識別子を気にせずにコンストラクトをインスタンス化して再利用でき、コンストラクトをより高いレベルの抽象化に構成できます。さらに、スコープを使用すると、コンストラクトのグループを一度に参照できます。例としては、 のタグ付けや、コンストラクトをデプロイする場所の指定などがあります。

- props – 言語に応じて、コンストラクトの初期設定を定義するプロパティまたはキーワード引数のセット。高レベルのコンストラクトはより多くのデフォルトを提供し、すべての prop 要素がオプションの場合、props パラメータを完全に省略できます。

構成

ほとんどのコンストラクトは、3 `props`番目の引数 (または Python ではキーワード引数) として、コンストラクトの設定を定義する名前/値コレクションを受け入れます。次の例では、 AWS Key Management Service （ AWS KMS) 暗号化と静的ウェブサイトホスティングが有効になっているバケットを定義します。暗号化キーは明示的に指定されないため、 `Bucket`コンストラクトは新しいを定義`kms.Key`し、バケットに関連付けます。

TypeScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

JavaScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

Python

```
s3.Bucket(self, "MyEncryptedBucket", encryption=s3.BucketEncryption.KMS,
    website_index_document="index.html")
```

Java

```
Bucket.Builder.create(this, "MyEncryptedBucket")
        .encryption(BucketEncryption.KMS_MANAGED)
        .websiteIndexDocument("index.html").build();
```

C#

```
new Bucket(this, "MyEncryptedBucket", new BucketProps
```

```
{
    Encryption = BucketEncryption.KMS_MANAGED,
    WebsiteIndexDocument = "index.html"
});
```

Go

```
awss3.NewBucket(stack, jsii.String("MyEncryptedBucket"), &awss3.BucketProps{
 Encryption: awss3.BucketEncryption_KMS,
 WebsiteIndexDocument: jsii.String("index.html"),
})
```

コンストラクトの操作

コンストラクトは、基本コンストラクトクラスを拡張するクラスです。コンストラクトをインスタンス化すると、コンストラクトオブジェクトは一連のメソッドとプロパティを公開します。これにより、コンストラクトを操作し、システムの他の部分への参照として渡すことができます。

AWS CDK フレームワークは、コンストラクトの APIsに制限を設けません。作成者は任意の API を定義できます。ただし、 などの AWS コンストラクトライブラリに含まれているコンストラクト AWS は`s3.Bucket`、ガイドラインと一般的なパターンに従います。これにより、すべての AWS リソースで一貫したエクスペリエンスが得られます。

ほとんどの AWS コンストラクトには、そのコンストラクトに対する AWS Identity and Access Management (IAM) アクセス許可をプリンシパルに付与するために使用できる一連の付与メソッドがあります。次の例では、Amazon S3 バケット から読み取るアクセス`data-science`許可を IAM グループに付与します`raw-data`。 Amazon S3

TypeScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

JavaScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

Python

```
raw_data = s3.Bucket(self, 'raw-data')
data_science = iam.Group(self, 'data-science')
raw_data.grant_read(data_science)
```

Java

```
Bucket rawData = new Bucket(this, "raw-data");
Group dataScience = new Group(this, "data-science");
rawData.grantRead(dataScience);
```

C#

```
var rawData = new Bucket(this, "raw-data");
var dataScience = new Group(this, "data-science");
rawData.GrantRead(dataScience);
```

Go

```
rawData := awss3.NewBucket(stack, jsii.String("raw-data"), nil)
dataScience := awsiam.NewGroup(stack, jsii.String("data-science"), nil)
rawData.GrantRead(dataScience, nil)
```

もう 1 つの一般的なパターンは、 AWS コンストラクトが他の場所で提供されるデータからリソースの属性の 1 つを設定することです。属性には、Amazon リソースネーム (ARNs名前、または URLs。

次のコードは、 AWS Lambda 関数を定義し、環境変数のキューの URL を介して Amazon Simple Queue Service (Amazon SQS) キューに関連付けます。

TypeScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
```

```
});
```

JavaScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

Python

```
jobs_queue = sqs.Queue(self, "jobs")
create_job_lambda = lambda_.Function(self, "create-job",
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    code=lambda_.Code.from_asset("./create-job-lambda-code"),
    environment=dict(
        QUEUE_URL=jobs_queue.queue_url
    )
)
```

Java

```
final Queue jobsQueue = new Queue(this, "jobs");
Function createJobLambda = Function.Builder.create(this, "create-job")
                .handler("index.handler")
                .code(Code.fromAsset("./create-job-lambda-code"))
                .environment(java.util.Map.of(   // Map.of is Java 9 or later
                    "QUEUE_URL", jobsQueue.getQueueUrl())
                .build();
```

C#

```
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
{
    Runtime = Runtime.NODEJS_18_X,
```

```
    Handler = "index.handler",
    Code = Code.FromAsset(@".\create-job-lambda-code"),
    Environment = new Dictionary<string, string>
    {
        ["QUEUE_URL"] = jobsQueue.QueueUrl
    }
});
```

Go

```
createJobLambda := awslambda.NewFunction(stack, jsii.String("create-job"),
&awslambda.FunctionProps{
 Runtime: awslambda.Runtime_NODEJS_18_X(),
 Handler: jsii.String("index.handler"),
 Code:    awslambda.Code_FromAsset(jsii.String(".\\create-job-lambda-code"), nil),
 Environment: &map[string]*string{
  "QUEUE_URL": jsii.String(*jobsQueue.QueueUrl()),
 },
})
```

コンストラクトライブラリの最も一般的な API AWS パターンについては、「」を参照してくださいthe section called “リソース”。

アプリケーションとスタックのコンストラクト

コンストラクトライブラリの AWS `App`および `Stack` クラスは一意のコンストラクトです。他のコンストラクトと比較して、 AWS リソースは独自に設定されません。代わりに、他のコンストラクトのコンテキストを提供するために使用されます。リソースを表す AWS すべてのコンストラクトは、`Stack`コンストラクトの範囲内で直接または間接的に定義する必要があります。 `Stack` コンストラクトは、`App`コンストラクトの範囲内で定義されます。

CDK アプリケーションの詳細については、「」を参照してくださいAWS CDK アプリ。CDK スタックの詳細については、「」を参照してくださいスタック。

次の例では、単一のスタックを持つアプリケーションを定義します。スタック内では、L2 コンストラクトを使用して Amazon S3 バケットリソースを設定します。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';
```

```
class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

Python

```
from aws_cdk import App, Stack
import aws_cdk.aws_s3 as s3
from constructs import Construct

class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
```

```
        s3.Bucket(self, "MyFirstBucket", versioned=True)

app = App()
HelloCdkStack(app, "HelloCdkStack")
```

Java

HelloCdkStack.java ファイルで定義されるスタック：

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

HelloCdkApp.java ファイルで定義されるアプリケーション：

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.StackProps;

public class HelloCdkApp {
    public static void main(final String[] args) {
        App app = new App();

        new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdkApp
{
    internal static class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new HelloCdkStack(app, "HelloCdkStack");
            app.Synth();
        }
    }

    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps { Versioned = true });
        }
    }
}
```

Go

```
func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}
```

コンストラクトの使用

L1 コンストラクトの使用

L1 コンストラクトは個々の AWS CloudFormation リソースに直接マッピングされます。リソースに必要な設定を指定する必要があります。

この例では、L1 `CfnBucket` コンストラクトを使用して`bucket`オブジェクトを作成します。

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket"
});
```

Python

```
bucket = s3.CfnBucket(self, "MyBucket", bucket_name="MyBucket")
```

Java

```
CfnBucket bucket = new CfnBucket.Builder().bucketName("MyBucket").build();
```

C#

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName= "MyBucket"
});
```

Go

```
 awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
```

```
  BucketName: jsii.String("MyBucket"),
 })
```

単純なブール値、文字列、数値、またはコンテナではないコンストラクトプロパティは、サポートされている言語で異なる方法で処理されます。

TypeScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "MyBucket", {
  bucketName: "MyBucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

Python

Python では、これらのプロパティは L1 コンストラクトの内部クラスとして定義された型で表されます。例えば、 のオプションプロパティ`cors_configuration`には、タイプ のラッパー`CfnBucket`が必要です`CfnBucket.CorsConfigurationProperty`。ここでは、`CfnBucket`インスタンス`cors_configuration`で を定義します。

```
bucket = CfnBucket(self, "MyBucket", bucket_name="MyBucket",
    cors_configuration=CfnBucket.CorsConfigurationProperty(
        cors_rules=[CfnBucket.CorsRuleProperty(
            allowed_origins=["*"],
```

```
            allowed_methods=["GET"]
        )]
    )
)
```

Java

Java では、これらのプロパティは L1 コンストラクトの内部クラスとして定義された型で表されます。例えば、 のオプションプロパティcorsConfigurationには、タイプ のラッパーCfnBucketが必要ですCfnBucket.CorsConfigurationProperty。ここでは、CfnBucketインスタンスcorsConfigurationで を定義します。

```
CfnBucket bucket = CfnBucket.Builder.create(this, "MyBucket")
                        .bucketName("MyBucket")
                        .corsConfiguration(new
 CfnBucket.CorsConfigurationProperty.Builder()
                            .corsRules(Arrays.asList(new
 CfnBucket.CorsRuleProperty.Builder()
                                .allowedOrigins(Arrays.asList("*"))
                                .allowedMethods(Arrays.asList("GET"))
                                .build()))
                            .build())
                        .build();
```

C#

C# では、これらのプロパティは L1 コンストラクトの内部クラスとして定義された型で表されます。例えば、 のオプションプロパティCorsConfigurationには、タイプ のラッパーCfnBucketが必要ですCfnBucket.CorsConfigurationProperty。ここでは、CfnBucketインスタンスCorsConfigurationで を定義します。

```
var bucket = new CfnBucket(this, "MyBucket", new CfnBucketProps
{
    BucketName = "MyBucket",
    CorsConfiguration = new CfnBucket.CorsConfigurationProperty
    {
        CorsRules = new object[] {
            new CfnBucket.CorsRuleProperty
            {
                AllowedOrigins = new string[] { "*" },
                AllowedMethods = new string[] { "GET" },
            }
```

```
        }
    }
});
```

Go

Go では、これらのタイプは L1 コンストラクトの名前、アンダースコア、プロパティ名を使用して名前が付けられます。例えば、 のオプションプロパティCorsConfigurationには、タイプ のラッパーCfnBucketが必要ですCfnBucket_CorsConfigurationProperty。ここでは、CfnBucketインスタンスCorsConfigurationで を定義します。

```
awss3.NewCfnBucket(stack, jsii.String("MyBucket"), &awss3.CfnBucketProps{
 BucketName: jsii.String("MyBucket"),
 CorsConfiguration: &awss3.CfnBucket_CorsConfigurationProperty{
  CorsRules: []awss3.CorsRule{
   awss3.CorsRule{
    AllowedOrigins: jsii.Strings("*"),
    AllowedMethods: &[]awss3.HttpMethods{"GET"},
   },
  },
 },
})
```

Important

L1 コンストラクトで L2 プロパティタイプを使用することはできません。その逆も同様です。 L1 L1 コンストラクトを使用する場合は、使用している L1 コンストラクトに定義されている型を常に使用してください。他の L1 コンストラクトの型は使用しないでください (名前は同じでも、タイプは同じではないものもあります) 。
言語固有の API リファレンスの中には、現在 L1 プロパティタイプへのパスにエラーがあるか、これらのクラスをまったく文書化していないものがあります。これについては、近日中に修正する予定です。その間、このようなタイプは常に、使用される L1 コンストラクトの内部クラスであることに注意してください。

L2 コンストラクトの使用

次の例では、L2 コンストラクトからオブジェクトを作成して Amazon S3 Bucket バケットを定義します。

TypeScript

```
import * as s3 from 'aws-cdk-lib/aws-s3';

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

JavaScript

```
const s3 = require('aws-cdk-lib/aws-s3');

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

Python

```
import aws_cdk.aws_s3 as s3

# "self" is HelloCdkStack
s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

```
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
                .versioned(true).build();
    }
}
```

C#

```
using Amazon.CDK.AWS.S3;

// "this" is HelloCdkStack
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true
});
```

Go

```
import (
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/jsii-runtime-go"
)

// stack is HelloCdkStack
awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })>
```

`MyFirstBucket` は、 が AWS CloudFormation 作成するバケットの名前ではありません。これは、CDK アプリケーションのコンテキスト内の新しいコンストラクトに与えられる論理識別子です。physicalName 値を使用して AWS CloudFormation リソースに名前を付けます。

サードパーティーのコンストラクトの使用

Construct Hub は、、サードパーティー AWS、オープンソース CDK コミュニティから追加のコンストラクトを発見するのに役立つリソースです。

独自のコンストラクトの作成

既存のコンストラクトを使用することに加えて、独自のコンストラクトを記述し、誰でもアプリで使用できるようにすることもできます。すべてのコンストラクトは で等しくなります AWS CDK。コンストラクトライブラリからの AWS コンストラクトは、、NPM、Mavenまたは を介して公開されたサードパーティーライブラリからのコンストラクトと同じように扱われますPyPI。会社の内部パッケージリポジトリに公開されたコンストラクトも同じように扱われます。

新しいコンストラクトを宣言するには、コンストラクトベースクラスを拡張するクラスを `constructs` パッケージに作成し、初期化引数のパターンに従います。

次の例は、Amazon S3 バケットを表すコンストラクトを宣言する方法を示しています。S3 バケットは、誰かがファイルをアップロードするたびに Amazon Simple Notification Service (Amazon SNS) 通知を送信します。

TypeScript

```
export interface NotifyingBucketProps {
  prefix?: string;
}

export class NotifyingBucket extends Construct {
  constructor(scope: Construct, id: string, props: NotifyingBucketProps = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {
  constructor(scope, id, props = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket }
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None):
```

```
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        Topic topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucketProps : BucketProps
{
    public string Prefix { get; set; }
}
```

```
public class NotifyingBucket : Construct
{
    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        var topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

```
type NotifyingBucketProps struct {
 awss3.BucketProps
 Prefix *string
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) awss3.Bucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 return bucket
}
```

> **Note**
>
> NotifyingBucket コンストラクトは からではなくBucket、 から継承しますConstruct。Amazon S3 バケットと Amazon SNS トピックをバンドルするには、継承ではなくコンポジションを使用しています。 Amazon SNS 一般に、 AWS CDK コンストラクトを開発する場合、コンポジションは継承よりも優先されます。

NotifyingBucket コンストラクタには、、scope、 という一般的なコンストラクト署名がありますidprops。最後の引数 propsはオプション (デフォルト値 を取得{}) です。すべての props はオプションであるためです。(基本Constructクラスは引props数を取りません)。このコンストラクトのインスタンスは、 なしでアプリで定義できます。例えばprops、次のとおりです。

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), nil)
```

または、 `props` (Java では追加のパラメータ) を使用して、フィルタリングするパスプレフィックスを指定できます。例えば、次のようになります。

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket", prefix="images/")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket", "/images");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), &NotifyingBucketProps{
 Prefix: jsii.String("images/"),
})
```

通常、コンストラクトでいくつかのプロパティやメソッドを公開することもできます。コンストラクトのユーザーはサブスクライブできないため、コンストラクトの背後にトピックを非表示にすることはあまり有用ではありません。`topic` プロパティを追加すると、次の例に示すように、コンシューマーは内部トピックにアクセスできます。

TypeScript

```
export class NotifyingBucket extends Construct {
  public readonly topic: sns.Topic;

  constructor(scope: Construct, id: string, props: NotifyingBucketProps) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {

  constructor(scope, id, props) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket };
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None, **kwargs):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        self.topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(self.topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public Topic topic = null;

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucket : Construct
{
    public readonly Topic topic;

    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
```

```
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

Go でこれを行うには、少し余分なパイプが必要です。元のNewNotifyingBucket関数が を返しましたawss3.Bucket。NotifyingBucket 構造体を作成してtopic、メンバーを含めるBucketように拡張する必要があります。その後、関数はこのタイプを返します。

```
type NotifyingBucket struct {
 awss3.Bucket
 topic awssns.Topic
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) NotifyingBucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
 }
 var nbucket NotifyingBucket
 nbucket.Bucket = bucket
 nbucket.topic = topic
 return nbucket
}
```

これで、コンシューマーはトピックをサブスクライブできます。例えば、次のようになります。

TypeScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

JavaScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

Python

```
queue = sqs.Queue(self, "NewImagesQueue")
images = NotifyingBucket(self, prefix="Images")
images.topic.add_subscription(sns_sub.SqsSubscription(queue))
```

Java

```
NotifyingBucket images = new NotifyingBucket(this, "MyNotifyingBucket", "/images");
images.topic.addSubscription(new SqsSubscription(queue));
```

C#

```
var queue = new Queue(this, "NewImagesQueue");
var images = new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
images.topic.AddSubscription(new SqsSubscription(queue));
```

Go

```
 queue := awssqs.NewQueue(stack, jsii.String("NewImagesQueue"), nil)
 images := NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"),
 &NotifyingBucketProps{
  Prefix: jsii.String("/images"),
 })
```

```
images.topic.AddSubscription(awssnssubscriptions.NewSqsSubscription(queue, nil))
```

詳細はこちら

次の動画では、CDK コンストラクトの包括的な概要と、CDK アプリケーションでのそれらの使用方法について説明します。

CDK コンストラクトの説明

環境

環境は、 AWS Cloud Development Kit (AWS CDK) スタックをデプロイ AWS リージョン する AWS アカウント と で構成されます。

AWS アカウント

を作成すると AWS アカウント、 アカウント ID を受け取ります。この ID は、 など、アカウントを一意に識別012345678901する 12 桁の数字です。詳細については、「 AWS Account Management リファレンスガイド」の「識別子の表示 AWS アカウント」を参照してください。

AWS リージョン

AWS リージョン は、地理的位置とリージョンのアベイラビリティーゾーンを表す番号を組み合わせて名前を付けます。例えば、 は米国東部 (バージニア北部) リージョンのアベイラビリティーゾーンus-east-1を表します。の詳細については AWS リージョン、「リージョンとアベイラビリティーゾーン」を参照してください。リージョンコードのリストについては、「 AWS 全般リファレンスガイド」の「リージョンエンドポイント」を参照してください。

AWS CDK は、認証情報と設定ファイルから環境を判断できます。これらのファイルは、 AWS Command Line Interface () を使用して作成および管理できますAWS CLI。これらのファイルの基本的な例を次に示します。

認証情報ファイル

```
[default]
aws_access_key_id=ASIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
aws_session_token =
 IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE
```

```
[user1]
aws_access_key_id=ASIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
aws_session_token =
 fcZib3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE
```

設定ファイル

```
[default]
region=us-west-2
output=json

[profile user1]
region=us-east-1
output=text
```

CDK コード内のこれらのファイルから、CDK によって提供される環境変数を介して環境情報を渡すことができます。などの CDK CLI コマンドを実行するときはcdk deploy、認証情報と設定ファイルからプロファイルを指定して環境情報を収集します。

CDK コードでこれらの環境変数を指定する例を次に示します。

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

以下は、 --profileオプションCLIを使用して、user1プロファイルに関連付けられた値を認証情報と設定ファイルから CDK に渡す例です。これらのファイルの値は、環境変数に渡されます。

```
$ cdk deploy myStack --profile user1
```

認証情報と設定ファイルの値を使用する代わりに、CDK コードで環境値をハードコーディングすることもできます。以下に例を示します。

```
const envEU = { account: '238383838383', region: 'eu-west-1' };
const envUSA = { account: '837873873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
```

```
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

詳細はこちら

で環境の使用を開始するには AWS CDK、「」を参照してくださいで使用する環境を設定する AWS CDK。

ブートストラッピング

ブートストラップは、 で使用する AWS 環境を準備するプロセスです AWS Cloud Development Kit (AWS CDK)。CDK スタックを AWS 環境にデプロイする前に、まず環境をブートストラップする必要があります。

トピック

- ブートストラップとは
- ブートストラップの仕組み
- 詳細はこちら

ブートストラップとは

ブートストラップは、 が使用する AWS 環境内の特定の AWS リソースをプロビジョニングすることで、環境を準備します AWS CDK。これには以下が含まれます。

- Amazon Simple Storage Service (Amazon S3) バケット – AWS Lambda 関数コードやアセットなどの CDK プロジェクトファイルを保存するために使用されます。
- Amazon Elastic Container Registry (Amazon ECR) リポジトリ – 主にDockerイメージの保存に使用されます。
- AWS Identity and Access Management (IAM) ロール – デプロイを実行するために が必要とするアクセス許可を付与 AWS CDK するように設定されています。

ブートストラップの仕組み

CDK で使用されるリソースとその設定は、 AWS CloudFormation テンプレートで定義されます。このテンプレートは CDK チームによって作成および管理されます。このテンプレートの最新バージョンについては、aws-cdk リポジトリ `bootstrap-template.yaml`の「」を参照してください。GitHub

環境をブートストラップするには、 AWS CDK コマンドラインインターフェイス (AWS CDK CLI) `cdk bootstrap` コマンドを使用します。CDK はテンプレートCLIを取得し、ブートストラップスタック と呼ばれるスタック AWS CloudFormation として にデプロイします。 デフォルトでは、スタック名は ですCDKToolkit。このテンプレートをデプロイすることで、 は 環境にリソースを CloudFormation プロビジョニングします。デプロイ後、ブートストラップスタックが環境の AWS CloudFormation コンソールに表示されます。

テンプレートを変更するか、 `cdk bootstrap` コマンドで CDK CLIオプションを使用して、ブートストラップをカスタマイズすることもできます。

AWS 環境は独立しています。で使用する各環境は、まずブートストラップ AWS CDK する必要があります。

詳細はこちら

環境のブートストラップの手順については、「」を参照してください[で使用する環境をブートストラップする AWS CDK](#)。

リソース

リソースは、アプリケーションで使用するために設定 AWS のサービス したものです。リソースは の機能です AWS CloudFormation。 AWS CloudFormation テンプレートでリソースとそのプロパティを設定することで、 にデプロイしてリソース AWS CloudFormation をプロビジョニングできます。では AWS Cloud Development Kit (AWS CDK)、 コンストラクトを使用してリソースを設定できます。次に、CDK アプリケーションをデプロイします。これには、 AWS CloudFormation テンプレートを合成し、 にデプロイ AWS CloudFormation してリソースをプロビジョニングします。

トピック

- [コンストラクトを使用したリソースの設定](#)
- [リソースの参照](#)
- [リソースの物理名](#)
- [一意のリソース識別子を渡す](#)
- [リソース間のアクセス許可の付与](#)
- [リソースメトリクスとアラーム](#)
- [ネットワークトラフィック](#)
- [イベント処理](#)

- 削除ポリシー

コンストラクトを使用したリソースの設定

で説明されているようにthe section called “構築”、 AWS CDK は、すべての AWS リソースを表すコンストラクト AWS と呼ばれるコンストラクトの豊富なクラスライブラリを提供します。

対応するコンストラクトを使用してリソースのインスタンスを作成するには、スコープを最初の引数、コンストラクトの論理 ID、および一連の設定プロパティ (props) として渡します。例えば、Construct Library の sqs.Queue コンストラクトを使用して AWS KMS 暗号化された Amazon SQS AWS キューを作成する方法は次のとおりです。

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

Python

```
import aws_cdk.aws_sqs as sqs

sqs.Queue(self, "MyQueue", encryption=sqs.QueueEncryption.KMS_MANAGED)
```

Java

```
import software.amazon.awscdk.services.sqs.*;

Queue.Builder.create(this, "MyQueue").encryption(
        QueueEncryption.KMS_MANAGED).build();
```

C#

```
using Amazon.CDK.AWS.SQS;

new Queue(this, "MyQueue", new QueueProps
{
    Encryption = QueueEncryption.KMS_MANAGED
});
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

sqs.NewQueue(stack, jsii.String("MyQueue"), &sqs.QueueProps{
  Encryption: sqs.QueueEncryption_KMS_MANAGED,
})
```

一部の設定プロパティはオプションであり、多くの場合、デフォルト値があります。場合によっては、すべてのprops がオプションであり、最後の引数を完全に省略できます。

リソース属性

AWS コンストラクトライブラリのほとんどのリソースは属性を公開し、デプロイ時に によって解決されます AWS CloudFormation。属性は、タイプ名をプレフィックスとして、リソースクラスのプロパティの形式で公開されます。次の例は、 `queueUrl` (Python: `queue_url`) プロパティを使用して Amazon SQS キューの URL を取得する方法を示しています。

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');
```

```
const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

Python

```
import aws_cdk.aws_sqs as sqs

queue = sqs.Queue(self, "MyQueue")
url = queue.queue_url # => A string representing a deploy-time value
```

Java

```
Queue queue = new Queue(this, "MyQueue");
String url = queue.getQueueUrl();    // => A string representing a deploy-time value
```

C#

```
var queue = new Queue(this, "MyQueue");
var url = queue.QueueUrl; // => A string representing a deploy-time value
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

queue := sqs.NewQueue(stack, jsii.String("MyQueue"), &sqs.QueueProps{})
url := queue.QueueUrl() // => A string representing a deploy-time value
```

がデプロイ時の属性を文字列として AWS CDK エンコードする方法については、the section called “トークン”「」を参照してください。

リソースの参照

リソースを設定するときは、多くの場合、別のリソースのプロパティを参照する必要があります。次に例を示します。

- Amazon Elastic Container Service (Amazon ECS) リソースでは、そのリソースが実行されているクラスターへの参照が必要です。
- Amazon CloudFront ディストリビューションでは、ソースコードを含む Amazon Simple Storage Service (Amazon S3) バケットへの参照が必要です。

リソースは、次のいずれかの方法で参照できます。

- CDK アプリケーションで定義されたリソースを、同じスタックまたは別のスタックで渡す
- AWS アカウントで定義されたリソースを参照するプロキシオブジェクトを渡すことで、リソースの一意の識別子 (ARN など) から作成されます。

コンストラクトの プロパティが別のリソースのコンストラクトを表す場合、そのタイプはコンストラクトのインターフェイスタイプのものです。例えば、Amazon ECS コンストラクトは `cluster`型のプロパティを受け取りますecs.ICluster。もう 1 つの例は、 型のプロパティ `sourceBucket` (Python: `source_bucket`) を取得する CloudFront ディストリビューションコンストラクトですs3.IBucket。

同じ AWS CDK アプリで定義されている適切なタイプのリソースオブジェクトを直接渡すことができます。次の例では、Amazon ECS クラスターを定義し、それを使用して Amazon ECS サービスを定義します。

TypeScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

JavaScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

Python

```
cluster = ecs.Cluster(self, "Cluster")

service = ecs.Ec2Service(self, "Service", cluster=cluster)
```

Java

```
Cluster cluster = new Cluster(this, "Cluster");
Ec2Service service = new Ec2Service(this, "Service",
        new Ec2ServiceProps.Builder().cluster(cluster).build());
```

C#

```
var cluster = new Cluster(this, "Cluster");
var service = new Ec2Service(this, "Service", new Ec2ServiceProps { Cluster =
 cluster });
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  ecs "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

cluster := ecs.NewCluster(stack, jsii.String("MyCluster"), &ecs.ClusterProps{})
service := ecs.NewEc2Service(stack, jsii.String("MyService"), &ecs.Ec2ServiceProps{
  Cluster: cluster,
})
```

別のスタック内のリソースの参照

同じアプリケーションで定義され、同じ AWS 環境にある限り、別のスタックのリソースを参照できます。次のパターンが一般的に使用されます。

- コンストラクトへの参照を、リソースを生成するスタックの属性として保存します。(現在のコンストラクトのスタックへの参照を取得するには、 を使用しますStack.of(this)。)
- このリファレンスを、リソースをパラメータまたはプロパティとして消費するスタックのコンストラクタに渡します。次に、消費スタックはそれをプロパティとして、それを必要とする任意のコンストラクトに渡します。

次の例では、スタック を定義しますstack1。このスタックは Amazon S3 バケットを定義し、バケットコンストラクトへの参照をスタックの属性として保存します。次にstack2、アプリケーショ

ンはインスタンス化時にバケットを受け入れる 2 番目のスタック を定義します。 は、例えば、データストレージにバケットを使用する AWS Glue テーブルを定義stack2できます。

TypeScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

JavaScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

Python

```
prod = core.Environment(account="123456789012", region="us-east-1")

stack1 = StackThatProvidesABucket(app, "Stack1", env=prod)

# stack2 will take a property "bucket"
stack2 = StackThatExpectsABucket(app, "Stack2", bucket=stack1.bucket, env=prod)
```

Java

```
// Helper method to build an environment
static Environment makeEnv(String account, String region) {
    return Environment.builder().account(account).region(region)
            .build();
```

```
}

App app = new App();

Environment prod = makeEnv("123456789012", "us-east-1");

StackThatProvidesABucket stack1 = new StackThatProvidesABucket(app, "Stack1",
        StackProps.builder().env(prod).build());

// stack2 will take an argument "bucket"
StackThatExpectsABucket stack2 = new StackThatExpectsABucket(app, "Stack,",
        StackProps.builder().env(prod).build(), stack1.bucket);
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment { Account = account, Region = region };
}

var prod = makeEnv(account: "123456789012", region: "us-east-1");

var stack1 = new StackThatProvidesABucket(app, "Stack1", new StackProps { Env =
 prod });

// stack2 will take a property "bucket"
var stack2 = new StackThatExpectsABucket(app, "Stack2", new StackProps { Env = prod,
    bucket = stack1.Bucket});
```

は、リソースが同じ環境にあり、別のスタックにある AWS CDK と判断した場合、プロデューサースタックの AWS CloudFormation エクスポートとコンシューマスタックの Fn::ImportValue を自動的に合成して、その情報を 1 つのスタックから別のスタックに転送します。

依存関係のデッドロックの解決

別のスタック内の 1 つのスタックからリソースを参照すると、2 つのスタック間に依存関係が作成されます。これにより、正しい順序でデプロイされます。スタックがデプロイされると、この依存関係は具体的になります。その後、共有リソースを消費スタックから削除すると、予期しないデプロイの失敗が発生する可能性があります。これは、2 つのスタック間に同じ順序で強制的にデプロイされる別の依存関係がある場合に発生します。また、プロデューサースタックが CDK Toolkit によって最初にデプロイされるように選択された場合も、依存関係なしで発生する可能性があります。 AWS

CloudFormation エクスポートは、不要になったため生成スタックから削除されますが、更新がまだデプロイされていないため、エクスポートされたリソースは消費スタックでまだ使用されています。したがって、プロデューサースタックのデプロイは失敗します。

このデッドロックを解除するには、共有リソースの使用をコンシューマースタックから削除します。(これにより、プロデューサースタックから自動エクスポートが削除されます。) 次に、自動生成されたエクスポートとまったく同じ論理 ID を使用して、プロデューサースタックに同じエクスポートを手動で追加します。コンシューマースタックの共有リソースの使用を削除し、両方のスタックをデプロイします。次に、手動エクスポート (および不要になった共有リソース) を削除し、両方のスタックを再度デプロイします。スタックの `exportValue()`メソッドは、この目的のために手動エクスポートを作成する便利な方法です。(リンクされたメソッドリファレンスの例を参照してください。)

AWS アカウント内のリソースの参照

AWS CDK アプリの AWS アカウントで既に利用可能なリソースを使用するとします。これは、コンソール、 AWS SDK、 で直接、 AWS CloudFormationまたは別の AWS CDK アプリケーションで定義されたリソースである可能性があります。リソースの ARN (または別の識別属性、または属性のグループ) をプロキシオブジェクトに変換できます。プロキシオブジェクトは、リソースのクラスで静的ファクトリメソッドを呼び出すことで、リソースへの参照として機能します。

このようなプロキシを作成すると、外部リソースはアプリケーションの一部にはなりません。 AWS CDK したがって、 AWS CDK アプリでプロキシに加えた変更は、デプロイされたリソースには影響しません。ただし、プロキシは、そのタイプのリソースを必要とする任意の AWS CDK メソッドに渡すことができます。

次の例は、ARN arn:aws:s3::my-bucket-name を持つ既存のバケットと、特定の ID を持つ既存の VPC に基づく Amazon Virtual Private Cloud に基づいてバケットを参照する方法を示しています。

TypeScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde',
```

```
});
```

JavaScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'my-bucket-name');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::my-bucket-name');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde'
});
```

Python

```
# Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.from_bucket_name(self, "MyBucket", "my-bucket-name")

# Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.from_bucket_arn(self, "MyBucket", "arn:aws:s3:::my-bucket-name")

# Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")
```

Java

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.fromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.fromBucketArn(this, "MyBucket",
        "arn:aws:s3:::my-bucket-name");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
        .vpcId("vpc-1234567890abcdef").build());
```

C#

```
// Construct a proxy for a bucket by its name (must be same account)
```

```
Bucket.FromBucketName(this, "MyBucket", "my-bucket-name");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.FromBucketArn(this, "MyBucket", "arn:aws:s3:::my-bucket-name");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
{
    VpcId = "vpc-1234567890abcdef"
});
```

Go

```
// Define a proxy for a bucket by its name (must be same account)
s3.Bucket_FromBucketName(stack, jsii.String("MyBucket"),
 jsii.String("MyBucketName"))

// Define a proxy for a bucket by its full ARN (can be another account)
s3.Bucket_FromBucketArn(stack, jsii.String("MyBucket"),
 jsii.String("arn:aws:s3:::my-bucket-name"))

// Define a proxy for an existing VPC from its attributes
ec2.Vpc_FromVpcAttributes(stack, jsii.String("MyVpc"), &ec2.VpcAttributes{
  VpcId: jsii.String("vpc-1234567890abcde"),
})
```

`Vpc.fromLookup()` メソッドを詳しく見てみましょう。`ec2.Vpc` コンストラクトは複雑なため、CDK アプリで使用する VPC を選択する方法は多数あります。これに対処するために、VPC コンストラクトには`fromLookup`静的メソッド (Python: `from_lookup`) があり、合成時に AWS アカウントをクエリすることで目的の Amazon VPC を検索できます。

を使用するには`Vpc.fromLookup()`、スタックを合成するシステムが Amazon VPC を所有するアカウントにアクセスできる必要があります。これは、CDK Toolkit がアカウントをクエリして、合成時に適切な Amazon VPC を見つけるためです。

さらに、 は明示的なアカウントとリージョンで定義されたスタックでのみ`Vpc.fromLookup()`機能します (「」を参照the section called “環境”)。が環境に依存しないスタック から Amazon VPC を検索 AWS CDK しようとすると、CDK Toolkit は VPC を見つけるためにクエリする環境を認識しません。 ???

アカウント AWS 内の VPC を一意に識別するのに十分なVpc.fromLookup()属性を指定する必要があります。例えば、デフォルト VPC は 1 つしか存在できないため、デフォルトとして VPC を指定すれば十分です。

TypeScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

Python

```
ec2.Vpc.from_lookup(self, "DefaultVpc", is_default=True)
```

Java

```
Vpc.fromLookup(this, "DefaultVpc", VpcLookupOptions.builder()
        .isDefault(true).build());
```

C#

```
Vpc.FromLookup(this, id = "DefaultVpc", new VpcLookupOptions { IsDefault = true });
```

Go

```
ec2.Vpc_FromLookup(this, jsii.String("DefaultVpc"), &ec2.VpcLookupOptions{
  IsDefault: jsii.Bool(true),
})
```

tags プロパティを使用して、タグで VPCsクエリすることもできます。 AWS CloudFormation または を使用して、作成時に Amazon VPC にタグを追加できます AWS CDK。、、 AWS CLIまたは AWS SDK を使用して AWS Management Console、作成後にいつでもタグを編集できます。自分で

追加したタグに加えて、 は作成するすべての VPC に次のタグ AWS CDK を自動的に追加します。VPCs

- 名前 – VPC の名前。
- aws-cdk:subnet-name – サブネットの名前。
- aws-cdk:subnet-type – サブネットのタイプ: Public、Private、または Isolated。

TypeScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

Python

```
ec2.Vpc.from_lookup(self, "PublicVpc",
    tags={"aws-cdk:subnet-type": "Public"})
```

Java

```
Vpc.fromLookup(this, "PublicVpc", VpcLookupOptions.builder()
        .tags(java.util.Map.of("aws-cdk:subnet-type", "Public"))  // Java 9 or later
        .build());
```

C#

```
Vpc.FromLookup(this, id = "PublicVpc", new VpcLookupOptions
     { Tags = new Dictionary<string, string> { ["aws-cdk:subnet-type"] =
 "Public" });
```

Go

```
ec2.Vpc_FromLookup(this, jsii.String("DefaultVpc"), &ec2.VpcLookupOptions{
  Tags: &map[string]*string{"aws-cdk:subnet-type": jsii.String("Public")},
})
```

の結果Vpc.fromLookup()はプロジェクトの cdk.context.json ファイルにキャッシュされます。(「the section called “Context”」を参照してください。) このファイルをバージョン管理にコミットして、アプリケーションが同じ Amazon VPC を引き続き参照できるようにします。これは、後で VPCsが選択されるように変更した場合でも機能します。これは、CDK Pipelines などの VPC を定義する AWS アカウントにアクセスできない環境にスタックをデプロイする場合に特に重要です。

外部リソースは、 AWS CDK アプリで定義された同様のリソースを使用する任意の場所で使用できますが、変更することはできません。例えば、外部で addToResourcePolicy (Python: add_to_resource_policy) を呼び出すと、s3.Bucket何も行われません。

リソースの物理名

のリソースの論理名 AWS CloudFormation は、 によってデプロイされた後に表示される AWS Management Console リソースの名前とは異なります AWS CloudFormation。は、これらの最終名の物理名 を AWS CDK 呼び出します。

例えば、Stack2MyBucket4DD88B4F前の例の論理 ID と物理名 を使用して Amazon S3 バケットを作成する AWS CloudFormation としますstack2mybucket4dd88b4f-iuv1rbv9z3to。

プロパティ *<resourceType >*Name を使用して、リソースを表すコンストラクトを作成するときに物理名を指定できます。次の例では、物理名 の Amazon S3 バケットを作成しますmy-bucket-name。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name',
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket-name'
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket-name")
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName("my-bucket-name").build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps { BucketName = "my-bucket-
name" });
```

Go

```
bucket := s3.NewBucket(this, jsii.String("MyBucket"), &s3.BucketProps{
  BucketName: jsii.String("my-bucket-name"),
})
```

リソースに物理名を割り当てると、 にいくつかの欠点があります AWS CloudFormation。最も重要なのは、リソースの作成後に変更不可能なリソースのプロパティの変更など、リソースの置換を必要とするデプロイされたリソースへの変更は、リソースに物理名が割り当てられている場合に失敗することです。その状態になった場合、唯一の解決策は AWS CloudFormation スタックを削除してから、 AWS CDK アプリを再度デプロイすることです。詳細については、AWS CloudFormation ドキュメントを参照してください。

環境間の参照を使用して AWS CDK アプリケーションを作成する場合など、 が正しく機能 AWS CDK するには物理名が必要です。このような場合、物理名を自分で設定することを希望しない場合は、 AWS CDK 名前を付けます。そのためには、PhysicalName.GENERATE_IF_NEEDED次のように特別な値 を使用します。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED,
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket",
                           bucket_name=core.PhysicalName.GENERATE_IF_NEEDED)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName(PhysicalName.GENERATE_IF_NEEDED).build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps
    { BucketName = PhysicalName.GENERATE_IF_NEEDED });
```

Go

```
bucket := s3.NewBucket(this, jsii.String("MyBucket"), &s3.BucketProps{
  BucketName: awscdk.PhysicalName_GENERATE_IF_NEEDED(),
})
```

一意のリソース識別子を渡す

可能な限り、前のセクションで説明したように、リソースをリファレンスで渡す必要があります。ただし、その属性のいずれかでリソースを参照する以外に選択肢がない場合もあります。ユースケースの例は次のとおりです。

- 低レベルの AWS CloudFormation リソースを使用している場合。
- 環境変数を介して Lambda 関数を参照する場合など、リソースを AWS CDK アプリケーションのランタイムコンポーネントに公開する必要がある場合。

これらの識別子は、次のようなリソースの属性として使用できます。

TypeScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

JavaScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

Python

```
bucket.bucket_name
lambda_func.function_arn
security_group_arn
```

Java

Java AWS CDK バインディングは、属性にゲッターメソッドを使用します。

```
bucket.getBucketName()
lambdaFunc.getFunctionArn()
securityGroup.getGroupArn()
```

C#

```
bucket.BucketName
lambdaFunc.FunctionArn
securityGroup.GroupArn
```

Go

```
bucket.BucketName()
fn.FunctionArn()
```

次の例は、生成されたバケット名を AWS Lambda 関数に渡す方法を示しています。

TypeScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
```

```
    BUCKET_NAME: bucket.bucketName,
  },
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "Bucket")

lambda.Function(self, "MyLambda", environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "Bucket");

Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Java 9 or later
                "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new Bucket(this, "Bucket");

new Function(this, "MyLambda", new FunctionProps
{
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

Go

```
bucket := s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{})
lambda.NewFunction(this, jsii.String("MyLambda"), &lambda.FunctionProps{
  Environment: &map[string]*string{"BUCKET_NAME": bucket.BucketName()},
})
```

リソース間のアクセス許可の付与

高レベルのコンストラクトでは、アクセス許可の要件を表現するためのシンプルなインテントベースの APIs を提供することで、最小特権のアクセス許可を実現できます。例えば、多くの L2 コンストラクトは、IAM アクセス許可ステートメントを手動で作成しなくても、リソースを操作するアクセス許可をエンティティ (IAM ロールやユーザーなど) に付与するために使用できる付与メソッドを提供します。

次の例では、Lambda 関数の実行ロールが特定の Amazon S3 バケットに対してオブジェクトを読み書きできるようにするアクセス許可を作成します。Amazon S3 バケットが AWS KMS キーで暗号化されている場合、このメソッドは Lambda 関数の実行ロールにキーで復号するアクセス許可も付与します。

TypeScript

```
if (bucket.grantReadWrite(func).success) {
  // ...
}
```

JavaScript

```
if ( bucket.grantReadWrite(func).success) {
  // ...
}
```

Python

```
if bucket.grant_read_write(func).success:
    # ...
```

Java

```
if (bucket.grantReadWrite(func).getSuccess()) {
```

```
    // ...
}
```

C#

```
if (bucket.GrantReadWrite(func).Success)
{
    // ...
}
```

Go

```
if *bucket.GrantReadWrite(function, nil).Success() {
  // ...
}
```

グラントメソッドは `iam.Grant` オブジェクトを返します。`Grant` オブジェクトの `success` 属性を使用して、グラントが効果的に適用されたかどうかを判断します (例えば、外部リソースに適用されていない可能性があります)。`Grant` オブジェクトの `assertSuccess` (Python: `assert_success`) メソッドを使用して、グラントが正常に適用されたように強制することもできます。

特定のグラントメソッドが特定のユースケースで利用できない場合は、一般的なグラントメソッドを使用して、指定されたアクションのリストで新しいグラントを定義できます。

次の例は、Amazon DynamoDB `CreateBackup`アクションへのアクセス権を Lambda 関数に付与する方法を示しています。

TypeScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

JavaScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

Python

```
table.grant(func, "dynamodb:CreateBackup")
```

Java

```
table.grant(func, "dynamodb:CreateBackup");
```

C#

```
table.Grant(func, "dynamodb:CreateBackup");
```

Go

```
table := dynamodb.NewTable(this, jsii.String("MyTable"), &dynamodb.TableProps{})
table.Grant(function, jsii.String("dynamodb:CreateBackup"))
```

Lambda 関数などの多くのリソースでは、コードの実行時にロールを引き受ける必要があります。設定プロパティを使用すると、 を指定できますiam.IRole。ロールが指定されていない場合、関数はこの使用専用のロールを自動的に作成します。その後、リソースのグラントメソッドを使用して、ロールにステートメントを追加できます。

許可メソッドは、IAM ポリシーで処理するための下位レベルの APIs を使用して構築されます。ポリシーはPolicyDocumentオブジェクトとしてモデル化されます。addToRolePolicy メソッド (Python:) を使用して ステートメントをロール (またはコンストラクトのアタッチされたロールadd_to_role_policy) に直接追加するか、 (Python:) メソッドを使用してリソースのポリシーaddToResourcePolicy (Bucketポリシーなどadd_to_resource_policy) に直接追加します。

リソースメトリクスとアラーム

多くのリソースは、モニタリングダッシュボードとアラームの設定に使用できる CloudWatch メトリクスを出力します。高レベルのコンストラクトには、使用する正しい名前を検索せずにメトリクスにアクセスできるメトリクスメソッドがあります。

次の例は、Amazon SQS キューApproximateNumberOfMessagesNotVisibleの が 100 を超えたときにアラームを定義する方法を示しています。

TypeScript

```
import * as cw from '@aws-cdk/aws-cloudwatch';
import * as sqs from '@aws-cdk/aws-sqs';
import { Duration } from '@aws-cdk/core';

const queue = new sqs.Queue(this, 'MyQueue');
```

```
const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5),
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100,
  // ...
});
```

JavaScript

```
const cw = require('@aws-cdk/aws-cloudwatch');
const sqs = require('@aws-cdk/aws-sqs');
const { Duration } = require('@aws-cdk/core');

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5)
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100
  // ...
});
```

Python

```
import aws_cdk.aws_cloudwatch as cw
import aws_cdk.aws_sqs as sqs
from aws_cdk.core import Duration

queue = sqs.Queue(self, "MyQueue")
metric = queue.metric_approximate_number_of_messages_not_visible(
    label="Messages Visible (Approx)",
    period=Duration.minutes(5),
    # ...
)
```

```
metric.create_alarm(self, "TooManyMessagesAlarm",
    comparison_operator=cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    threshold=100,
    # ...
)
```

Java

```
import software.amazon.awscdk.core.Duration;
import software.amazon.awscdk.services.sqs.Queue;
import software.amazon.awscdk.services.cloudwatch.Metric;
import software.amazon.awscdk.services.cloudwatch.MetricOptions;
import software.amazon.awscdk.services.cloudwatch.CreateAlarmOptions;
import software.amazon.awscdk.services.cloudwatch.ComparisonOperator;

Queue queue = new Queue(this, "MyQueue");

Metric metric = queue
        .metricApproximateNumberOfMessagesNotVisible(MetricOptions.builder()
                .label("Messages Visible (Approx)")
                .period(Duration.minutes(5)).build());

metric.createAlarm(this, "TooManyMessagesAlarm", CreateAlarmOptions.builder()
                .comparisonOperator(ComparisonOperator.GREATER_THAN_THRESHOLD)
                .threshold(100)
                // ...
                .build());
```

C#

```
using cdk = Amazon.CDK;
using cw = Amazon.CDK.AWS.CloudWatch;
using sqs = Amazon.CDK.AWS.SQS;

var queue = new sqs.Queue(this, "MyQueue");
var metric = queue.MetricApproximateNumberOfMessagesNotVisible(new cw.MetricOptions
{
    Label = "Messages Visible (Approx)",
    Period = cdk.Duration.Minutes(5),
    // ...
});
metric.CreateAlarm(this, "TooManyMessagesAlarm", new cw.CreateAlarmOptions
{
```

```
    ComparisonOperator = cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    Threshold = 100,
    // ..
});
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  cw "github.com/aws/aws-cdk-go/awscdk/v2/awscloudwatch"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

queue := sqs.NewQueue(this, jsii.String("MyQueue"), &sqs.QueueProps{})
metric := queue.MetricApproximateNumberOfMessagesNotVisible(&cw.MetricOptions{
  Label: jsii.String("Messages Visible (Approx)"),
  Period: awscdk.Duration_Minutes(jsii.Number(5)),
})

metric.CreateAlarm(this, jsii.String("TooManyMessagesAlarm"),
 &cw.CreateAlarmOptions{
  ComparisonOperator: cw.ComparisonOperator_GREATER_THAN_THRESHOLD,
  Threshold: jsii.Number(100),
})
```

特定のメトリクスにメソッドがない場合は、一般的なメトリクスメソッドを使用してメトリクス名を手動で指定できます。

メトリクスは CloudWatch ダッシュボードに追加することもできます。「CloudWatch」を参照してください。

ネットワークトラフィック

多くの場合、コンピューティングインフラストラクチャが永続化レイヤーにアクセスする必要がある場合など、アプリケーションが機能するためには、ネットワークに対するアクセス許可を有効にする必要があります。接続を確立またはリッスンするリソースは、セキュリティグループルールやネットワーク ACLs。

IConnectable リソースには、ネットワークトラフィックルール設定へのゲートウェイである connections プロパティがあります。

allow メソッドを使用して、特定のネットワークパスでデータをフローできるようにします。次の例では、ウェブへの HTTPS 接続と Amazon EC2 Auto Scaling グループ からの受信接続を有効にしますfleet2。

TypeScript

```
import * as asg from '@aws-cdk/aws-autoscaling';
import * as ec2 from '@aws-cdk/aws-ec2';

const fleet1: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

JavaScript

```
const asg = require('@aws-cdk/aws-autoscaling');
const ec2 = require('@aws-cdk/aws-ec2');

const fleet1 = asg.AutoScalingGroup();

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2 = asg.AutoScalingGroup();
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

Python

```
import aws_cdk.aws_autoscaling as asg
import aws_cdk.aws_ec2 as ec2

fleet1 = asg.AutoScalingGroup( ... )

# Allow surfing the (secure) web
fleet1.connections.allow_to(ec2.Peer.any_ipv4(),
  ec2.Port(PortProps(from_port=443, to_port=443)))
```

```
fleet2 = asg.AutoScalingGroup( ... )
fleet1.connections.allow_from(fleet2, ec2.Port.all_traffic())
```

Java

```
import software.amazon.awscdk.services.autoscaling.AutoScalingGroup;
import software.amazon.awscdk.services.ec2.Peer;
import software.amazon.awscdk.services.ec2.Port;

AutoScalingGroup fleet1 = AutoScalingGroup.Builder.create(this, "MyFleet")
        /* ... */.build();

// Allow surfing the (secure) Web
fleet1.getConnections().allowTo(Peer.anyIpv4(),
        Port.Builder.create().fromPort(443).toPort(443).build());

AutoScalingGroup fleet2 = AutoScalingGroup.Builder.create(this, "MyFleet2")
        /* ... */.build();
fleet1.getConnections().allowFrom(fleet2, Port.allTraffic());
```

C#

```
using cdk = Amazon.CDK;
using asg = Amazon.CDK.AWS.AutoScaling;
using ec2 = Amazon.CDK.AWS.EC2;

// Allow surfing the (secure) Web
var fleet1 = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
 { /* ... */ });
fleet1.Connections.AllowTo(ec2.Peer.AnyIpv4(), new ec2.Port(new ec2.PortProps
  { FromPort = 443, ToPort = 443 });

var fleet2 = new asg.AutoScalingGroup(this, "MyFleet2", new
 asg.AutoScalingGroupProps { /* ... */ });
fleet1.Connections.AllowFrom(fleet2, ec2.Port.AllTraffic());
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
```

```
  autoscaling "github.com/aws/aws-cdk-go/awscdk/v2/awsautoscaling"
  ec2 "github.com/aws/aws-cdk-go/awscdk/v2/awsec2"
)

fleet1 := autoscaling.NewAutoScalingGroup(this, jsii.String("MyFleet1"),
 &autoscaling.AutoScalingGroupProps{})
fleet1.Connections().AllowTo(ec2.Peer_AnyIpv4(),ec2.NewPort(&ec2.PortProps{ FromPort:
 jsii.Number(443), ToPort: jsii.Number(443) }),jsii.String("secure web"))

fleet2 := autoscaling.NewAutoScalingGroup(this, jsii.String("MyFleet2"),
 &autoscaling.AutoScalingGroupProps{})
fleet1.Connections().AllowFrom(fleet2, ec2.Port_AllTraffic(),jsii.String("all
 traffic"))
```

特定のリソースには、デフォルトのポートが関連付けられています。例としては、パブリックポートのロードバランサーのリスナーや、データベースエンジンが Amazon RDS データベースのインスタンスの接続を受け入れるポートなどがあります。このような場合、ポートを手動で指定しなくても、厳密なネットワーク制御を適用できます。そのためには、 `allowDefaultPortFrom`メソッドと `allowToDefaultPort`メソッド (Python: `allow_default_port_from`、) を使用しますallow_to_default_port。

次の例は、任意の IPV4 アドレスからの接続と、Auto Scaling グループからのデータベースへのアクセスを有効にする方法を示しています。

TypeScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

JavaScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

Python

```
listener.connections.allow_default_port_from_any_ipv4("Allow public access")
```

```
fleet.connections.allow_to_default_port(rds_database, "Fleet can access database")
```

Java

```
listener.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");

fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

C#

```
listener.Connections.AllowDefaultPortFromAnyIpv4("Allow public access");

fleet.Connections.AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

Go

```
listener.Connections().AllowDefaultPortFromAnyIpv4(jsii.String("Allow public
 Access"))
fleet.Connections().AllowToDefaultPort(rdsDatabase, jsii.String("Fleet can access
 database"))
```

イベント処理

一部のリソースはイベントソースとして機能します。addEventNotification メソッド (Python: add_event_notification) を使用して、 リソースによって出力される特定のイベントタイプにイベントターゲットを登録します。これに加えて、 addXxxNotificationメソッドは一般的なイベントタイプのハンドラーを簡単に登録する方法を提供します。

次の例は、オブジェクトが Amazon S3 バケットに追加されたときに Lambda 関数をトリガーする方法を示しています。

TypeScript

```
import * as s3nots from '@aws-cdk/aws-s3-notifications';

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

JavaScript

```
const s3nots = require('@aws-cdk/aws-s3-notifications');

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

Python

```
import aws_cdk.aws_s3_notifications as s3_nots

handler = lambda_.Function(self, "Handler", ...)
bucket = s3.Bucket(self, "Bucket")
bucket.add_object_created_notification(s3_nots.LambdaDestination(handler))
```

Java

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.s3.notifications.LambdaDestination;

Function handler = Function.Builder.create(this, "Handler")/* ... */.build();
Bucket bucket = new Bucket(this, "Bucket");
bucket.addObjectCreatedNotification(new LambdaDestination(handler));
```

C#

```
using lambda = Amazon.CDK.AWS.Lambda;
using s3 = Amazon.CDK.AWS.S3;
using s3Nots = Amazon.CDK.AWS.S3.Notifications;

var handler = new lambda.Function(this, "Handler", new lambda.FunctionProps { .. });
var bucket = new s3.Bucket(this, "Bucket");
bucket.AddObjectCreatedNotification(new s3Nots.LambdaDestination(handler));
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  s3 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
```

```
  s3nots "github.com/aws/aws-cdk-go/awscdk/v2/awss3notifications"
)

handler := lambda.NewFunction(this, jsii.String("MyFunction"),
 &lambda.FunctionProps{})
bucket := s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{})
bucket.AddObjectCreatedNotification(s3nots.NewLambdaDestination(handler), nil)
```

削除ポリシー

データベース、Amazon S3 バケット、Amazon ECR レジストリなどの永続的データを維持するリソースには、削除ポリシー があります。削除ポリシーは、永続オブジェクトを含むスタックが破棄されたときに AWS CDK 永続オブジェクトを削除するかどうかを示します。削除ポリシーを指定する値は、モジュールの`RemovalPolicy`列挙 AWS CDK `core`を通じて使用できます。

> **Note**
>
> データを継続的に保存するリソース以外のリソースにも、別の目的で使用される `removalPolicy`がある場合があります。例えば、Lambda 関数のバージョンは `removalPolicy` 属性を使用して、新しいバージョンがデプロイされたときに特定のバージョンが保持されるかどうかを判断します。これらは、Amazon S3 バケットまたは DynamoDB テーブルの削除ポリシーとは異なる意味とデフォルトを持ちます。 DynamoDB

値	意味
`RemovalPolicy.RETAIN`	スタックを破棄するときは、リソースの内容を保持します (デフォルト) 。リソースはスタックから孤立しているため、手動で削除する必要があります。リソースがまだ存在している間にスタックを再デプロイしようとすると、名前の競合によりエラーメッセージが表示されます。
`RemovalPolicy.DESTROY`	リソースはスタックとともに破棄されます。

AWS CloudFormation は、削除ポリシーが に設定されている場合でも、ファイルを含む Amazon S3 バケットを削除しません`DESTROY`。試行は AWS CloudFormation エラーです。でバケットからすべ

てのファイル AWS CDK を削除してから破棄するには、バケットの autoDeleteObjectsプロパティを に設定しますtrue。

以下は、 を RemovalPolicyDESTROYに設定し、 を に設定して Amazon S3 バケットを作成する例ですtrue。 autoDeleteOjbects

TypeScript

```
import * as cdk from '@aws-cdk/core';
import * as s3 from '@aws-cdk/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}
```

JavaScript

```
const cdk = require('@aws-cdk/core');
const s3 = require('@aws-cdk/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk.core as cdk
```

```
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY,
            auto_delete_objects=True)
```

Java

```
software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.s3.*;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY)
                .autoDeleteObjects(true).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY,
        AutoDeleteObjects = true
    });
}
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  s3 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
)

s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{
  RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
  AutoDeleteObjects: jsii.Bool(true),
})
```

削除ポリシーは、 `applyRemovalPolicy()`メソッドを使用して基盤となる AWS CloudFormation リソースに直接適用することもできます。このメソッドは、L2 リソースの props に `removalPolicy`プロパティを持たないステートフルリソースで使用できます。次に例を示します。

- AWS CloudFormation スタック
- Amazon Cognito ユーザープール
- Amazon DocumentDB データベースインスタンス
- Amazon EC2 ボリューム
- Amazon OpenSearch Service ドメイン
- Amazon FSx ファイルシステム
- Amazon SQS キュー

TypeScript

```
const resource = bucket.node.findChild('Resource') as cdk.CfnResource;
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

JavaScript

```
const resource = bucket.node.findChild('Resource');
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

Python

```
resource = bucket.node.find_child('Resource')
```

```
resource.apply_removal_policy(cdk.RemovalPolicy.DESTROY);
```

Java

```
CfnResource resource = (CfnResource)bucket.node.findChild("Resource");
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

C#

```
var resource = (CfnResource)bucket.node.findChild('Resource');
resource.ApplyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

> **Note**
>
> AWS CDKの は AWS CloudFormationの `RemovalPolicy`に変換されま すDeletionPolicy。ただし、 のデフォルト AWS CDK はデータを保持することであり、これは AWS CloudFormation デフォルトとは逆です。

識別子

AWS Cloud Development Kit (AWS CDK) アプリケーションを構築するときは、さまざまなタイプの識別子と名前を使用します。 AWS CDK を効果的に使用し、エラーを回避するには、識別子のタイプを理解することが重要です。

識別子は、作成された範囲内で一意である必要があります。 AWS CDK アプリケーションでグローバルに一意である必要はありません。

同じ範囲内で同じ値を持つ識別子を作成しようとすると、 は例外を AWS CDK スローします。

トピック

コンストラクト IDs

最も一般的な識別子である はid、コンストラクトオブジェクトをインスタンス化するときに 2 番目の引数として渡される識別子です。この識別子は、すべての識別子と同様に、作成先のスコープ内で一意である必要があります。これは、コンストラクトオブジェクトをインスタンス化する最初の引数です。

> **Note**
>
> スタックidの は、 で参照するために使用する識別子でもありますthe section called “AWS CDK ツールキット”。

アプリMyBucketに 識別子を持つ 2 つのコンストラクトがある例を見てみましょう。1 つ目は、識別子 を持つスタックのスコープで定義されますStack1。2 つ目は、識別子 を持つスタックのスコープで定義されますStack2。これらは異なるスコープで定義されているため、競合は発生せず、問題なく同じアプリに共存できます。

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

class MyStack extends Stack {
  constructor(scope: Construct, id: string, props: StackProps = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');
```

```
class MyStack extends Stack {
  constructor(scope, id, props = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

Python

```
from aws_cdk import App, Construct, Stack, StackProps
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MyStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)
        s3.Bucket(self, "MyBucket")

app = App()
MyStack(app, 'Stack1')
MyStack(app, 'Stack2')
```

Java

```
// MyStack.java
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.services.s3.Bucket;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
```

```
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.App;

public class Main {
    public static void main(String[] args) {
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

C#

```
using Amazon.CDK;
using constructs;
using Amazon.CDK.AWS.S3;

public class MyStack : Stack
{
    public MyStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
    {
        new Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
```

```
    }
}
```

パス

AWS CDK アプリケーションのコンストラクトは、 App クラスにルートされた階層を形成します。特定のコンストラクト、その親コンストラクト、その祖先などのコンストラクトツリーのルートからの IDs のコレクションをパス と呼びます。

AWS CDK には通常、テンプレート内のパスを文字列として表示します。レベルの IDs はスラッシュで区切られ、通常はスタックであるルートAppインスタンスのすぐ下にあるノードから始まります。例えば、前のコード例の 2 つの Amazon S3 バケットリソースのパスは `Stack1/MyBucket`と ですStack2/MyBucket。

次の例に示すように、任意のコンストラクトのパスにプログラムでアクセスできます。これにより、myConstruct (または my_constructPython デベロッパーが記述する) のパスを取得します。IDs は作成されるスコープ内で一意でなければならないため、パスは常に AWS CDK アプリケーション内で一意です。

TypeScript

```
const path: string = myConstruct.node.path;
```

JavaScript

```
const path = myConstruct.node.path;
```

Python

```
path = my_construct.node.path
```

Java

```
String path = myConstruct.getNode().getPath();
```

C#

```
string path = myConstruct.Node.Path;
```

一意の ID

AWS CloudFormation では、テンプレート内のすべての論理 IDs が一意である必要があります。このため、 はアプリケーション内のコンストラクトごとに一意の識別子を生成できる AWS CDK 必要があります。リソースには、グローバルに一意なパス (スタックから特定のリソースへのすべてのスコープの名前) があります。したがって、 はパスの要素を連結し、8 桁のハッシュを追加することで、必要な一意の識別子 AWS CDK を生成します。 (ハッシュは、同じ AWS CloudFormation 識別子になる A/B/Cや などの個別のパスを区別するために必要です。 AWS CloudFormation 識別子は英数字でありA/BC、スラッシュやその他の区切り文字を含めることはできません) 。は、この文字列をコンストラクトの一意の ID として AWS CDK 呼び出します。

一般的に、 AWS CDK アプリは一意の IDsについて知る必要はありません。ただし、次の例に示すように、任意のコンストラクトの一意の ID にプログラムでアクセスできます。

TypeScript

```
const uid: string = Names.uniqueId(myConstruct);
```

JavaScript

```
const uid = Names.uniqueId(myConstruct);
```

Python

```
uid = Names.unique_id(my_construct)
```

Java

```
String uid = Names.uniqueId(myConstruct);
```

C#

```
string uid = Names.Uniqueid(myConstruct);
```

アドレスは、CDK リソースを一意に区別する別の種類の一意の識別子です。パスの SHA-1 ハッシュから派生したもので、人間が読み取れません。ただし、定数の比較的短い長さ (常に 42 桁の 16 進数文字) は、「従来の」一意の ID が長すぎる可能性がある状況で役立ちます。一部のコンストラクト

では、一意の ID の代わりに合成された AWS CloudFormation テンプレートのアドレスを使用できます。繰り返しになりますが、アプリケーションは通常、コンストラクトのアドレスを知る必要はありませんが、コンストラクトのアドレスは次のように取得できます。

TypeScript

```
const addr: string = myConstruct.node.addr;
```

JavaScript

```
const addr = myConstruct.node.addr;
```

Python

```
addr = my_construct.node.addr
```

Java

```
String addr = myConstruct.getNode().getAddr();
```

C#

```
string addr = myConstruct.Node.Addr;
```

論理 IDs

一意の IDs、リソースを表すコンストラクト用に生成された AWS CloudFormation テンプレート内の AWS リソースの論理識別子 (または論理名) として機能します。

例えば、前の例で 内に作成された Amazon S3 バケットは、 `AWS::S3::Bucket`リソースStack2になります。リソースの論理 ID は、結果の AWS CloudFormation テンプレートStack2MyBucket4DD88B4Fにあります。(この識別子の生成方法の詳細については、「」を参照してくださいthe section called “一意の ID”。)

論理 ID の安定性

リソースの作成後に論理 ID を変更することは避けてください。 は、リソースを論理 ID で AWS CloudFormation 識別します。したがって、リソースの論理 ID を変更すると、 は新しい論理 ID を使

用して新しいリソース AWS CloudFormation を作成し、既存のリソースを削除します。リソースのタイプによっては、サービスの中断、データ損失、またはその両方が発生する可能性があります。

トークン

トークンは、アプリケーションライフサイクル で後でしか解決できない値を表します。例えば、CDK アプリケーションで定義した Amazon Simple Storage Service (Amazon S3) バケットの名前は、 AWS CloudFormation テンプレートが合成されている場合にのみ割り当てられます。文字列である `bucket.bucketName` 属性を出力すると、次のような内容が含まれていることがわかります。

```
${TOKEN[Bucket.Name.1234]}
```

これは、 が構築時に値がまだ認識されていないが、後で利用可能になるトークンを AWS CDK エンコードする方法です。は、これらのプレースホルダートークンを AWS CDK 呼び出します。 この場合、これは文字列としてエンコードされたトークンです。

この文字列は、バケットの名前であるかのように渡すことができます。次の例では、バケット名は AWS Lambda 関数の環境変数として指定されています。

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  }
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket');

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
```

```
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket")

fn = lambda_.Function(stack, "MyLambda",
        environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "MyBucket");

Function fn = Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Map.of requires Java 9+
            "BUCKET_NAME", bucket.getBucketName()))
        .build();
```

C#

```
var bucket = new s3.Bucket(this, "MyBucket");

var fn = new Function(this, "MyLambda", new FunctionProps {
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

AWS CloudFormation テンプレートが最後に合成されると、トークンは AWS CloudFormation 組み込み としてレンダリングされます`{ "Ref": "MyBucket" }`。デプロイ時に、 はこの組み込みを、作成されたバケットの実際の名前 AWS CloudFormation に置き換えます。

トピック

- トークンとトークンエンコーディング
- 文字列エンコードされたトークン
- リストエンコードされたトークン
- 数値エンコードされたトークン

- 遅延値
- JSON への変換

トークンとトークンエンコーディング

トークンは、単一のresolveメソッドを含む IResolvable インターフェイスを実装するオブジェクトです。は、合成中にこのメソッドを AWS CDK AWS CloudFormation 呼び出して、テンプレートの最終値を生成します。トークンは合成プロセスに関与し、任意のタイプの値を生成します。

> **Note**
>
> IResolvable インターフェイスを直接操作することはほとんどありません。ほとんどの場合、トークンの文字列エンコードされたバージョンのみが表示されます。

他の関数は通常、 stringや などの基本型の引数のみを受け入れますnumber。このような場合にトークンを使用するには、cdk.Token クラスで静的メソッドを使用して、3 つのタイプのいずれかにエンコードできます。

- Token.asString 文字列エンコーディングを生成する (またはトークンオブジェクト.toString()で を呼び出す)
- Token.asList リストエンコーディングを生成するには
- Token.asNumber 数値エンコーディングを生成するには

これらは である任意の値を取りIResolvable、示されたタイプのプリミティブ値にエンコードします。

> **Important**
>
> 前述のタイプのいずれかがエンコードされたトークンである可能性があるため、コンテンツを解析または読み込もうとするときは注意が必要です。例えば、文字列を解析してその文字列から値を抽出しようとしたときに、その文字列がエンコードされたトークンである場合、解析は失敗します。同様に、配列の長さをクエリしたり、数値を使用して数学演算を実行したりする場合は、まずそれらがエンコードされたトークンでないことを確認する必要があります。

値に未解決のトークンが含まれているかどうかを確認するには、 `Token.isUnresolved` (Python: `is_unresolved`) メソッドを呼び出します。

次の例では、トークンである可能性がある文字列値が 10 文字を超えないことを検証します。

TypeScript

```
if (!Token.isUnresolved(name) && name.length > 10) {
  throw new Error(`Maximum length for name is 10 characters`);
}
```

JavaScript

```
if ( !Token.isUnresolved(name) && name.length > 10) {
  throw ( new Error(`Maximum length for name is 10 characters`));
}
```

Python

```
if not Token.is_unresolved(name) and len(name) > 10:
    raise ValueError("Maximum length for name is 10 characters")
```

Java

```
if (!Token.isUnresolved(name) && name.length() > 10)
    throw new IllegalArgumentException("Maximum length for name is 10 characters");
```

C#

```
if (!Token.IsUnresolved(name) && name.Length > 10)
    throw new ArgumentException("Maximum length for name is 10 characters");
```

名前がトークンの場合、検証は実行されず、デプロイ中など、ライフサイクルの後の段階でエラーが発生する可能性があります。

> **Note**
>
> トークンエンコーディングを使用して、タイプシステムをエスケープできます。例えば、合成時に数値を生成するトークンを文字列エンコードできます。これらの関数を使用する場合は、合成後にテンプレートが使用可能な状態に解決されるようにする責任があります。

文字列エンコードされたトークン

文字列エンコードされたトークンは次のようになります。

```
${TOKEN[Bucket.Name.1234]}
```

次の例に示すように、通常の文字列のように渡したり、連結したりできます。

TypeScript

```
const functionName = bucket.bucketName + 'Function';
```

JavaScript

```
const functionName = bucket.bucketName + 'Function';
```

Python

```
function_name = bucket.bucket_name + "Function"
```

Java

```
String functionName = bucket.getBucketName().concat("Function");
```

C#

```
string functionName = bucket.BucketName + "Function";
```

次の例に示すように、言語がサポートしている場合は、文字列補間を使用することもできます。

TypeScript

```
const functionName = `${bucket.bucketName}Function`;
```

JavaScript

```
const functionName = `${bucket.bucketName}Function`;
```

Python

```
function_name = f"{bucket.bucket_name}Function"
```

Java

```
String functionName = String.format("%sFunction". bucket.getBucketName());
```

C#

```
string functionName = $"${bucket.bucketName}Function";
```

文字列を他の方法で操作することは避けてください。例えば、文字列の部分文字列を取ると、文字列トークンが壊れる可能性があります。

リストエンコードされたトークン

リストエンコードされたトークンは次のようになります。

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

これらのリストを操作する唯一の安全な方法は、他のコンストラクトに直接渡すことです。文字列リスト形式のトークンは連結できず、トークンから要素を取得することもできません。これらを操作する唯一の安全な方法は、Fn.select などの AWS CloudFormation 組み込み関数を使用することです。

数値エンコードされたトークン

数値エンコードされたトークンは、次のような小さな負の浮動小数点数のセットです。

```
-1.8881545897087626e+289
```

リストトークンと同様に、数値を変更することはできません。変更すると、数値トークンが破損する可能性があります。許可されるオペレーションは、値を別のコンストラクトに渡すことです。

遅延値

トークンは、 AWS CloudFormation パラメータ などのデプロイ時値を表すだけでなく、一般的に合成時遅延値を表すために使用されます。これらは、合成が完了する前に最終的な値を決定する値であり、値が構築された時点では決定されません。トークンを使用してリテラル文字列または数値を別のコンストラクトに渡しますが、合成時の実際の値は、まだ発生していない計算に依存する場合があります。

Lazy.string や Lazy.number などのLazyクラスの静的メソッドを使用して、同期時のレイジー値を表すトークンを作成できます。これらのメソッドは、 produce プロパティがコンテキスト引数を受け入れる関数であるオブジェクトを受け入れ、呼び出されたときに最終的な値を返します。

次の例では、キャパシティーが作成後に決定される Auto Scaling グループを作成します。

TypeScript

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

JavaScript

```
let actualValue;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return (actualValue);
    }
  })
```

```
});

// At some later point
actualValue = 10;
```

Python

```
class Producer:
    def __init__(self, func):
        self.produce = func

actual_value = None

AutoScalingGroup(self, "Group",
    desired_capacity=Lazy.number_value(Producer(lambda context: actual_value))
)

# At some later point
actual_value = 10
```

Java

```
double actualValue = 0;

class ProduceActualValue implements INumberProducer {

    @Override
    public Number produce(IResolveContext context) {
        return actualValue;
    }
}

AutoScalingGroup.Builder.create(this, "Group")
    .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point
actualValue = 10;
```

C#

```
public class NumberProducer : INumberProducer
{
    Func<Double> function;
```

```
    public NumberProducer(Func<Double> function)
    {
        this.function = function;
    }

    public Double Produce(IResolveContext context)
    {
        return function();
    }
}

double actualValue = 0;

new AutoScalingGroup(this, "Group", new AutoScalingGroupProps
{
    DesiredCapacity = Lazy.NumberValue(new NumberProducer(() => actualValue))
});

// At some later point
actualValue = 10;
```

JSON への変換

任意データの JSON 文字列を生成したいが、データにトークンが含まれているかどうかわからない場合があります。トークンが含まれているかどうかにかかわらず、データ構造を適切に JSON エンコードするには、次の例に示すように、 メソッドスタックを使用します。toJsonString

TypeScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

JavaScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

Python

```
stack = Stack.of(self)
string = stack.to_json_string(dict(value=bucket.bucket_name))
```

Java

```
Stack stack = Stack.of(this);
String stringVal = stack.toJsonString(java.util.Map.of(    // Map.of requires Java
 9+
        put("value", bucket.getBucketName())));
```

C#

```
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{
    ["value"] = bucket.BucketName
});
```

パラメータ

パラメータは、デプロイ時に提供されるカスタム値です。パラメータは の機能です AWS CloudFormation。は AWS Cloud Development Kit (AWS CDK) AWS CloudFormation テンプレートを合成するため、デプロイ時間パラメータもサポートしています。

トピック

- パラメータについて
- パラメータの定義
- パラメータの使用
- パラメータを使用したデプロイ

パラメータについて

を使用して AWS CDKパラメータを定義し、作成したコンストラクトのプロパティで使用できます。パラメータを含むスタックをデプロイすることもできます。

AWS CDK Toolkit を使用して AWS CloudFormation テンプレートをデプロイする場合は、コマンドラインにパラメータ値を指定します。 AWS CloudFormation コンソールからテンプレートをデプロイすると、パラメータ値の入力を求められます。

一般に、 で AWS CloudFormation パラメータを使用しないことをお勧めします AWS CDK。アプリケーションに AWS CDK 値を渡す通常の方法は、コンテキスト値と環境変数です。合成時に使用できないため、パラメータ値は CDK アプリのフロー制御やその他の目的に簡単に使用することはできません。

> **Note**
>
> パラメータを使用してフローを制御するには、`CfnCondition`コンストラクトを使用できますが、これはネイティブ`if`ステートメントに比べて扱いにくくなります。

パラメータを使用するには、記述するコードがデプロイ時および合成時にどのように動作するかに注意する必要があります。これにより、多くの場合、ほとんど利点がないので、 AWS CDK アプリケーションを理解し、理由を説明することが難しくなります。

一般的に、CDK アプリは必要な情報を明確に定義された方法で受け入れ、それを CDK アプリでコンストラクトを直接宣言する方がよいでしょう。理想的な AWS CDK生成 AWS CloudFormation テンプレートは具体的であり、デプロイ時に指定される値は残りません。

ただし、 AWS CloudFormation パラメータが一意に適しているユースケースがあります。例えば、インフラストラクチャを定義してデプロイするチームが別々の場合は、パラメータを使用して、生成されたテンプレートをより広く有用にすることができます。また、 は AWS CloudFormation パラメータ AWS CDK をサポートしているため、 AWS CloudFormation テンプレートを使用する AWS サービス (Service Catalog など) AWS CDK で を使用できます。これらの AWS サービスは、パラメータを使用して、デプロイされるテンプレートを設定します。

パラメータの定義

`CfnParameter` クラスを使用してパラメータを定義します。ほとんどのパラメータには少なくとも 1 つのタイプと説明を指定する必要がありますが、どちらも技術的にはオプションです。説明は、ユーザーが AWS CloudFormation コンソールにパラメータの値を入力するように求められたときに表示されます。使用可能なタイプの詳細については、「タイプhttps://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html#parameters-section-structure-properties-type」を参照してください。

> **Note**
>
> パラメータは任意のスコープで定義できます。ただし、コードのリファクタリング時に論理 ID が変更されないように、スタックレベルでパラメータを定義することをお勧めします。

TypeScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

JavaScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

Python

```
upload_bucket_name = CfnParameter(self, "uploadBucketName", type="String",
    description="The name of the Amazon S3 bucket where uploaded files will be
 stored.")
```

Java

```
CfnParameter uploadBucketName = CfnParameter.Builder.create(this,
 "uploadBucketName")
        .type("String")
        .description("The name of the Amazon S3 bucket where uploaded files will be
 stored")
        .build();
```

C#

```
var uploadBucketName = new CfnParameter(this, "uploadBucketName", new
 CfnParameterProps
```

```
{
    Type = "String",
    Description = "The name of the Amazon S3 bucket where uploaded files will be
 stored"
});
```

パラメータの使用

CfnParameter インスタンスは、トークン を介してその値を AWS CDK アプリケーションに公開します。すべてのトークンと同様に、パラメータのトークンは合成時に解決されます。ただし、具体的な値ではなく、 AWS CloudFormation テンプレートで定義された パラメータ (デプロイ時に解決されます) への参照が解決されます。

トークンは、Tokenクラスのインスタンスとして取得することも、文字列、文字列リスト、または数値エンコーディングで取得することもできます。選択は、 パラメータを使用するクラスまたはメソッドに必要な値の種類によって異なります。

TypeScript

プロパティ	値の種類
value	Token クラスインスタンス
valueAsList	文字列リストとして表されるトークン
valueAsNumber	数値で表されるトークン
valueAsString	文字列として表されるトークン

JavaScript

プロパティ	値の種類
value	Token クラスインスタンス
valueAsList	文字列リストとして表されるトークン
valueAsNumber	数値で表されるトークン

プロパティ	値の種類
`valueAsString`	文字列として表されるトークン

Python

プロパティ	値の種類
`value`	Token クラスインスタンス
`value_as_list`	文字列リストとして表されるトークン
`value_as_number`	数値で表されるトークン
`value_as_string`	文字列として表されるトークン

Java

プロパティ	値の種類
`getValue()`	Token クラスインスタンス
`getValueAsList()`	文字列リストとして表されるトークン
`getValueAsNumber()`	数値で表されるトークン
`getValueAsString()`	文字列として表されるトークン

C#

プロパティ	値の種類
`Value`	Token クラスインスタンス
`ValueAsList`	文字列リストとして表されるトークン
`ValueAsNumber`	数値で表されるトークン

プロパティ	値の種類
`ValueAsString`	文字列として表されるトークン

例えば、Bucket定義で パラメータを使用するには、次のようにします。

TypeScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

JavaScript

```
const bucket = new Bucket(this, "myBucket",
  { bucketName: uploadBucketName.valueAsString});
```

Python

```
bucket = Bucket(self, "myBucket",
    bucket_name=upload_bucket_name.value_as_string)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "myBucket")
        .bucketName(uploadBucketName.getValueAsString())
        .build();
```

C#

```
var bucket = new Bucket(this, "myBucket")
{
    BucketName = uploadBucketName.ValueAsString
};
```

パラメータを使用したデプロイ

AWS CloudFormation コンソールから生成された AWS CloudFormation テンプレートをデプロイすると、各パラメータの値を指定するように求められます。

CDK CLI cdk deploy コマンドを使用するか、CDK プロジェクトのスタックファイルでパラメータ値を指定して、パラメータ値を指定することもできます。

でのパラメータ値の提供 cdk deploy

CDK CLI cdk deploy コマンドを使用してデプロイする場合、 --parametersオプションを使用してデプロイ時にパラメータ値を指定できます。

cdk deploy コマンド構造の例を次に示します。

```
$ cdk deploy stack-logical-id --parameters stack-name:parameter-name=parameter-value
```

CDK アプリケーションに 1 つのスタックが含まれている場合は、スタックの論理 ID 引数または --parametersオプション*stack-name*の値を指定する必要はありません。CDK CLIはこれらの値を自動的に検索して提供します。以下は、CDK アプリ内の単一スタックの uploadBucketNameパラメータuploadbucketの値を指定する例です。

```
$ cdk deploy --parameters uploadBucketName=uploadbucket
```

マルチスタックアプリケーションの cdk デプロイでパラメータ値を提供する

以下は、2 つの CDK スタックを含む TypeScript の CDK アプリケーションの例です。各スタックには、Amazon S3 バケットインスタンスと、Amazon S3 バケット名を設定するためのパラメータが含まれています。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

// Define the CDK app
const app = new cdk.App();

// First stack
export class MyFirstStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Set a default parameter name
    const bucketNameParam = new cdk.CfnParameter(this, 'bucketNameParam', {
      type: 'String',
      default: 'myfirststackdefaultbucketname'
```

```
    });

    // Define an S3 bucket
    new s3.Bucket(this, 'MyFirstBucket', {
      bucketName: bucketNameParam.valueAsString
    });
  }
}

// Second stack
export class MySecondStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Set a default parameter name
    const bucketNameParam = new cdk.CfnParameter(this, 'bucketNameParam', {
      type: 'String',
      default: 'mysecondstackdefaultbucketname'
    });

    // Define an S3 bucket
    new s3.Bucket(this, 'MySecondBucket', {
      bucketName: bucketNameParam.valueAsString
    });
  }
}

// Instantiate the stacks
new MyFirstStack(app, 'MyFirstStack', {
  stackName: 'MyFirstDeployedStack',
});

new MySecondStack(app, 'MySecondStack', {
  stackName: 'MySecondDeployedStack',
});
```

複数のスタックを含む CDK アプリケーションの場合、次のことができます。

- パラメータを使用して 1 つのスタックをデプロイする – マルチスタックアプリケーションから 1 つのスタックをデプロイするには、スタックの論理 ID を引数として指定します。

  以下は、 のパラメータ値mynewbucketnameとして MySecondStackをデプロイする例ですbucketNameParam。

```
$ cdk deploy MySecondStack --parameters bucketNameParam='mynewbucketname'
```

- すべてのスタックをデプロイし、各スタックのパラメータ値を指定する – '*'ワイルドカードまたはすべてのスタックをデプロイする--allオプションを指定します。--parameters オプションを 1 つのコマンドで複数回指定して、各スタックのパラメータ値を指定します。以下に例を示します。

```
$ cdk deploy '*' --
parameters MyFirstDeployedStack:bucketNameParam='mynewfirststackbucketname' --
parameters MySecondDeployedStack:bucketNameParam='mynewsecondstackbucketname'
```

- すべてのスタックをデプロイし、1 つのスタックのパラメータ値を指定する – '*'ワイルドカードまたはすべてのスタックをデプロイする--allオプションを指定します。次に、 --parameters オプションで のパラメータを定義するスタックを指定します。CDK アプリケーション内のすべてのスタックをデプロイし、MySecondDeployedStack AWS CloudFormation スタックのパラメータ値を指定する例を次に示します。他のすべてのスタックは、デフォルトのパラメータ値をデプロイして使用します。

```
$ cdk deploy '*' --parameters MySecondDeployedStack:bucketNameParam='mynewbucketname'
$ cdk deploy --all --
parameters MySecondDeployedStack:bucketNameParam='mynewbucketname'
```

ネストされたスタックを持つアプリケーションcdk deployに でパラメータ値を提供する

ネストされたスタックを含むアプリケーションを操作するときの CDK CLIの動作は、マルチスタックアプリケーションと似ています。主な違いは、ネストされたすべてのスタックをデプロイする場合は、 '**' ワイルドカードを使用することです。'*' ワイルドカードはすべてのスタックをデプロイしますが、ネストされたスタックはデプロイしません。'**' ワイルドカードは、ネストされたスタックを含むすべてのスタックをデプロイします。

以下は、ネストされたスタックのパラメータ値を指定しながら、ネストされたスタックをデプロイする例です。

```
$ cdk deploy '**' --parameters MultiStackCdkApp/
SecondStack:bucketNameParam='mysecondstackbucketname'
```

`cdk deploy` コマンドオプションの詳細については、「」を参照してくださいcdk deploy。

タグ付け

タグは、 AWS CDK アプリのコンストラクトに追加できる情報キーと値の要素です。特定のコンストラクトに適用されるタグは、そのタグ付け可能なすべての子にも適用されます。タグはアプリケーションから合成された AWS CloudFormation テンプレートに含まれ、デプロイする AWS リソースに適用されます。タグを使用して、以下の目的でリソースを識別および分類できます。

- 管理の簡素化
- コスト配分
- アクセスコントロール
- ユーザーが考案したその他の目的

> **Tip**
>
> AWS リソースでタグを使用する方法の詳細については、AWS ホワイトペーパーの「リソースのタグ付けのベストプラクティス AWS」を参照してください。

トピック

タグの使用

`Tags` クラスには静的メソッド が含まれておりof()、このメソッドを使用して、指定したコンストラクトにタグを追加したり、そこからタグを削除したりできます。

- `Tags.of(SCOPE).add()` は、指定されたコンストラクトとそのすべての子に新しいタグを適用します。

- `Tags.of(SCOPE).remove()` は、子コンストラクトがそれ自体に適用した可能性のあるタグを含め、特定のコンストラクトとその子コンストラクトからタグを削除します。

> **Note**
>
> タグ付けは を使用して実装されますthe section called “側面”。アスペクトとは、特定のスコープ内のすべてのコンストラクトにオペレーション (タグ付けなど) を適用する方法です。

次の例では、値 を持つタグキーを コンストラクトに適用します。

TypeScript

```
Tags.of(myConstruct).add('key', 'value');
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value');
```

Python

```
Tags.of(my_construct).add("key", "value")
```

Java

```
Tags.of(myConstruct).add("key", "value");
```

C#

```
Tags.Of(myConstruct).Add("key", "value");
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("key"), jsii.String("value"),
 &awscdk.TagProps{})
```

次の例では、 コンストラクトからタグキーを削除します。

TypeScript

```
Tags.of(myConstruct).remove('key');
```

JavaScript

```
Tags.of(myConstruct).remove('key');
```

Python

```
Tags.of(my_construct).remove("key")
```

Java

```
Tags.of(myConstruct).remove("key");
```

C#

```
Tags.Of(myConstruct).Remove("key");
```

Go

```
awscdk.Tags_Of(myConstruct).Remove(jsii.String("key"), &awscdk.TagProps{})
```

Stage コンストラクトを使用している場合は、 タグをStageレベル 以下に適用します。タグはStage境界を越えて適用されません。

タグの優先順位

はタグを再帰的に AWS CDK 適用および削除します。競合がある場合、優先順位が最も高いタグ付けオペレーションが優先されます。(優先度はオプションの priorityプロパティを使用して設定されます。) 2 つのオペレーションの優先順位が同じ場合、コンストラクトツリーの下部に最も近いタグ付けオペレーションが優先されます。デフォルトでは、タグの適用の優先度は 100 です (優先度が 50 の AWS CloudFormation リソースに直接追加されたタグを除く)。タグを削除するデフォルトの優先度は 200 です。

次に、優先度が 300 のタグをコンストラクトに適用します。

TypeScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

Python

```
Tags.of(my_construct).add("key", "value", priority=300)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
        .priority(300).build());
```

C#

```
Tags.Of(myConstruct).Add("key", "value", new TagProps { Priority = 300 });
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("key"), jsii.String("value"),
 &awscdk.TagProps{
  Priority: jsii.Number(300),
})
```

オプションのプロパティ

タグは[properties](#)、リソースへのタグの適用または削除方法を微調整するサポートを提供します。すべてのプロパティはオプションです。

`applyToLaunchedInstances` (Python: `apply_to_launched_instances`)

add() でのみ使用できます。デフォルトでは、タグは Auto Scaling グループで起動されたインスタンスに適用されます。Auto Scaling グループで起動されたインスタンスを無視するには、このプロパティを false に設定します。

`includeResourceTypes`/`excludeResourceTypes` (Python: `include_resource_types`/`exclude_resource_types`)

これらを使用して、リソースタイプに基づいて、 AWS CloudFormation リソースのサブセットでのみタグを操作します。デフォルトでは、 オペレーションはコンストラクトサブツリー内のすべてのリソースに適用されますが、これは特定のリソースタイプを含めるか除外するかによって変更できます。両方が指定されている場合、exclude は include よりも優先されます。

`priority`

これを使用して、他の `Tags.add()`および オペレーションに関するこの`Tags.remove()`オペレーションの優先度を設定します。値が大きいほど、小さい値よりも優先されます。デフォルトは、追加オペレーションの場合は 100 (AWS CloudFormation リソースに直接適用されるタグの場合は 50)、削除オペレーションの場合は 200 です。

次の例では、値 と優先度 100 のタグ tagname を コンストラクトの タイプのリソースに適用AWS::Xxx::Yyyします。タグは、Amazon EC2 Auto Scaling グループで起動されたインスタンスや タイプのリソースには適用されませんAWS::Xxx::Zzz。 (これらは 2 つの任意の異なる AWS CloudFormation リソースタイプのプレースホルダーです。)

TypeScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 100,
});
```

JavaScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
```

```
  priority: 100
});
```

Python

```
Tags.of(my_construct).add("tagname", "value",
    apply_to_launched_instances=False,
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=100)
```

Java

```
Tags.of(myConstruct).add("tagname", "value", TagProps.builder()
                .applyToLaunchedInstances(false)
                .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
                .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
                .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Add("tagname", "value", new TagProps
{
    ApplyToLaunchedInstances = false,
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("tagname"), jsii.String("value"),
 &awscdk.TagProps{
  ApplyToLaunchedInstances: jsii.Bool(false),
  IncludeResourceTypes:     &[]*string{jsii.String("AWS::Xxx:Yyy")},
  ExcludeResourceTypes:     &[]*string{jsii.String("AWS::Xxx:Zzz")},
  Priority:                 jsii.Number(100),
})
```

次の例では、優先順位が 200 のタグタグ名を コンストラクトAWS::Xxx::Yyyの 型のリソースから削除しますが、 型のリソースからは削除しませんAWS::Xxx::Zzz。

TypeScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200,
});
```

JavaScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200
});
```

Python

```
Tags.of(my_construct).remove("tagname",
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=200,)
```

Java

```
Tags.of((myConstruct).remove("tagname", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
        .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
        .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Remove("tagname", new TagProps
{
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

Go

```
awscdk.Tags_Of(myConstruct).Remove(jsii.String("tagname"), &awscdk.TagProps{
```

```
  IncludeResourceTypes: &[]*string{jsii.String("AWS::Xxx:Yyy")},
  ExcludeResourceTypes: &[]*string{jsii.String("AWS::Xxx:Zzz")},
  Priority:             jsii.Number(200),
})
```

例

次の例では、 というStack名前の 内で作成されたすべてのリソースTheBestに 値StackTypeを持つタグキーを追加しますMarketingSystem。その後、Amazon EC2 VPC サブネットを除くすべてのリソースから再度削除されます。その結果、タグが適用されたのはサブネットのみです。

TypeScript

```
import { App, Stack, Tags } from 'aws-cdk-lib';

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

JavaScript

```
const { App, Stack, Tags } = require('aws-cdk-lib');

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

Python

```
from aws_cdk import App, Stack, Tags

app = App();
the_best_stack = Stack(app, 'MarketingSystem')

# Add a tag to all constructs in the stack
Tags.of(the_best_stack).add("StackType", "TheBest")

# Remove the tag from all resources except subnet resources
Tags.of(the_best_stack).remove("StackType",
    exclude_resource_types=["AWS::EC2::Subnet"])
```

Java

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Tags;

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove("StackType", TagProps.builder()
        .excludeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
using Amazon.CDK;

var app = new App();
var theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.Of(theBestStack).Add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.Of(theBestStack).Remove("StackType", new TagProps
{
    ExcludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

Go

```
import "github.com/aws/aws-cdk-go/awscdk/v2"
app := awscdk.NewApp(nil)
theBestStack := awscdk.NewStack(app, jsii.String("MarketingSystem"),
 &awscdk.StackProps{})

// Add a tag to all constructs in the stack
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{})

// Remove the tag from all resources except subnet resources
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{
  ExcludeResourceTypes: &[]*string{jsii.String("AWS::EC2::Subnet")},
})
```

次のコードでも同じ結果が得られます。どのアプローチ (包含または除外) でインテントがより明確になるかを検討してください。

TypeScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

JavaScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

Python

```
Tags.of(the_best_stack).add("StackType", "TheBest",
    include_resource_types=["AWS::EC2::Subnet"])
```

Java

```
Tags.of(theBestStack).add("StackType", "TheBest", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
Tags.Of(theBestStack).Add("StackType", "TheBest", new TagProps {
    IncludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

Go

```
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{
  IncludeResourceTypes: &[]*string{jsii.String("AWS::EC2::Subnet")},
})
```

単一コンストラクトのタグ付け

`Tags.of(scope).add(key, value)` は、 のコンストラクトにタグを追加する標準的な方法です AWS CDK。そのツリーウォーキング動作は、指定されたスコープ内のすべてのタグ付け可能なリソースに再帰的にタグ付けされ、ほとんどの場合、必要な動作です。ただし、特定の任意のコンストラクト (またはコンストラクト) にタグを付ける必要がある場合があります。

このようなケースの 1 つは、タグ付けされるコンストラクトのプロパティから値が導出されるタグの適用です。標準のタグ付けアプローチは、スコープ内のすべての一致するリソースに同じキーと値を再帰的に適用します。ただし、ここでは、タグ付けされたコンストラクトごとに値が異なる場合があります。

タグは側面 を使用して実装され、CDK は を使用して指定したスコープで各コンストラクトのタグの `visit()`メソッドを呼び出しますTags.of(scope)。Tag.visit() を直接呼び出して、タグを 1 つのコンストラクトに適用できます。

TypeScript

```
new cdk.Tag(key, value).visit(scope);
```

JavaScript

```
new cdk.Tag(key, value).visit(scope);
```

Python

```
cdk.Tag(key, value).visit(scope)
```

Java

```
Tag.Builder.create(key, value).build().visit(scope);
```

C#

```
new Tag(key, value).Visit(scope);
```

Go

```
awscdk.NewTag(key, value, &awscdk.TagProps{}).Visit(scope)
```

スコープ内のすべてのコンストラクトにタグを付けることはできますが、タグの値は各コンストラクトのプロパティから派生させることができます。これを行うには、前の例に示すように、アスペクトを記述し、アスペクトの `visit()`メソッドに タグを適用します。次に、 を使用して目的のスコープに アスペクトを追加します`Aspects.of(scope).add(aspect)`。

次の例では、リソースのパスを含むスタック内の各リソースにタグを適用します。

TypeScript

```
class PathTagger implements cdk.IAspect {
  visit(node: IConstruct) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

JavaScript

```
class PathTagger {
  visit(node) {
```

```
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

Python

```
@jsii.implements(cdk.IAspect)
class PathTagger:
    def visit(self, node: IConstruct):
        cdk.Tag("aws-cdk-path", node.node.path).visit(node)

stack = MyStack(app)
cdk.Aspects.of(stack).add(PathTagger())
```

Java

```
final class PathTagger implements IAspect {
 public void visit(IConstruct node) {
  Tag.Builder.create("aws-cdk-path", node.getNode().getPath()).build().visit(node);
 }
}

stack stack = new MyStack(app);
Aspects.of(stack).add(new PathTagger());
```

C#

```
public class PathTagger : IAspect
{
    public void Visit(IConstruct node)
    {
        new Tag("aws-cdk-path", node.Node.Path).Visit(node);
    }
}

var stack = new MyStack(app);
Aspects.Of(stack).Add(new PathTagger);
```

> **Tip**
>
> 優先順位、リソースタイプなどの条件付きタグ付けのロジックは、 Tag クラスに組み込まれています。これらの機能は、任意のリソースにタグを適用するときに使用できます。条件が満たされない場合、タグは適用されません。また、 Tag クラスはタグ付け可能なリソースのみをタグ付けするため、タグを適用する前にコンストラクトがタグ付け可能かどうかをテストする必要はありません。

アセット

アセットは、 AWS CDK ライブラリやアプリケーションにバンドルできるローカルファイル、ディレクトリ、または Docker イメージです。例えば、アセットは AWS Lambda 関数のハンドラーコードを含むディレクトリである可能性があります。アセットは、アプリが動作するために必要なアーティファクトを表すことができます。

次のチュートリアルビデオでは、CDK アセットの包括的な概要と、コードとしてのインフラストラクチャ (IaC) でそれらを使用する方法について説明します。

CDK アセットの説明

特定の AWS コンストラクトによって公開される APIs を使用してアセットを追加します。例えば、lambda.Function コンストラクトを定義すると、 コードプロパティを使用してアセット (ディレクトリ) を渡すことができます。 Functionはアセットを使用してディレクトリの内容をバンドルし、それを関数のコードとして使用します。同様に、ecs.ContainerImage.fromAsset は、Amazon ECS タスク定義を定義するときに、ローカルディレクトリから構築された Docker イメージを使用します。

アセットの詳細

アプリでアセットを参照すると、アプリケーションから合成されたクラウドアセンブリには、 AWS CDK CLI の手順を含むメタデータ情報が含まれます。手順には、ローカルディスク上のアセットの場所と、圧縮するディレクトリ (zip) や構築する Docker イメージなど、アセットタイプに基づいて実行するバンドルのタイプが含まれます。

はアセットの出典ハッシュ AWS CDK を生成します。これは、アセットの内容が変更されたかどうかを判断するために、構築時に使用できます。

デフォルトでは、 はクラウドアセンブリディレクトリにアセットのコピー AWS CDK を作成します。これはデフォルトで、出典ハッシュの`cdk.out`下の になります。このようにして、クラウドアセンブリは自己完結型であるため、デプロイのために別のホストに移動した場合でもデプロイできます。詳細については、「the section called “クラウドアセンブリ”」を参照してください。

がアセットを参照するアプリを (アプリコードから直接、またはライブラリを介して) AWS CDK デプロイすると、 AWS CDK CLI はまずアセットを準備して Amazon S3 バケットまたは Amazon ECR リポジトリに公開します。(S3 バケットまたはリポジトリはブートストラップ中に作成されます) 。スタックで定義されたリソースがデプロイされるのは、その後のみです。

このセクションでは、フレームワークで使用できる低レベル APIs について説明します。

アセットタイプ

では、次のタイプのアセット AWS CDK がサポートされています。

Amazon S3 アセット

これらは、 が Amazon S3 AWS CDK にアップロードするローカルファイルとディレクトリです。

Docker イメージ

これらは、 が Amazon ECR AWS CDK にアップロードする Docker イメージです。

これらのアセットタイプについては、以降のセクションで説明します。

Amazon S3 アセット

ローカルファイルとディレクトリをアセットとして定義し、 AWS CDK パッケージを作成して aws-s3-assets モジュールを介して Amazon S3 にアップロードできます。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_s3_assets-readme.html

次の例では、ローカルディレクトリアセットとファイルアセットを定義します。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
```

```
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

Python

```
import os.path
dirname = os.path.dirname(__file__)

from aws_cdk.aws_s3_assets import Asset

# Archived and uploaded to Amazon S3 as a .zip file
directory_asset = Asset(self, "SampleZippedDirAsset",
  path=os.path.join(dirname, "sample-asset-directory")
)

# Uploaded to Amazon S3 as-is
file_asset = Asset(self, 'SampleSingleFileAsset',
  path=os.path.join(dirname, 'file-asset.txt')
)
```

Java

```
import java.io.File;
```

```
import software.amazon.awscdk.services.s3.assets.Asset;

// Directory where app was started
File startDir = new File(System.getProperty("user.dir"));

// Archived and uploaded to Amazon S3 as a .zip file
Asset directoryAsset = Asset.Builder.create(this, "SampleZippedDirAsset")
                .path(new File(startDir, "sample-asset-
directory").toString()).build();

// Uploaded to Amazon S3 as-is
Asset fileAsset = Asset.Builder.create(this, "SampleSingleFileAsset")
                .path(new File(startDir, "file-asset.txt").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.S3.Assets;

// Archived and uploaded to Amazon S3 as a .zip file
var directoryAsset = new Asset(this, "SampleZippedDirAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
var fileAsset = new Asset(this, "SampleSingleFileAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "file-asset.txt")
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

awss3assets.NewAsset(stack, jsii.String("SampleZippedDirAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "sample-asset-directory")),
})
```

```
awss3assets.NewAsset(stack, jsii.String("SampleSingleFileAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "file-asset.txt")),
})
```

ほとんどの場合、aws-s3-assetsモジュールで APIsを直接使用する必要はありません。などのアセットをサポートするモジュールaws-lambdaには、アセットを使用できるように便利な方法があります。Lambda 関数の場合、fromAsset () 静的メソッドを使用すると、ローカルファイルシステム内のディレクトリまたは .zip ファイルを指定できます。

Lambda 関数の例

一般的なユースケースは、ハンドラーコードを Amazon S3 アセットとして Lambda 関数を作成することです。

次の例では、Amazon S3 アセットを使用してローカルディレクトリ に Python ハンドラーを定義しますhandler。また、ローカルディレクトリアセットを codeプロパティとして Lambda 関数を作成します。ハンドラーの Python コードを次に示します。

```
def lambda_handler(event, context):
  message = 'Hello World!'
  return {
    'message': message
  }
```

メイン AWS CDK アプリのコードは次のようになります。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Constructs } from 'constructs';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as path from 'path';

export class HelloAssetStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
```

```
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const path = require('path');

class HelloAssetStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}

module.exports = { HelloAssetStack }
```

Python

```
from aws_cdk import Stack
from constructs import Construct
from aws_cdk import aws_lambda as lambda_

import os.path
dirname = os.path.dirname(__file__)

class HelloAssetStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        lambda_.Function(self, 'myLambdaFunction',
            code=lambda_.Code.from_asset(os.path.join(dirname, 'handler')),
            runtime=lambda_.Runtime.PYTHON_3_6,
            handler="index.lambda_handler")
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class HelloAssetStack extends Stack {

    public HelloAssetStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public HelloAssetStack(final App scope, final String id, final StackProps props)
 {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler").build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using System.IO;

public class HelloAssetStack : Stack
{
    public HelloAssetStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        new Function(this, "myLambdaFunction", new FunctionProps
        {
            Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(),
 "handler")),
```

```
            Runtime = Runtime.PYTHON_3_6,
            Handler = "index.lambda_handler"
        });
    }
}
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

func HelloAssetStack(scope constructs.Construct, id string, props
 *HelloAssetStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  dirName, err := os.Getwd()
  if err != nil {
    panic(err)
  }

  awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
    Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler")),
 &awss3assets.AssetOptions{}),
    Runtime: awslambda.Runtime_PYTHON_3_6(),
    Handler: jsii.String("index.lambda_handler"),
  })

  return stack
}
```

Function メソッドは、アセットを使用してディレクトリの内容をバンドルし、関数のコードに使用します。

> **Tip**
>
> Java .jarファイルは、異なる拡張子を持つ ZIP ファイルです。これらは Amazon S3 にそのままアップロードされますが、Lambda 関数としてデプロイされると、含まれるファイルが抽出されますが、これは不要な場合があります。これを回避するには、.jarファイルをディレクトリに配置し、そのディレクトリをアセットとして指定します。

デプロイ時属性の例

Amazon S3 アセットタイプは、 AWS CDK ライブラリやアプリケーションで参照できるデプロイ時属性も公開します。 AWS CDK CLI コマンドは、アセットプロパティを AWS CloudFormation パラメータとしてcdk synth表示します。

次の例では、デプロイ時属性を使用して、イメージアセットの場所を環境変数として Lambda 関数に渡します。(ファイルの種類は関係ありません。ここで使用する PNG イメージは一例にすぎません。)

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

Python

```
import os.path

import aws_cdk.aws_lambda as lambda_
from aws_cdk.aws_s3_assets import Asset

dirname = os.path.dirname(__file__)

image_asset = Asset(self, "SampleAsset",
    path=os.path.join(dirname, "images/my-image.png"))

lambda_.Function(self, "myLambdaFunction",
    code=lambda_.Code.asset(os.path.join(dirname, "handler")),
    runtime=lambda_.Runtime.PYTHON_3_6,
    handler="index.lambda_handler",
    environment=dict(
        S3_BUCKET_NAME=image_asset.s3_bucket_name,
        S3_OBJECT_KEY=image_asset.s3_object_key,
        S3_OBJECT_URL=image_asset.s3_object_url))
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.assets.Asset;

public class FunctionStack extends Stack {
    public FunctionStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset imageAsset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build())

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler")
                .environment(java.util.Map.of(    // Java 9 or later
                    "S3_BUCKET_NAME", imageAsset.getS3BucketName(),
                    "S3_OBJECT_KEY", imageAsset.getS3ObjectKey(),
                    "S3_OBJECT_URL", imageAsset.getS3ObjectUrl()))
                .build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png")
});
```

```
new Function(this, "myLambdaFunction", new FunctionProps
{
    Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), "handler")),
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionary<string, string>
    {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_OBJECT_URL"] = imageAsset.S3ObjectUrl
    }
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

imageAsset := awss3assets.NewAsset(stack, jsii.String("SampleAsset"),
 &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "images/my-image.png")),
})

awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 &awslambda.FunctionProps{
  Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler"))),
  Runtime: awslambda.Runtime_PYTHON_3_6(),
  Handler: jsii.String("index.lambda_handler"),
  Environment: &map[string]*string{
    "S3_BUCKET_NAME": imageAsset.S3BucketName(),
    "S3_OBJECT_KEY": imageAsset.S3ObjectKey(),
```

```
    "S3_URL": imageAsset.S3ObjectUrl(),
  },
})
```

アクセス許可

aws-s3-assets モジュール、IAM ロール、ユーザー、またはグループを通じて Amazon S3 アセットを直接使用し、ランタイムにアセットを読み取る必要がある場合は、set.grantRead メソッドを通じてそれらのアセットに IAM アクセス許可を付与します。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_s3_assets-readme.html

次の例では、ファイルアセットに対する読み取りアクセス許可を IAM グループに付与します。

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

Python

```
from aws_cdk.aws_s3_assets import Asset
import aws_cdk.aws_iam as iam
```

```
import os.path
dirname = os.path.dirname(__file__)

        asset = Asset(self, "MyFile",
            path=os.path.join(dirname, "my-image.png"))

        group = iam.Group(self, "MyUserGroup")
        asset.grant_read(group)
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset asset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build();

        Group group = new Group(this, "MyUserGroup");
        asset.grantRead(group);   }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @"images\my-
image.png"))
});
```

```
var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsiam"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awss3assets.NewAsset(stack, jsii.String("MyFile"), &awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "my-image.png")),
})

group := awsiam.NewGroup(stack, jsii.String("MyUserGroup"), &awsiam.GroupProps{})

asset.GrantRead(group)
```

Docker イメージアセット

AWS CDK では、ローカル Docker イメージを aws-ecr-assetsモジュールを介してアセットとしてバンドルできます。

次の例では、ローカルに構築され、Amazon ECR にプッシュされる Docker イメージを定義します。イメージはローカル Docker コンテキストディレクトリ (Dockerfile を使用) から構築され、AWS CDK CLI またはアプリケーションの CI/CD パイプラインによって Amazon ECR にアップロードされます。イメージは AWS CDK アプリで自然に参照できます。

TypeScript

```
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';
```

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

Python

```
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))
```

Java

```
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

my-image ディレクトリには Dockerfile が含まれている必要があります。 AWS CDK CLI は から Docker イメージを構築しmy-image、Amazon ECR リポジトリにプッシュして、 スタックの パラメータとして AWS CloudFormation リポジトリの名前を指定します。Docker イメージアセットタイプは、 AWS CDK ライブラリやアプリケーションで参照できるデプロイ時属性を公開します。 AWS CDK CLI コマンドは、アセットプロパティを AWS CloudFormation パラメータとしてcdk synth表示します。

Amazon ECS タスク定義の例

一般的なユースケースは、Docker コンテナを実行する Amazon ECS を作成するTaskDefinitionことです。次の例では、 がローカルで AWS CDK ビルドして Amazon ECR にプッシュする Docker イメージアセットの場所を指定します。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecr_assets from 'aws-cdk-lib/aws-ecr-assets';
import * as path from 'path';

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
```

```
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const ecr_assets = require('aws-cdk-lib/aws-ecr-assets');
const path = require('path');

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

Python

```
import aws_cdk.aws_ecs as ecs
import aws_cdk.aws_ecr_assets as ecr_assets

import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

asset = ecr_assets.DockerImageAsset(self, 'MyBuildImage',
```

```
    directory=os.path.join(dirname, 'my-image'))

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_docker_image_asset(asset))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder()
            .image(ContainerImage.fromDockerImageAsset(asset))
            .build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.Ecr.Assets;

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
```

```
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromDockerImageAsset(asset)
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

taskDefinition := awsecs.NewTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.TaskDefinitionProps{
  MemoryMiB: jsii.String("1024"),
  Cpu: jsii.String("512"),
})

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromDockerImageAsset(asset),
})
```

デプロイ時属性の例

次の例は、デプロイ時属性 `repository`および を使用して`imageUri`、 AWS Fargate 起動タイプで Amazon ECS タスク定義を作成する方法を示しています。Amazon ECR リポジトリルックアップには URI ではなくイメージのタグが必要なため、アセットの URI の末尾からスニップすることに注意してください。

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
```

```
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

Python

```
import aws_cdk.aws_ecs as ecs
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'my-image',
    directory=os.path.join(dirname, "..", "demo-image"))

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024, cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_ecr_repository(
        asset.repository, asset.image_uri.rpartition(":")[-1]))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
            .directory(new File(startDir, "demo-image").toString()).build();

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset's image URI for use in ECR repo lookup
String imageUri = asset.getImageUri();
String imageTag = imageUri.substring(imageUri.lastIndexOf(":") + 1);
```

```
taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
                asset.getRepository(), imageTag)).build());
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "demo-image")
});

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromEcrRepository(asset.Repository,
 asset.ImageUri.Split(":").Last())
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}
```

```
asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "demo-image")),
})

taskDefinition := awsecs.NewFargateTaskDefinition(stack, jsii.String("TaskDef"),
 &awsecs.FargateTaskDefinitionProps{
  MemoryLimitMiB: jsii.Number(1024),
  Cpu: jsii.Number(512),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 &awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromEcrRepository(asset.Repository(),
 asset.ImageTag()),
})
```

ビルド引数の例

デプロイ中に AWS CDK CLI がイメージをビルドするときに、 `buildArgs` (Python: `build_args`) プロパティオプションを使用して、Docker ビルドステップのカスタマイズされたビルド引数を指定できます。

TypeScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

JavaScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

Python

```
asset = DockerImageAsset(self, "MyBulidImage",
    directory=os.path.join(dirname, "my-image"),
    build_args=dict(HTTP_PROXY="http://10.20.30.2:1234"))
```

Java

```
DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image"),
            .directory(new File(startDir, "my-image").toString())
            .buildArgs(java.util.Map.of(    // Java 9 or later
                "HTTP_PROXY", "http://10.20.30.2:1234"))
            .build();
```

C#

```
var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image"),
    BuildArgs = new Dictionary<string, string>
    {
        ["HTTP_PROXY"] = "http://10.20.30.2:1234"
    }
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 &awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
  BuildArgs: &map[string]*string{
    "HTTP_PROXY": jsii.String("http://10.20.30.2:1234"),
  },
})
```

アクセス許可

aws-ecs などの Docker イメージアセットをサポートするモジュールを使用する場合、アセットを直接または を介して使用すると、 がアクセス許可 AWS CDK を管理しますContainerImage。fromEcrRepository (Python: `from_ecr_repository`) 。Docker イメージアセットを直接使用する場合は、コンシューマープリンシパルにイメージをプルするアクセス許可があることを確認してください。

ほとんどの場合、set.repository.grantPull メソッド (Python:) を使用する必要がありますgrant_pull。 これにより、プリンシパルの IAM ポリシーが変更され、このリポジトリからイメージをプルできるようになります。 イメージをプルするプリンシパルが同じアカウントにない場合、またはアカウントでロールを引き受ける AWS サービス (など AWS CodeBuild) である場合は、プリンシパルのポリシーではなく、リソースポリシーでプル許可を付与する必要があります。asset.repository.addToResourcePolicy メソッド (Python: `add_to_resource_policy`) を使用して、適切なプリンシパル許可を付与します。

AWS CloudFormation リソースメタデータ

> **Note**
>
> このセクションは、コンストラクト作成者にのみ関係します。状況によっては、ツールで特定の CFN リソースがローカルアセットを使用していることを知っている必要があります。例えば、 AWS SAM CLI を使用して、デバッグ目的で Lambda 関数をローカルで呼び出すことができます。詳細については、「the section called “AWS SAM 統合”」を参照してください。

このようなユースケースを有効にするために、外部ツールは AWS CloudFormation リソース上のメタデータエントリのセットを参照します。

- `aws:asset:path` - アセットのローカルパスを指します。
- `aws:asset:property` – アセットが使用されるリソースプロパティの名前。

これら 2 つのメタデータエントリを使用すると、ツールはアセットが特定のリソースによって使用されていることを確認し、高度なローカルエクスペリエンスを実現できます。

これらのメタデータエントリをリソースに追加するには、 `asset.addResourceMetadata` (Python: `add_resource_metadata`) メソッドを使用します。

アクセス許可

AWS Construct Library は、アクセスとアクセス許可を管理するために、一般的に広く実装されているいくつかのイディオムを使用します。IAM モジュールは、これらのイディオムを使用するために必要なツールを提供します。

AWS CDK は を使用して変更 AWS CloudFormation をデプロイします。すべてのデプロイには、AWS CloudFormation デプロイを開始するアクター (デベロッパーまたは自動システム) が含まれます。これを行う過程で、アクターは 1 つ以上の IAM ID (ユーザーまたはロール) を引き受け、オプションでロールを に渡します AWS CloudFormation。

AWS IAM Identity Center を使用してユーザーとして認証する場合、シングルサインオンプロバイダーは、事前定義された IAM ロールとして行動することを許可する有効期間の短いセッション認証情報を提供します。が IAM Identity Center 認証から AWS 認証情報 AWS CDK を取得する方法については、「 SDK およびツールリファレンスガイド」の「IAM Identity Center 認証を理解する」を参照してください。 AWS SDKs

プリンシパル

IAM プリンシパルは、API を呼び出すことができるユーザー、サービス、またはアプリケーションを表す認証された AWS エンティティです。 AWS APIs AWS Construct Library では、プリンシパルに AWS リソースへのアクセスを許可する複数の柔軟な方法でプリンシパルを指定できます。

セキュリティコンテキストでは、「プリンシパル」という用語は、特にユーザーなどの認証されたエンティティを指します。グループやロールなどのオブジェクトは、ユーザー (およびその他の認証されたエンティティ) を表すのではなく、アクセス許可を付与する目的で間接的に識別します。

例えば、IAM グループを作成する場合、グループ (およびそのメンバー) に Amazon RDS テーブルへの書き込みアクセスを許可できます。ただし、グループ自体は単一のエンティティを表していないため (グループにログインすることもできない) 、プリンシパルではありません。

CDK の IAM ライブラリでは、プリンシパルを直接または間接的に識別するクラスが`IPrincipal`インターフェイスを実装し、これらのオブジェクトをアクセスポリシーで同じ意味で使用できるようにします。ただし、セキュリティ上の意味では、これらすべてがプリンシパルというわけではありません。これらのオブジェクトには以下が含まれます。

1. `Role`、、 `User`などの IAM リソース `Group`
2. サービスプリンシパル (`new iam.ServicePrincipal('service.amazonaws.com')`)

3. フェデレーティッドプリンシパル (new iam.FederatedPrincipal('cognito-identity.amazonaws.com'))
4. アカウントプリンシパル (new iam.AccountPrincipal('0123456789012'))
5. 正規ユーザープリンシパル (new iam.CanonicalUserPrincipal('79a59d[...]7ef2be'))
6. AWS Organizations プリンシパル (new iam.OrganizationPrincipal('org-id'))
7. 任意の ARN プリンシパル (new iam.ArnPrincipal(res.arn))
8. 複数のプリンシパルを信頼iam.CompositePrincipal(principal1, principal2, ...)する

権限

Amazon S3 バケットや Amazon DynamoDB テーブルなど、アクセスできるリソースを表すすべてのコンストラクトには、別のエンティティへのアクセスを許可するメソッドがあります。このようなメソッドには、すべて grant で始まる名前が付いています。

例えば、Amazon S3 バケットには、エンティティからバケットへの読み取りおよび読み取り/書き込みアクセスをそれぞれ有効にするメソッド grantReadと grantReadWrite (Python: grant_read、grant_read_write) があります。エンティティは、これらのオペレーションを実行するために必要な Amazon S3 IAM アクセス許可を正確に把握する必要はありません。

グラントメソッドの最初の引数は、常にタイプ IGrantable です。このインターフェイスは、アクセス許可を付与できるエンティティを表します。つまり、IAM オブジェクト 、Role、 Userなどのロールを持つリソースを表しますGroup。

他のエンティティにもアクセス許可を付与できます。例えば、このトピックの後半で、Amazon S3 バケットへのアクセス権をプロジェクトに付与 CodeBuildする方法を示します。通常、関連付けられたロールは、アクセス権が付与されているエンティティの roleプロパティを介して取得されます。

などの実行ロールを使用するリソースlambda.Functionも を実装IGrantableするため、ロールへのアクセス権を付与する代わりに、直接アクセス権を付与できます。例えば、 bucketが Amazon S3 バケットで、 functionが Lambda 関数の場合、次のコードは関数にバケットへの読み取りアクセスを許可します。

TypeScript

```
bucket.grantRead(function);
```

JavaScript

```
bucket.grantRead(function);
```

Python

```
bucket.grant_read(function)
```

Java

```
bucket.grantRead(function);
```

C#

```
bucket.GrantRead(function);
```

スタックのデプロイ中にアクセス許可を適用する必要がある場合があります。このようなケースの 1 つは、 AWS CloudFormation カスタムリソースに他のリソースへのアクセスを許可する場合です。カスタムリソースはデプロイ時に呼び出されるため、デプロイ時に指定されたアクセス許可が必要です。

もう 1 つのケースは、サービスが渡すロールに適切なポリシーが適用されていることを確認する場合です。(ポリシーの設定を忘れないようにするために、多くの AWS のサービスでこれを行っています)。このような場合、アクセス許可の適用が遅すぎると、デプロイが失敗する可能性があります。

別のリソースが作成される前にグラントのアクセス許可を強制的に適用するには、次に示すように、グラント自体に依存関係を追加できます。グラントメソッドの戻り値は一般的に破棄されますが、実際にはすべてのグラントメソッドが `iam.Grant` オブジェクトを返します。

TypeScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

JavaScript

```
const grant = bucket.grantRead(lambda);
```

```
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

Python

```
grant = bucket.grant_read(function)
custom = CustomResource(...)
custom.node.add_dependency(grant)
```

Java

```
Grant grant = bucket.grantRead(function);
CustomResource custom = new CustomResource(...);
custom.node.addDependency(grant);
```

C#

```
var grant = bucket.GrantRead(function);
var custom = new CustomResource(...);
custom.node.AddDependency(grant);
```

ロール

IAM パッケージには、IAM ロールを表すRoleコンストラクトが含まれています。次のコードは、Amazon EC2 サービスを信頼する新しいロールを作成します。

TypeScript

```
import * as iam from 'aws-cdk-lib/aws-iam';

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

JavaScript

```
const iam = require('aws-cdk-lib/aws-iam');

const role = new iam.Role(this, 'Role', {
```

```
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com')   // required
});
```

Python

```
import aws_cdk.aws_iam as iam

role = iam.Role(self, "Role",
          assumed_by=iam.ServicePrincipal("ec2.amazonaws.com")) # required
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.iam.ServicePrincipal;

Role role = Role.Builder.create(this, "Role")
        .assumedBy(new ServicePrincipal("ec2.amazonaws.com")).build();
```

C#

```
using Amazon.CDK.AWS.IAM;

var role = new Role(this, "Role", new RoleProps
{
    AssumedBy = new ServicePrincipal("ec2.amazonaws.com"),   // required
});
```

ロールの addToPolicyメソッド (Python: add_to_policy) を呼び出し、追加するルールPolicyStatementを定義する を渡すことで、ロールにアクセス許可を追加できます。ステートメントがロールのデフォルトポリシーに追加されます。ポリシーがない場合は、作成されます。

次の例では、承認されたサービスが であるという条件で、 リソース ec2:SomeAction および otherRole (Python: other_role) s3:AnotherActionに対するアクション bucket および のロールにDenyポリシーステートメントを追加します AWS CodeBuild。

TypeScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
```

```
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com',
}}}));
```

JavaScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com'
}}}));
```

Python

```
role.add_to_policy(iam.PolicyStatement(
    effect=iam.Effect.DENY,
    resources=[bucket.bucket_arn, other_role.role_arn],
    actions=["ec2:SomeAction", "s3:AnotherAction"],
    conditions={"StringEquals": {
        "ec2:AuthorizedService": "codebuild.amazonaws.com"}}
))
```

Java

```
role.addToPolicy(PolicyStatement.Builder.create()
        .effect(Effect.DENY)
        .resources(Arrays.asList(bucket.getBucketArn(), otherRole.getRoleArn()))
        .actions(Arrays.asList("ec2:SomeAction", "s3:AnotherAction"))
        .conditions(java.util.Map.of(    // Map.of requires Java 9 or later
            "StringEquals", java.util.Map.of(
                "ec2:AuthorizedService", "codebuild.amazonaws.com")))
        .build());
```

C#

```
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.DENY,
    Resources = new string[] { bucket.BucketArn, otherRole.RoleArn },
```

```
    Actions = new string[] { "ec2:SomeAction", "s3:AnotherAction" },
    Conditions = new Dictionary<string, object>
    {
        ["StringEquals"] = new Dictionary<string, string>
        {
            ["ec2:AuthorizedService"] = "codebuild.amazonaws.com"
        }
    }
}));
```

前の例では、 addToPolicy (Python: add_to_policy) 呼び出しを使用して新しいPolicyStatementインラインを作成しました。既存のポリシーステートメントまたは変更したポリシーステートメントを渡すこともできます。PolicyStatement オブジェクトには、プリンシパル、リソース、条件、およびアクションを追加するための多数のメソッドがあります。

ロールが正しく機能する必要があるコンストラクトを使用している場合は、次のいずれかを実行できます。

- コンストラクトオブジェクトをインスタンス化するときに、既存のロールを渡します。
- 適切なサービスプリンシパルを信頼して、コンストラクトに新しいロールを作成させます。次の例では、このようなコンストラクトを使用します: CodeBuild プロジェクト。

TypeScript

```
import * as codebuild from 'aws-cdk-lib/aws-codebuild';

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole: iam.IRole | undefined = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole,
});
```

JavaScript

```
const codebuild = require('aws-cdk-lib/aws-codebuild');
```

```
// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole
});
```

Python

```
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
# if role is None, the Project creates a new default role,
# trusting the codebuild.amazonaws.com service principal
role=some_role)
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
        .role(someRole).build();
```

C#

```
using Amazon.CDK.AWS.CodeBuild;

// imagine roleOrNull is a function that might return a Role object
```

```
// under some conditions, and null under other conditions
var someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
var project = new Project(this, "Project", new ProjectProps
{
    Role = someRole
});
```

オブジェクトが作成されると、ロール (ロールが渡されたか、コンストラクトによって作成されたデフォルトのロールかにかかわらず) はプロパティ として使用できますrole。ただし、このプロパティは外部リソースでは使用できません。したがって、これらのコンストラクトには addToRolePolicy (Python: add_to_role_policy) メソッドがあります。

コンストラクトが外部リソースである場合、 メソッドは何もせず、それ以外の場合は roleプロパティの addToPolicy (Python: add_to_policy) メソッドを呼び出します。これにより、未定義のケースを明示的に処理する手間が省けます。

次の例は、以下を示しています。

TypeScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,   // ... and so on defining the policy
}));
```

JavaScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW   // ... and so on defining the policy
}));
```

Python

```
# project is imported into the CDK application
project = codebuild.Project.from_project_name(self, 'Project', 'ProjectName')

# project is imported, so project.role is undefined, and this call has no effect
project.add_to_role_policy(iam.PolicyStatement(
  effect=iam.Effect.ALLOW,   # ... and so on defining the policy
)
```

Java

```
// project is imported into the CDK application
Project project = Project.fromProjectName(this, "Project", "ProjectName");

// project is imported, so project.getRole() is null, and this call has no effect
project.addToRolePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)   // .. and so on defining the policy
        .build());
```

C#

```
// project is imported into the CDK application
var project = Project.FromProjectName(this, "Project", "ProjectName");

// project is imported, so project.role is null, and this call has no effect
project.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW, // ... and so on defining the policy
}));
```

リソースポリシー

Amazon S3 バケットや IAM ロール AWSなど、 の一部のリソースにもリソースポリシーがあります。これらのコンストラクトには addToResourcePolicyメソッド (Python: add_to_resource_policy) があり、引数PolicyStatementとして を受け取ります。リソースポリシーに追加されるすべてのポリシーステートメントは、少なくとも 1 つのプリンシパルを指定する必要があります。

次の例では、Amazon S3 バケットbucketはそれ自体に アクセスs3:SomeAction許可を持つロールを付与します。

TypeScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

JavaScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

Python

```
bucket.add_to_resource_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["s3:SomeAction"],
    resources=[bucket.bucket_arn],
    principals=role))
```

Java

```
bucket.addToResourcePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)
        .actions(Arrays.asList("s3:SomeAction"))
        .resources(Arrays.asList(bucket.getBucketArn()))
        .principals(Arrays.asList(role))
        .build());
```

C#

```
bucket.AddToResourcePolicy(new PolicyStatement(new PolicyStatementProps
```

```
{
    Effect = Effect.ALLOW,
    Actions = new string[] { "s3:SomeAction" },
    Resources = new string[] { bucket.BucketArn },
    Principals = new IPrincipal[] { role }
}));
```

外部 IAM オブジェクトの使用

AWS CDK アプリ外で IAM ユーザー、プリンシパル、グループ、またはロールを定義している場合は、 AWS CDK アプリでその IAM オブジェクトを使用できます。そのためには、ARN またはその名前を使用して参照を作成します。(ユーザー、グループ、ロールの名前を使用します) 。その後、返されたリファレンスを使用して、前述のようにアクセス許可を付与したり、ポリシーステートメントを作成したりできます。

- ユーザーの場合は、 `User.fromUserArn()`または を呼び出します`User.fromUserName()`。`User.fromUserAttributes()` も利用できますが、現在は と同じ機能を提供していますUser.fromUserArn()。
- プリンシパルの場合は、`ArnPrincipal`オブジェクトをインスタンス化します。
- グループの場合は、 `Group.fromGroupArn()`または を呼び出しますGroup.fromGroupName()。
- ロールの場合は、 `Role.fromRoleArn()`または を呼び出しますRole.fromRoleName()。

ポリシー (マネージドポリシーを含む) は、次の方法を使用して同様の方法で使用できます。これらのオブジェクトへの参照は、IAM ポリシーが必要な任意の場所で使用できます。

- `Policy.fromPolicyName`
- `ManagedPolicy.fromManagedPolicyArn`
- `ManagedPolicy.fromManagedPolicyName`
- `ManagedPolicy.fromAwsManagedPolicyName`

> **Note**
>
> 外部 AWS リソースへのすべての参照と同様に、CDK アプリで外部 IAM オブジェクトを変更することはできません。

ランタイムのコンテキスト

コンテキスト値は、アプリ、スタック、またはコンストラクトに関連付けることのできるキーと値のペアです。これらは、 ファイル (通常はプロジェクトディレクトリ`cdk.context.json`の`cdk.json`または) またはコマンドラインからアプリケーションに提供されます。

CDK Toolkit は、コンテキストを使用して、合成中に AWS アカウントから取得した値をキャッシュします。値には、アカウントのアベイラビリティーゾーン、または Amazon EC2 インスタンスで現在利用可能な Amazon マシンイメージ (AMI) IDs が含まれます。これらの値は AWS アカウントによって提供されるため、CDK アプリケーションの実行間で変更される可能性があります。これにより、意図しない変更の原因となる可能性があります。CDK Toolkit のキャッシュ動作は、新しい値を受け入れるまで、CDK アプリケーションのこれらの値を「フリーズ」します。

コンテキストキャッシュのない次のシナリオを想像してみてください。Amazon EC2 インスタンスの AMI として「最新の Amazon Linux」を指定し、この AMI の新しいバージョンがリリースされたとします。次に、次に CDK スタックをデプロイするときに、既にデプロイされているインスタンスが古い (「間違った」) AMI を使用しているため、アップグレードする必要があります。アップグレードすると、既存のすべてのインスタンスが新しいインスタンスに置き換えられ、予期しない望ましくない可能性があります。

代わりに、CDK はアカウントの使用可能な AMIs をプロジェクトの `cdk.context.json` ファイルに記録し、保存された値を将来の合成オペレーションに使用します。これにより、AMIs のリストは変更の潜在的なソースではなくなります。また、スタックが常に同じ AWS CloudFormation テンプレートに合成されるようにすることもできます。

すべてのコンテキスト値が AWS 環境からキャッシュされるわけではありません。 the section called “機能フラグ” はコンテキスト値でもあります。アプリケーションまたはコンストラクトで使用する独自のコンテキスト値を作成することもできます。

コンテキストキーは文字列です。値は、数値、文字列、配列、オブジェクトなど、JSON でサポートされている任意のタイプにすることができます。

> **Tip**
>
> コンストラクトが独自のコンテキスト値を作成する場合は、ライブラリのパッケージ名をキーに組み込み、他のパッケージのコンテキスト値と競合しないようにします。

多くのコンテキスト値が特定の AWS 環境に関連付けられており、特定の CDK アプリを複数の環境にデプロイできます。このような値のキーには、異なる環境の値が競合しないように、 AWS アカウントとリージョンが含まれます。

次のコンテキストキーは、アカウントとリージョンを含む AWS CDK、 で使用される形式を示しています。

```
availability-zones:account=123456789012:region=eu-central-1
```

> **Important**
>
> キャッシュされたコンテキスト値は、書き込むことができるコンストラクトを含め、 AWS CDK とそのコンストラクトによって管理されます。ファイルを手動で編集して、キャッシュされたコンテキスト値を追加または変更しないでください。ただし、キャッシュされている値を確認するために、`cdk.context.json`ときどき を確認すると便利です。キャッシュされた値を表さないコンテキスト値は、 の`context`キーの下に保存する必要がありますcdk.json。これにより、キャッシュされた値がクリアされたときにクリアされません。

コンテキスト値のソース

コンテキスト値は、次の 6 つの異なる方法で AWS CDK アプリに提供できます。

- 現在の AWS アカウントから自動的に。
- cdk コマンドへの `--context`オプションを使用します。(これらの値は常に文字列です。)
- プロジェクトの `cdk.context.json` ファイル。
- プロジェクトの `cdk.json` ファイルの `context`キー。
- `~/.cdk.json` ファイルの `context`キー。
- `construct.node.setContext()` メソッドを使用して AWS CDK アプリで。

プロジェクトファイルは、 AWS アカウントから取得したコンテキスト値を AWS CDK キャッシュする場所`cdk.context.json`です。この方法により、新しいアベイラビリティーゾーンの導入など、デプロイに予期しない変更が加えられるのを回避できます。 AWS CDK は、リストされている他のファイルにはコンテキストデータを書き込みません。

> **Important**
>
> これらはアプリケーションの状態の一部`cdk.json`であるため、アプリケーションの残りのソースコードとともにソース管理にコミット`cdk.context.json`する必要があります。そうしないと、他の環境 (CI パイプラインなど) にデプロイすると、一貫性のない結果が得られる可能性があります。

コンテキスト値は、それらを作成したコンストラクトに限定されます。子コンストラクトには表示されますが、親や兄弟には表示されません。 AWS CDK Toolkit (cdk コマンド) によって設定されたコンテキスト値は、自動的に、ファイルから、または --contextオプションから設定できます。これらのソースのコンテキスト値は、 Appコンストラクトに暗黙的に設定されます。したがって、アプリケーション内のすべてのスタックのすべてのコンストラクトに表示されます。

アプリケーションは、 `construct.node.tryGetContext`メソッドを使用してコンテキスト値を読み取ることができます。リクエストされたエントリが現在のコンストラクトまたはその親のいずれにも見つからない場合、結果は `undefined`です。 (または、NonePython など、言語と同等の結果が得られる可能性があります) 。

context メソッド

は、 AWS CDK アプリケーションが環境から AWS コンテキスト情報を取得できるようにするいくつかのコンテキストメソッド AWS CDK をサポートしています。例えば、Stack.availabilityZones メソッドを使用して、特定の AWS アカウントとリージョンで使用できるアベイラビリティーゾーンのリストを取得できます。

コンテキストメソッドは次のとおりです。

HostedZone.fromLookup

アカウントのホストゾーンを取得します。

stack.availabilityZones

サポートされているアベイラビリティーゾーンを取得します。

StringParameter.valueFromLookup

現在のリージョンの Amazon EC2 Systems Manager パラメータストアから値を取得します。

Vpc.fromLookup

アカウント内の既存の Amazon Virtual Private Cloud を取得します。

LookupMachineImage

Amazon Virtual Private Cloud の NAT インスタンスで使用するマシンイメージを検索します。

必要なコンテキスト値が使用できない場合、 AWS CDK アプリケーションはコンテキスト情報が欠落していることを CDK Toolkit に通知します。次に、 CLI は現在の AWS アカウントに情報をクエリし、結果のコンテキスト情報を cdk.context.json ファイルに保存します。次に、コンテキスト値を使用して AWS CDK アプリケーションを再度実行します。

コンテキストの表示と管理

cdk context コマンドを使用して、 cdk.context.json ファイル内の情報を表示および管理します。この情報を表示するには、オプションなしで cdk context コマンドを使用します。出力は次のようになります。

```
Context found in cdk.json:

##############################################################################################
# # # Key                                                                  # Value
                                    #
##############################################################################################
# 1 # availability-zones:account=123456789012:region=eu-central-1 # [ "eu-central-1a",
 "eu-central-1b", "eu-central-1c" ]   #
##############################################################################################
# 2 # availability-zones:account=123456789012:region=eu-west-1    # [ "eu-west-1a",
 "eu-west-1b", "eu-west-1c" ]            #
##############################################################################################

Run cdk context --reset KEY_OR_NUMBER to remove a context key. If it is a cached value,
 it will be refreshed on the next cdk synth.
```

コンテキスト値を削除するには、値に対応するキーまたは数値cdk context --resetを指定して を実行します。次の例では、前の例の 2 番目のキーに対応する値を削除します。この値は、欧州 (アイルランド) リージョンのアベイラビリティーゾーンのリストを表します。

```
cdk context --reset 2
```

```
Context value
availability-zones:account=123456789012:region=eu-west-1
reset. It will be refreshed on the next SDK synthesis run.
```

したがって、Amazon Linux AMI の最新バージョンに更新する場合は、前の例を使用してコンテキスト値の制御された更新を行い、リセットします。次に、アプリを合成して再度デプロイします。

```
cdk synth
```

アプリケーションの保存されたコンテキスト値をすべてクリアするには、cdk context --clear次のように を実行します。

```
cdk context --clear
```

に保存されているコンテキスト値のみがリセットまたはクリアcdk.context.jsonできます。AWS CDK は他のコンテキスト値には影響しません。したがって、これらのコマンドを使用してコンテキスト値がリセットされないように保護するために、値を にコピーできますcdk.json。

AWS CDK ツールキット--contextフラグ

合成またはデプロイ中にランタイムコンテキスト値を CDK アプリケーションに渡すには、 --context (-c 略) オプションを使用します。

```
cdk synth --context key=value MyStack
```

複数のコンテキスト値を指定するには、--contextオプションを何度でも繰り返し、毎回 1 つのキーと値のペアを指定します。

```
cdk synth --context key1=value1 --context key2=value2 MyStack
```

複数のスタックを合成すると、指定されたコンテキスト値がすべてのスタックに渡されます。個々のスタックに異なるコンテキスト値を指定するには、値に異なるキーを使用するか、複数の cdk synth または cdk deploy コマンドを使用します。

コマンドラインから渡されるコンテキスト値は、常に文字列です。通常、値が他のタイプである場合は、コードを変換または解析する準備を整える必要があります。文字列以外のコンテキスト値が他の方法 (など) で提供されている場合がありますcdk.context.json。この種の値が期待どおりに動作することを確認するには、変換する前に値が文字列であることを確認します。

例

AWS CDK コンテキストを使用して既存の Amazon VPC を使用する例を次に示します。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class ExistsVpcStack extends cdk.Stack {

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid,
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString(),
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const ec2 = require('aws-cdk-lib/aws-ec2');

class ExistsVpcStack extends cdk.Stack {

  constructor(scope, id, props) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});
```

```
    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString()
    });
  }
}

module.exports = { ExistsVpcStack }
```

Python

```
import aws_cdk as cdk
import aws_cdk.aws_ec2 as ec2
from constructs import Construct

class ExistsVpcStack(cdk.Stack):

    def __init__(scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)

        vpcid = self.node.try_get_context("vpcid")
        vpc = ec2.Vpc.from_lookup(self, "VPC", vpc_id=vpcid)

        pubsubnets = vpc.select_subnets(subnetType=ec2.SubnetType.PUBLIC)

        cdk.CfnOutput(self, "publicsubnets",
            value=pubsubnets.subnet_ids.to_string())
```

Java

```
import software.amazon.awscdk.CfnOutput;

import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.ec2.VpcLookupOptions;
import software.amazon.awscdk.services.ec2.SelectedSubnets;
import software.amazon.awscdk.services.ec2.SubnetSelection;
import software.amazon.awscdk.services.ec2.SubnetType;
import software.constructs.Construct;

public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(Construct context, String id) {
        this(context, id, null);
    }
```

```
    public ExistsVpcStack(Construct context, String id, StackProps props) {
        super(context, id, props);

        String vpcId = (String)this.getNode().tryGetContext("vpcid");
        Vpc vpc = (Vpc)Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()
                .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()
                .subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
                .value(pubSubNets.getSubnetIds().toString()).build();

    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.EC2;
using Constructs;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        var vpcId = (string)this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
        {
            VpcId = vpcId
        });

        SelectedSubnets pubSubNets = vpc.SelectSubnets([new SubnetSelection
        {
            SubnetType = SubnetType.PUBLIC
        }]);

        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
```

```
}
```

cdk diff を使用して、コマンドラインでコンテキスト値を渡すことの効果を確認できます。

```
cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

```
Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
 {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
```

結果のコンテキスト値は、次に示すように表示できます。

```
cdk context -j
```

```
{
  "vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-
east-1": {
    "vpcId": "vpc-0cb9c31031d0d3e22",
    "availabilityZones": [
      "us-east-1a",
      "us-east-1b"
    ],
    "privateSubnetIds": [
      "subnet-03ecfc033225be285",
      "subnet-0cded5da53180ebfa"
    ],
    "privateSubnetNames": [
      "Private"
    ],
    "privateSubnetRouteTableIds": [
      "rtb-0e955393ced0ada04",
      "rtb-05602e7b9f310e5b0"
    ],
    "publicSubnetIds": [
      "subnet-06e0ea7dd302d3e8f",
      "subnet-01fc0acfb58f3128f"
    ],
    "publicSubnetNames": [
      "Public"
```

```
    ],
    "publicSubnetRouteTableIds": [
      "rtb-00d1fdfd823c82289",
      "rtb-04bb1969b42969bcb"
    ]
  }
}
```

機能フラグ

AWS CDK は、機能フラグを使用して、リリースで破壊的になり得る動作を有効にします。フラグは cdk.json (または) にthe section called “Context”値として保存されます~/.cdk.json。これらは、 cdk context --resetまたは cdk context --clear コマンドでは削除されません。

機能フラグはデフォルトで無効になっています。フラグを指定しない既存のプロジェクトは、後続の AWS CDK リリースと同様に動作します。を使用して作成された新しいプロジェクトcdk initには、プロジェクトを作成したリリースで利用できるすべての機能を有効にするフラグが含まれています。を編集cdk.jsonして、以前の動作を優先するフラグを無効にします。をアップグレードした後、フラグを追加して新しい動作を有効にすることもできます AWS CDK。

現在のすべての機能フラグのリストは、 の AWS CDK GitHub リポジトリにありますFEATURE_FLAGS.md。そのリリースCHANGELOGで追加された新しい機能フラグの説明については、特定のリリースの を参照してください。

v1 の動作に戻す

CDK v2 では、v1 に関して一部の機能フラグのデフォルトが変更されています。これらを に戻すfalseと、特定の AWS CDK v1 の動作に戻すことができます。cdk diff コマンドを使用して、合成されたテンプレートの変更を調べ、これらのフラグのいずれかが必要かどうかを確認します。

`@aws-cdk/core:newStyleStackSynthesis`

新しいスタック合成メソッドを使用します。これは、よく知られている名前のブートストラップリソースを想定しています。最新のブートストラップが必要ですが、代わりに CDK Pipelines とクロスアカウントデプロイによる CI/CD をすぐに使用できます。

`@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId`

アプリケーションが複数の Amazon API Gateway API キーを使用していて、それらを使用量プランに関連付ける場合。

`@aws-cdk/aws-rds:lowercaseDbIdentifier`

アプリケーションが Amazon RDS データベースインスタンスまたはデータベースクラスターを使用している場合、 と はこれらの識別子を明示的に指定します。

`@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021`

アプリケーションがディス Amazon CloudFront トリビューションで TLS_V1_2_2019 セキュリティポリシーを使用している場合。CDK v2 は、デフォルトでセキュリティポリシー TLSv1.2_2021 を使用します。

`@aws-cdk/core:stackRelativeExports`

アプリケーションで複数のスタックを使用し、あるスタックから別のスタックのリソースを参照する場合、絶対パスまたは相対パスのどちらを使用して AWS CloudFormation エクスポートを構築するかが決まります。

`@aws-cdk/aws-lambda:recognizeVersionProps`

に設定すると`false`、CDK は Lambda 関数が変更されたかどうかを検出するときにメタデータを含めます。これにより、重複バージョンは許可されないため、メタデータのみが変更されたときにデプロイが失敗する可能性があります。アプリケーション内のすべての Lambda 関数に少なくとも 1 つの変更を加えた場合、このフラグを元に戻す必要はありません。

で`cdk.json`これらのフラグを元に戻すための構文を以下に示します。

```
{
  "context": {
    "@aws-cdk/core:newStyleStackSynthesis": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021": false,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/core:stackRelativeExports": false,
    "@aws-cdk/aws-lambda:recognizeVersionProps": false
  }
}
```

側面

アスペクトは、特定のスコープ内のすべてのコンストラクトにオペレーションを適用する方法です。側面は、タグを追加するなどしてコンストラクトを変更できます。または、すべてのバケットが暗号化されていることを確認するなど、コンストラクトの状態について何かを検証することもできます。

同じスコープ内のコンストラクトとすべてのコンストラクトにアスペクトを適用するには、次の例に示すように、新しいアスペクト`Aspects.of(SCOPE).add()`で を呼び出します。

TypeScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

JavaScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

Python

```
Aspects.of(my_construct).add(SomeAspect(...))
```

Java

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

C#

```
Aspects.Of(myConstruct).add(new SomeAspect(...));
```

Go

```
awscdk.Aspects_Of(stack).Add(awscdk.NewTag(...))
```

AWS CDK は の側面を使用してリソース にタグを付けますが、フレームワークは他の目的にも使用できます。例えば、これを使用して、上位レベルのコンストラクトによって定義された AWS CloudFormation リソースを検証または変更できます。

アスペクトの詳細

アスペクトは訪問者パターン を採用します。アスペクトとは、次のインターフェイスを実装するクラスです。

TypeScript

```
interface IAspect {
```

```
    visit(node: IConstruct): void;}
```

JavaScript

JavaScript には言語機能としてインターフェイスはありません。したがって、アスペクトとは、操作対象のノードを受け入れるvisitメソッドを持つクラスのインスタンスのことです。

Python

Python には言語機能としてインターフェイスはありません。したがって、アスペクトとは、運用するノードを受け入れるvisitメソッドを持つクラスのインスタンスにすぎません。

Java

```
public interface IAspect {
    public void visit(Construct node);
}
```

C#

```
public interface IAspect
{
    void Visit(IConstruct node);
}
```

Go

```
type IAspect interface {
  Visit(node constructs.IConstruct)
}
```

を呼び出すとAspects.of(*SCOPE*).add(...)、コンストラクトは アスペクトの内部リストに アスペクトを追加します。リストは で取得できますAspects.of(*SCOPE*)。

準備フェーズ 中に、 はコンストラクトとその子それぞれについて、 オブジェクトの visitメソッドを上位の順序で AWS CDK 呼び出します。

visit メソッドは、コンストラクト内の任意の内容を自由に変更できます。厳密に型指定された言語で、コンストラクト固有のプロパティやメソッドにアクセスする前に、受信したコンストラクトをより具体的なタイプにキャストします。

は定義後に自己完結型でイミュータブルであるため、アスペクトStagesはStageコンストラクトの境界を越えて伝播されません。内のStageコンストラクトにアクセスする場合は、コンストラクト自体 (またはそれ以下) に側面を適用しますStage。

例

次の例では、スタックで作成されたすべてのバケットでバージョニングが有効になっていることを確認します。アスペクトは、検証に失敗したコンストラクトにエラーアノテーションを追加します。これにより、synthオペレーションが失敗し、結果のクラウドアセンブリがデプロイされなくなります。

TypeScript

```
class BucketVersioningChecker implements IAspect {
  public visit(node: IConstruct): void {
    // See that we're dealing with a CfnBucket
    if (node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || (!Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

JavaScript

```
class BucketVersioningChecker {
   visit(node) {
    // See that we're dealing with a CfnBucket
    if ( node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
```

```
        || !Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

Python

```
@jsii.implements(cdk.IAspect)
class BucketVersioningChecker:

  def visit(self, node):
    # See that we're dealing with a CfnBucket
    if isinstance(node, s3.CfnBucket):

      # Check for versioning property, exclude the case where the property
      # can be a token (IResolvable).
      if (not node.versioning_configuration or
              not Tokenization.is_resolvable(node.versioning_configuration)
                  and node.versioning_configuration.status != "Enabled"):
          Annotations.of(node).add_error('Bucket versioning is not enabled')

# Later, apply to the stack
Aspects.of(stack).add(BucketVersioningChecker())
```

Java

```
public class BucketVersioningChecker implements IAspect
{
    @Override
    public void visit(Construct node)
    {
        // See that we're dealing with a CfnBucket
        if (node instanceof CfnBucket)
        {
            CfnBucket bucket = (CfnBucket)node;
            Object versioningConfiguration = bucket.getVersioningConfiguration();
            if (versioningConfiguration == null ||
```

```
                    !Tokenization.isResolvable(versioningConfiguration.toString())
 &&
                    !versioningConfiguration.toString().contains("Enabled"))
                Annotations.of(bucket.getNode()).addError("Bucket versioning is not
 enabled");
        }
    }
}


// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

C#

```
class BucketVersioningChecker : Amazon.Jsii.Runtime.Deputy.DeputyBase, IAspect
{
    public void Visit(IConstruct node)
    {
        // See that we're dealing with a CfnBucket
        if (node is CfnBucket)
        {
            var bucket = (CfnBucket)node;
            if (bucket.VersioningConfiguration is null ||
                    !Tokenization.IsResolvable(bucket.VersioningConfiguration) &&
                    !bucket.VersioningConfiguration.ToString().Contains("Enabled"))
                Annotations.Of(bucket.Node).AddError("Bucket versioning is not
 enabled");
        }
    }
}

// Later, apply to the stack
Aspects.Of(stack).add(new BucketVersioningChecker());
```

の開始方法 AWS CDK

をインストールし AWS Cloud Development Kit (AWS CDK) 、最初の CDK アプリを作成して、AWS CDK CLIの使用を開始します。

トピック

前提条件

推奨リソース

の使用を開始する前に AWS CDK、以下の基本的な理解を深めておくことをお勧めします。

- の概要 AWS CDK。詳細については、「は何ですか AWS CDK?」を参照してください。
- の背後にあるコア概念 AWS CDK。詳細については、「AWS CDK 主要概念を学ぶ」を参照してください。
- で AWS のサービス 管理する AWS CDK。
- AWS Identity and Access Management。 詳細については、「IAM とは」および「IAM Identity Center とは」を参照してください。
- AWS CloudFormation は AWS CloudFormation サービス AWS CDK を利用して CDK で作成されたリソースをプロビジョニングするためです。詳細については、「AWS CloudFormationとは?」を参照してください。
- で使用する予定のサポートされているプログラミング言語 AWS CDK。

ローカル環境を準備する

希望する言語に関係なく、すべての AWS CDK デベロッパーには Node.js 14.15.0 以降が必要です。サポートされているすべてのプログラミング言語は、 で実行される同じバックエンドを使用しますNode.js。アクティブな長期サポート のバージョンをお勧めします。組織によっては、別のレコメンデーションがある場合があります。

> **Important**
>
> Node.js バージョン 13.0.0 から 13.6.0 は、依存関係との互換性の問題 AWS CDK のため、 と互換性がありません。

その他の前提条件は、 AWS CDK アプリケーションを開発する言語によって異なり、次のようになります。

TypeScript

- TypeScript 3.8 以降 (`npm -g install typescript`)

JavaScript

追加要件なし

Python

- `pip` および を含む Python 3.7 以降 `virtualenv`

Java

- Java Development Kit (JDK) 8 (a.k.a. 1.8) 以降
- Apache Maven 3.5 以降

Java IDE が推奨されます (このガイドのいくつかの例では Eclipse を使用しています)。IDE は Maven プロジェクトをインポートできる必要があります。プロジェクトが Java 1.8 を使用するように設定されていることを確認します。JAVA_HOME 環境変数を JDK をインストールしたパスに設定します。

C#

.NET Core 3.1 以降、または .NET 6.0 以降。

Visual Studio 2019 (任意の エディション) または Visual Studio Code を推奨します。

Go

Go 1.1.8 以降。

詳細については、言語の前提条件セクションを参照してください。

- the section called “内 TypeScript”
- the section called “内 JavaScript”
- the section called “Python の場合”
- the section called “Java の場合”
- the section called “C# で”
- the section called “Go で”

> **サードパーティー言語の廃止**
>
> 各言語バージョンは、 EOL (サポート終了) までのみサポートされ、事前に通知して変更される可能性があります。

ステップ 1: を作成する AWS アカウント

を初めて使用する場合は AWS、 にサインアップ AWS アカウント して管理ユーザーを作成する必要があります。詳細については、「IAM ユーザーガイド」の「IAM のセットアップ」を参照してください。

を操作するときは AWS、 AWS セキュリティ認証情報を指定して、本人であること、およびリクエストしているリソースへのアクセス許可があるかどうかを検証します。 は、リクエストを認証および承認するためにセキュリティ認証情報 AWS を使用します。詳細については、「IAM ユーザーガイド」のAWS 「 セキュリティ認証情報」を参照してください。

ステップ 2: プログラムによるアクセスを設定する

AWS CDK ローカル環境で を使用して開発する場合、 を使用して AWS CDK CLI AWS リソースを操作し AWS のサービス 、管理します。を使用するには AWS CDK CLI、プログラムによるアクセスを設定する必要があります。プログラムによるアクセスを設定するさまざまな方法の詳細については、「 SDK とツールのリファレンスガイド」の「認証とアクセス」を参照してください。 AWS SDKs

雇用主から認証方法が与えられていない新規ユーザーには、 を使用することをお勧めします AWS IAM Identity Center。この方法には、 AWS Command Line Interface （ AWS CLI) をインストー

ルし、設定と AWS アクセスポータルへのサインインに使用することが含まれます。IAM Identity Center を使用してプログラムによるアクセスを設定するには、「 SDK およびツールリファレンスガイド」の「IAM Identity Center 認証」を参照してください。 AWS SDKs 完了したら、環境に次の要素が含まれている必要があります。

- アプリケーションを実行する前に AWS アクセスポータルセッションを開始 AWS CLIするために使用する 。
- から参照できる設定値のセットを持つ[default]プロファイルを持つ共有 AWSconfigファイル AWS CDK。このファイルの場所を確認するには、AWS SDK とツールのリファレンスガイドの「共有ファイルの場所」を参照してください。
- 共有 config ファイルは region 設定を設定します。これにより、 AWS リージョン が AWS リクエスト AWS CDK に使用するデフォルトが設定されます。
- AWS CDK は、リクエストを に送信する前に、プロファイルの SSO トークンプロバイダー設定を使用して認証情報を取得します AWS。IAM Identity Center アクセス許可セットに接続された IAM ロールである sso_role_name値は、アプリケーションで AWS のサービス 使用されている へのアクセスを許可する必要があります。

  次のサンプル config ファイルは、SSO トークンプロバイダー設定で設定されたデフォルトプロファイルを示しています。プロファイルの sso_session 設定は、指定された sso-session セクションを参照します。sso-session セクションには、 AWS アクセスポータルセッションを開始するための設定が含まれています。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

AWS アクセスポータルセッションを開始する

にアクセスする前に AWS のサービス、 が IAM Identity Center 認証を使用して認証情報 AWS CDK を解決するためのアクティブな AWS アクセスポータルセッションが必要です。設定したセッション

の長さによっては、アクセスが最終的に期限切れになり、 で認証エラー AWS CDK が発生します。で次のコマンドを実行して AWS CLI 、 AWS アクセスポータルにサインインします。

```
aws sso login
```

SSO トークンプロバイダーの設定で、デフォルトのプロファイルではなく名前付きプロファイルを使用している場合、コマンドは ですaws sso login --profile *NAME*。また、 --profileオプションまたは AWS_PROFILE環境変数を使用してcdkコマンドを発行するときにも、このプロファイルを指定します。

既にアクティブなセッションがあるかどうかをテストするには、次の AWS CLI コマンドを実行します。

```
aws sts get-caller-identity
```

このコマンドへの応答により、共有 config ファイルに設定されている IAM Identity Center アカウントとアクセス許可のセットが報告されます。

> **Note**
>
> 既にアクティブな AWS アクセスポータルセッションがあり、 を実行している場合はaws sso login、 認証情報を提供する必要はありません。
> サインインプロセスでは、データ AWS CLI へのアクセスを許可するように求められる場合があります。 AWS CLI は SDK for Python 上に構築されているため、アクセス許可メッセージにはbotocore名前のバリエーションが含まれている場合があります。

ステップ 3: をインストールする AWS CDKCLI

次の Node Package Manager コマンドを使用して、 AWS CDK CLI をグローバルにインストールします。

```
npm install -g aws-cdk
```

> **Note**
>
> アクセス許可エラーが発生し、システムで管理者アクセス権がある場合は、 を試してくださいsudo npm install -g aws-cdk。

次のコマンドを実行して、正常にインストールされたことを確認します。はバージョン番号を出力 AWS CDK CLIする必要があります。

```
cdk --version
```

エラーメッセージが表示された場合は、以下を実行して を AWS CDK CLIアンインストールしてみてください。

```
npm uninstall -g aws-cdk
```

次に、ステップを繰り返して を再インストールします AWS CDK CLI。

それでもエラーが表示される場合は、現在のプロジェクトからnode-modulesフォルダを削除し、グローバルnode-modulesフォルダからも削除します。このフォルダを見つけるには、 を実行しますnpm config get prefix。

AWS CDK CLI は、前の手順で設定したソースからセキュリティ認証情報を取得します。

> **Note**
>
> CDK Toolkit v2 は、既存の CDK v1 プロジェクトで動作します。ただし、新しい CDK v1 プロジェクトを初期化することはできません。これを実行する必要があるthe section called “新しい前提条件”かどうかを確認します。

ステップ 4: 環境をブートストラップする

リソースをデプロイする予定の各 AWS 環境は、ブートストラップ する必要があります。

ブートストラップするには、以下を実行します。

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

> **Tip**
>
> AWS アカウント番号をお持ちでない場合は、 から入手できます AWS Management Console。または、 AWS CLI がインストールされている場合は、次のコマンドで、アカウント番号を含むデフォルトのアカウント情報が表示されます。

```
aws sts get-caller-identity
```

ローカル AWS 設定で名前付きプロファイルを作成した場合は、 `--profile`オプションを使用して、特定のプロファイルのアカウント情報を表示できます。次の例は、prod プロファイルのアカウント情報を表示する方法を示しています。

```
aws sts get-caller-identity --profile prod
```

デフォルトのリージョンを表示するには、 を使用しますaws configure get。

```
aws configure get region
aws configure get region --profile prod
```

オプションの AWS CDK ツール

AWS Toolkit for Visual Studio Code は Visual Studio Code 用のオープンソースプラグインで、 でのアプリケーションの作成、デバッグ、デプロイに役立ちます AWS。このツールキットは、 AWS CDK アプリケーションを開発するための統合エクスペリエンスを提供します。これには、プロジェクトを AWS CDK 一覧表示し、 AWS CDK アプリケーションのさまざまなコンポーネントを参照する CDK Explorer 機能が含まれています。プラグインをインストールし、AWS CDK Explorer の使用に関する詳細を確認します。

次のステップ

をインストールしたので AWS CDK CLI、それを使用して最初の AWS CDK アプリ を構築します。

任意のプログラミング言語で を使用する方法の詳細については、 AWS CDK 「」を参照してくださいサポートされているプログラミング言語で AWS CDK の の操作。

AWS CDK はオープンソースプロジェクトです。貢献するには、「 への寄稿 AWS Cloud Development Kit (AWS CDK)」を参照してください。

詳細はこちら

の詳細については AWS CDK、以下を参照してください。

- CDK ワークショップ — 詳細な実践的なワークショップ。
- API リファレンス — 使用する で使用できるコンストラクト AWS のサービス を調べます。
- Construct Hub — CDK コミュニティからコンストラクトを検索します。
- AWS CDK examples – AWS CDK プロジェクトのコード例を調べます。

最初の AWS CDK アプリ

最初の CDK アプリを構築 AWS Cloud Development Kit (AWS CDK) して、 の使用を開始します。

このチュートリアルを開始する前に、以下を完了することをお勧めします。

- の概要は何ですか AWS CDK?については、「」を参照してください AWS CDK。
- のコアコンセプトAWS CDK 主要概念を学ぶについては、「」を参照してください AWS CDK。
- の前提条件と AWS CDK セットアップ手順について説明しますの開始方法 AWS CDK。

トピック

このチュートリアルの内容

このチュートリアルでは、シンプルな AWS CDK アプリケーションを作成してデプロイします。このアプリには、単一の Amazon Simple Storage Service (Amazon S3) バケットリソースを持つ 1 つのスタックが含まれています。このチュートリアルでは、以下について学習します。

- AWS CDK プロジェクトの構造。
- AWS CDK アプリを作成する方法。
- AWS 構築ライブラリを使用してアプリケーション、スタック、 AWS リソースを定義する方法。
- CDK を使用して CDK アプリをCLI合成、差分、デプロイ、削除する方法。
- CDK アプリを変更して再デプロイし、デプロイされたリソースを更新する方法。

標準 AWS CDK の開発ワークフローは、次のステップで構成されます。

1. AWS CDK アプリを作成する – ここでは、CDK が提供するテンプレートを使用しますCLI。
2. スタックとリソースの定義 — コンストラクトを使用して、アプリケーション内のスタックと AWS リソースを定義します。
3. アプリの構築 — このステップはオプションです。CDK は、必要に応じてこのステップCLIを自動的に実行します。構文エラーとタイプエラーを特定するには、このステップを実行することをお勧めします。
4. スタックを合成する – このステップでは、アプリケーション内のスタックごとに AWS CloudFormation テンプレートを作成します。このステップは、定義した AWS リソースの論理エラーを特定するのに役立ちます。
5. アプリケーションをデプロイする – を使用して AWS 環境にデプロイし AWS CloudFormation 、リソースをプロビジョニングします。デプロイ中に、アプリのアクセス許可の問題を特定します。

一般的なワークフローでは、前のステップに戻って繰り返し、アプリを変更またはデバッグします。

AWS CDK プロジェクトにはバージョン管理を使用することをお勧めします。

ステップ 1: アプリケーションを作成する

CDK アプリケーションは、独自のローカルモジュールの依存関係を持つ独自のディレクトリにある必要があります。開発マシンで、新しいディレクトリを作成します。新しいhello-cdkディレクトリを作成する例を次に示します。

```
$ mkdir hello-cdk
$ cd hello-cdk
```

> **Important**
>
> ここに示されているhello-cdkとおり、プロジェクトディレクトリに という名前を付けます。 AWS CDK プロジェクトテンプレートは、ディレクトリ名を使用して、生成されたコード内のモノに名前を付けます。別の名前を使用すると、このチュートリアルのコードは機能しません。

次に、新しいディレクトリから、 cdk init コマンドを使用してアプリケーションを初期化します。--language オプションを使用して、appテンプレートと任意のプログラミング言語を指定します。以下に例を示します。

TypeScript

```
$ cdk init app --language typescript
```

JavaScript

```
$ cdk init app --language javascript
```

Python

```
$ cdk init app --language python
```

アプリを作成したら、次の 2 つのコマンドも入力します。これにより、アプリケーションの Python 仮想環境がアクティブ化され、 AWS CDK コア依存関係がインストールされます。

```
$ source .venv/bin/activate # On Windows, run `.\venv\Scripts\activate` instead
$ python -m pip install -r requirements.txt
```

Java

```
$ cdk init app --language java
```

IDE を使用している場合は、プロジェクトを開くかインポートできるようになりました。例えば、Eclipse で、ファイル > インポート > Maven > 既存の Maven プロジェクト を選択します。プロジェクト設定が Java 8 (1.8) を使用するように設定されていることを確認します。

C#

```
$ cdk init app --language csharp
```

Visual Studio を使用している場合は、 `src` ディレクトリでソリューションファイルを開きます。

Go

```
$ cdk init app --language go
```

アプリケーションを作成したら、次のコマンドを入力して、アプリケーションに必要な AWS コンストラクトライブラリモジュールをインストールします。

```
$ go get
```

`cdk init` コマンドは、 AWS CDK アプリケーションのソースコードを整理しやすくするために、`hello-cdk` ディレクトリ内に多数のファイルとフォルダを作成します。総称して、これはプロジェクト AWS CDK と呼ばれます。 少し時間をとって CDK プロジェクトを試してください。

Git がインストールされている場合、 を使用して作成した各プロジェクト`cdk init`もGitリポジトリとして初期化されます。

ステップ 2: アプリを構築する

ほとんどのプログラミング環境では、変更後にコードを構築またはコンパイルします。CDK は自動的にこのステップを実行する AWS CDK ためCLI、 では必要ありません。ただし、構文と型エラーをキャッチする場合は、手動で構築できます。以下に例を示します。

TypeScript

```
$ npm run build
```

JavaScript

ビルドステップは必要ありません。

Python

ビルドステップは必要ありません。

Java

```
$ mvn compile -q
```

または、Eclipse `Control-B`で を押します (他の Java IDEs は異なる場合があります)。

C#

```
$ dotnet build src
```

または、Visual Studio で F6 キーを押します。

Go

```
$ go build
```

ステップ 3: アプリ内のスタックを一覧表示する

アプリケーションにスタックを一覧表示して、アプリケーションが正しく作成されたことを確認します。下記を実行します。

```
$ cdk ls
```

出力には と表示されます`HelloCdkStack`。この出力が表示されない場合は、プロジェクトの正しい作業ディレクトリにあることを確認し、もう一度試してください。それでもスタックが表示されない場合は、 を繰り返しthe section called “ステップ 1: アプリケーションを作成する”て再試行してください。

ステップ 4: Amazon S3 バケットを追加する

この時点で、CDK アプリには 1 つのスタックが含まれています。次に、スタック内で Amazon Simple Storage Service (Amazon S3) バケットリソースを定義します。これを行うには、コンストラクトライブラリから `Bucket` L2 AWS コンストラクトをインポートして使用します。

`Bucket` コンストラクトをインポートし、Amazon S3 バケットリソースを定義して、CDK アプリケーションを変更します。以下に例を示します。

TypeScript

Eclipse `lib/hello-cdk-stack.ts`:

```
import * as cdk from 'aws-cdk-lib';
import { aws_s3 as s3 } from 'aws-cdk-lib';

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

JavaScript

Eclipse `lib/hello-cdk-stack.js`:

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

module.exports = { HelloCdkStack }
```

Python

Eclipse `hello_cdk/hello_cdk_stack.py`:

```
from aws_cdk import (
    Stack,
    aws_s3 as s3,
)
from constructs import Construct
```

```
class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        bucket = s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

Eclipse `src/main/java/com/myorg/HelloCdkStack.java`:

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.Bucket;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

C#

Eclipse `src/HelloCdk/HelloCdkStack.cs`:

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdk
{
    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(App scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
```

```
            new Bucket(this, "MyFirstBucket", new BucketProps
            {
                Versioned = true
            });
        }
    }
}
```

Go

Eclipse `hello-cdk.go`:

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type HelloCdkStackProps struct {
 awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}

func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)
```

```
 NewHelloCdkStack(app, "HelloCdkStack", &HelloCdkStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 app.Synth(nil)
}

func env() *awscdk.Environment {
 return nil
}
```

Bucket コンストラクトを詳しく見てみましょう。すべてのコンストラクトと同様に、 Bucket クラスは 3 つのパラメータを取ります。

- scope – Stack クラスをBucketコンストラクトの親として定義します。 AWS リソースを定義するすべてのコンストラクトは、スタックの範囲内で作成されます。コンストラクト内でコンストラクトを定義し、階層 (ツリー) を作成できます。ここで、ほとんどの場合、スコープは this (self の) ですPython。
- ID – AWS CDK アプリBucket内の の論理 ID。この ID と、スタック内のバケットの場所に基づくハッシュは、デプロイ中にバケットを一意に識別します。は、アプリで コンストラクトを更新し、デプロイされたリソースを更新するために再デプロイするときに AWS CDK も、この ID を参照します。ここで、論理 ID は ですMyFirstBucket。バケットには、 bucketNameプロパティで指定された名前を付けることもできます。これは論理 ID とは異なります。
- props – バケットのプロパティを定義する値のバンドル。ここでは、 versionedプロパティを として定義しました。これによりtrue、バケット内のファイルのバージョニングが有効になります。

  Props は、 でサポートされている言語で異なる表現になっています AWS CDK。

  - TypeScript および では JavaScript、 propsは単一の引数であり、目的のプロパティを含むオブジェクトを渡します。
  - Python では、props はキーワード引数として渡されます。
  - Java では、props を渡すための Builder が用意されています。2 つあります。1 つは用BucketProps、もう 1 つは 用で、コンストラクトとその props オブジェクトを 1 ステップで構築Bucketできます。このコードは後者を使用します。

- C# では、BucketPropsオブジェクトイニシャライザを使用してオブジェクトをインスタンス化し、3 番目のパラメータとして渡します。

コンストラクトの props がオプションの場合は、 propsパラメータを完全に省略できます。

すべてのコンストラクトはこれらと同じ 3 つの引数を取るため、新しい引数について学習しても、常に向きを合わせることは簡単です。また、予想どおり、任意のコンストラクトをサブクラスして、ニーズに合わせて拡張したり、デフォルトを変更したりできます。

ステップ 5: AWS CloudFormation テンプレートを合成する

次のように、アプリケーションの AWS CloudFormation テンプレートを合成します。

```
$ cdk synth
```

アプリケーションに複数のスタックが含まれている場合は、合成するスタックを指定する必要があります。アプリケーションには 1 つのスタックが含まれているため、CDK は合成するスタックCLIを自動的に検出します。

を実行しない場合cdk synth、デプロイ時に CDK CLIが自動的にこのステップを実行します。ただし、各デプロイの前にこのステップを実行することをお勧めします。

> **Tip**
>
> などのエラーが表示された場合は--app is required ...、CDK CLI コマンドを実行しているディレクトリを確認します。メインアプリケーションディレクトリにある必要があります。

cdk synth コマンドはアプリを実行します。これにより、アプリケーション内のスタックごとに AWS CloudFormation テンプレートが作成されます。CDK CLIはコマンドラインに YAML 形式のテンプレートバージョンを表示し、JSON 形式のテンプレートバージョンを cdk.out ディレクトリに保存します。以下は、 AWS CloudFormation テンプレートで定義されているバケットを示すコマンドライン出力のスニペットです。

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
```

```
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
    Metadata: #...
```

> **Note**
>
> 生成されたすべてのテンプレートには、デフォルトで AWS::CDK::Metadataリソースが含まれています。 AWS CDK チームはこのメタデータを使用して AWS CDK 使用状況を把握し、改善する方法を見つけます。バージョンレポートをオプトアウトする方法などの詳細については、「」を参照してくださいバージョンレポート。

生成されたテンプレートは、 AWS CloudFormation コンソールまたは任意のデプロイツールを使用して AWS CloudFormation デプロイできます。CDK を使用してCLIデプロイすることもできます。次のステップでは、CDK CLI を使用してデプロイします。

ステップ 6: スタックをデプロイする

CDK AWS CloudFormation を使用して CDK スタックを にデプロイするにはCLI、以下を実行します。

```
$ cdk deploy
```

> **Important**
>
> デプロイする前に、 AWS 環境のブートストラップを 1 回実行する必要があります。手順については、「環境のブートストラップ」を参照してください。

と同様にcdk synth、アプリケーションには 1 つの AWS CDK スタックが含まれているため、スタックを指定する必要はありません。

コードにセキュリティ上の影響がある場合、CDK CLI は概要を出力します。デプロイを続行するには、それらを確認する必要があります。このチュートリアルのアプリには、このような影響はありません。

を実行するとcdk deploy、スタックがデプロイされると CDK に進行状況情報CLIが表示されます。完了したら、AWS CloudFormation コンソールに移動してHelloCdkStackスタックを表示できます。Amazon S3 コンソールに移動してMyFirstBucketリソースを表示することもできます。

お疲れ様でした。を使用して最初のスタックをデプロイしました AWS CDK。次に、アプリを変更し、再デプロイしてリソースを更新します。

ステップ 7: アプリを変更する

このステップでは、スタックが削除されたときに自動的に削除されるように設定することで、Amazon S3 バケットを変更します。この変更には、バケットの RemovalPolicyプロパティの変更が含まれます。また、破棄する前にバケットからオブジェクトCLIを削除するように CDK を設定するように autoDeleteObjectsプロパティを設定します。デフォルトでは、 オブジェクトを含む Amazon AWS CloudFormation S3 バケットは削除されません。 Amazon S3

次の例を使用してリソースを変更します。

TypeScript

lib/hello-cdk-stack.ts を更新する。

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

JavaScript

lib/hello-cdk-stack.js を更新する。

```
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  autoDeleteObjects: true
});
```

Python

hello_cdk/hello_cdk_stack.py を更新する。

```
from aws_cdk import (
```

```
    # ...
    RemovalPolicy,
)
# ...

class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        bucket = s3.Bucket(self, "MyFirstBucket",
                           versioned=True,
                           removal_policy=RemovalPolicy.DESTROY,
                           auto_delete_objects=True)
```

Java

src/main/java/com/myorg/HelloCdkStack.java を更新する。

```
Bucket.Builder.create(this, "MyFirstBucket")
        .versioned(true)
        .removalPolicy(RemovalPolicy.DESTROY)
        .autoDeleteObjects(true)
        .build();
```

C#

src/HelloCdk/HelloCdkStack.cs を更新する。

```
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true,
    RemovalPolicy = RemovalPolicy.DESTROY,
    AutoDeleteObjects = true
});
```

Go

hello-cdk.go を更新する。

```
   awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned:         jsii.Bool(true),
  RemovalPolicy:     awscdk.RemovalPolicy_DESTROY,
```

```
 AutoDeleteObjects: jsii.Bool(true),
})
```

現在、コードの変更によって、デプロイされた Amazon S3 バケットリソースに直接更新されることはありません。コードは、リソースの目的の状態を定義します。デプロイされたリソースを変更するには、CDK CLI を使用して目的の状態を新しい AWS CloudFormation テンプレートに合成します。次に、新しい AWS CloudFormation テンプレートを変更セットとしてデプロイします。変更セットは、新しい目的の状態に到達するために必要な変更のみを行います。

これらの変更を確認するには、 `cdk diff` コマンドを使用します。下記を実行します。

```
$ cdk diff
```

CDK は、`HelloCdkStack`スタックの最新の AWS CloudFormation テンプレートを AWS アカウント アカウントでCLIクエリします。次に、最新のテンプレートと、アプリケーションから合成したばかりのテンプレートを比較します。出力は以下のようになります。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                    # Effect # Action                      # Principal
                  # Condition #
###############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole              #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                           #
                  #           #
#   # .Arn}                       #        #                           #
                  #           #
###############################################################################################
# + # ${MyFirstBucket.Arn}        # Allow  # s3:DeleteObject*            # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*      #        # s3:GetBucket*              #
 jectsCustomResourceProvider/ #           #
#   #                             #        # s3:GetObject*              # Role.Arn}
                  #           #
#   #                             #        # s3:List*                   #
                  #           #
###############################################################################################
IAM Policy Changes
###############################################################################################
```

```
#   # Resource                                                  # Managed Policy ARN
                                #
##############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                     # ice-role/
AWSLambdaBasicExecutionRole"}                 #
##############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
```

```
    ## [-] Retain
    ## [+] Delete
```

この差分には 4 つのセクションがあります。

- IAM ステートメントの変更と IAM ポリシーの変更 — Amazon S3 バケットに `AutoDeleteObjects`プロパティを設定するため、これらのアクセス許可の変更があります。自動削除機能は、カスタムリソースを使用して、バケット自体が削除される前にバケット内のオブジェクトを削除します。IAM オブジェクトは、カスタムリソースのコードにバケットへのアクセスを許可します。
- パラメータ – AWS CDK はこれらのエントリを使用して、カスタムリソースの関数アセットを見つけ AWS Lambda ます。
- リソース — このスタックの新しいリソースと変更されたリソース。前述の IAM オブジェクト、カスタムリソース、および関連する Lambda 関数が追加されていることを確認できます。また、バケットの 属性`DeletionPolicy`と `UpdateReplacePolicy` 属性が更新されていることも確認できます。これにより、バケットをスタックとともに削除し、新しいバケットに置き換えることができます。

AWS CDK アプリ`RemovalPolicy`で を指定しましたが、結果の AWS CloudFormation テンプレートに `DeletionPolicy` プロパティが取得されていることに気付くかもしれません。これは、 が プロパティに別の名前 AWS CDK を使用するためです。デフォルトでは AWS CDK 、スタックが削除されたときにバケットを保持し、 AWS CloudFormation デフォルトではバケットを削除します。詳細については、「the section called “削除ポリシー”」を参照してください。

新しい AWS CloudFormation テンプレートを表示するには、 を実行しますcdk synth。CDK アプリケーションにいくつかの変更を加えることで、新しい AWS CloudFormation テンプレートに元の AWS CloudFormation テンプレートと比較して多くのコード行が追加されるようになりました。

次に、以下を実行してアプリケーションをデプロイします。

```
$ cdk deploy
```

AWS CDK は、差分で既に確認したセキュリティポリシーの変更について通知します。y を入力して変更を承認し、更新されたスタックをデプロイします。CDK CLIはスタックをデプロイして、必要な変更を加えます。以下は、その出力例です。

```
HelloCdkStack: deploying...
```

```
[0%] start: Publishing
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
[100%] success: Published
 4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392:current
HelloCdkStack: creating CloudFormation changeset...
 0/5 | 4:32:31 PM | UPDATE_IN_PROGRESS   | AWS::CloudFormation::Stack  | HelloCdkStack
 User Initiated
 0/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
 1/5 | 4:32:36 PM | UPDATE_COMPLETE      | AWS::S3::Bucket             | MyFirstBucket
 (MyFirstBucketB8884501)
 1/5 | 4:32:36 PM | CREATE_IN_PROGRESS   | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092) Resource creation
 Initiated
 3/5 | 4:32:54 PM | CREATE_COMPLETE      | AWS::IAM::Role
       | Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 (CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
     | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
 3/5 | 4:32:56 PM | CREATE_IN_PROGRESS   | AWS::Lambda::Function
     | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F) Resource creation
 Initiated
 3/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::Lambda::Function
     | Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 (CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F)
 3/5 | 4:32:57 PM | CREATE_IN_PROGRESS   | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD) Resource creation Initiated
 4/5 | 4:32:57 PM | CREATE_COMPLETE      | AWS::S3::BucketPolicy       | MyFirstBucket/
Policy (MyFirstBucketPolicy3243DEFD)
 4/5 | 4:32:59 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
 5/5 | 4:33:06 PM | CREATE_IN_PROGRESS   | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E) Resource creation Initiated
 5/5 | 4:33:06 PM | CREATE_COMPLETE      | Custom::S3AutoDeleteObjects
 | MyFirstBucket/AutoDeleteObjectsCustomResource/Default
 (MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E)
```

```
 5/5 | 4:33:08 PM | UPDATE_COMPLETE_CLEA | AWS::CloudFormation::Stack  | HelloCdkStack
 6/5 | 4:33:09 PM | UPDATE_COMPLETE      | AWS::CloudFormation::Stack  | HelloCdkStack

 #  HelloCdkStack

Stack ARN:
arn:aws:cloudformation:REGION:ACCOUNT:stack/HelloCdkStack/UNIQUE-ID
```

ステップ 8: アプリのリソースを破棄する

このチュートリアルを完了したので、デプロイされた AWS CloudFormation スタックとそれに関連付けられているすべてのリソースを削除できます。これは、不要なコストを最小限に抑え、環境をクリーンに保つための良い方法です。下記を実行します。

```
$ cdk destroy
```

y を入力して変更を承認し、スタックを削除します。

> **Note**
>
> バケットの を変更しなかった場合RemovalPolicy、スタックの削除は正常に完了しますが、バケットは孤立します (スタックに関連付けられなくなります) 。

次のステップ

お疲れ様でした。このチュートリアルを完了し、 を使用して でリソースを AWS CDK 正常に作成、変更、削除しました AWS クラウド。これで、 の使用を開始する準備ができました AWS CDK。

AWS CDK 任意のプログラミング言語で を使用する方法の詳細については、「」を参照してくださいサポートされているプログラミング言語で AWS CDK の の操作。

その他のリソースについては、以下を参照してください。

- CDK ワークショップを試して、より複雑なプロジェクトを含むより詳細なツアーをお試しください。
- 、the section called “環境”、、the section called “アセット”、、 the section called “アクセス許可”などの概念について詳しく説明しますthe section called “Context”the section called “パラメータ”the section called “コンストラクトのカスタマイズ”。

- API リファレンスを参照して、お好みのサービスで利用可能な CDK コンストラクトの探索を開始してください AWS 。
- Construct Hub にアクセスして、 AWS などによって作成されたコンストラクトを見つけます。
- の使用例をご覧ください AWS CDK。

AWS CDK はオープンソースプロジェクトです。貢献するには、「 への貢献 AWS Cloud Development Kit (AWS CDK)」を参照してください。

サポートされているプログラミング言語で AWS CDK の の操作

を使用して AWS Cloud Development Kit (AWS CDK) 、サポートされているプログラミング言語 で AWS クラウド インフラストラクチャを定義します。

トピック

- AWS コンストラクトライブラリのインポート
- 依存関係の管理
- AWS CDKTypeScript と他の言語の比較
- AWS CDK での の使用 TypeScript
- AWS CDK での の使用 JavaScript
- Python AWS CDK での の操作
- Java AWS CDK での の使用
- C# AWS CDK での の操作
- Go AWS CDK での の使用

AWS コンストラクトライブラリのインポート

AWS CDK には、 AWS サービス別に整理されたコンストラクトのコレクションであるコンストラクトライブラリが含まれています AWS 。ライブラリの安定したコンストラクトは、TypeScriptパッケージ名 によって呼び出される単一のモジュールで提供されますaws-cdk-lib。実際のパッケージ名は言語によって異なります。

TypeScript

インストール	`npm ###### aws-cdk-lib`
[Import] (インポート)	`const cdk = require('aws-cdk-lib'),`

JavaScript

インストール	`npm ###### aws-cdk-lib`
[Import] (インポート)	`const cdk = require('aws-cdk-lib'),`

Python

インストール	`Python -m pip install aws-cdk-lib`
[Import] (インポート)	`aws_cdk # cdk ##########`

Java

を に追加する `pom.xml`	Group `software.amazon.awscdk` ; artifact `aws-cdk-lib`
[Import] (インポート)	`software.amazon.awscdk.App ##### #####` (for example)

C#

インストール	`dotnet ####### Amazon.CDK.Lib`
[Import] (インポート)	`Amazon.CDK ###`

`construct` 基本クラスとサポートコードは `constructs`モジュールにあります。API がまだ改良中の実験コンストラクトは、個別のモジュールとして分散されます。

AWS CDK API リファレンス

AWS CDK API リファレンスは、ライブラリ内のコンストラクト (およびその他のコンポーネント) の詳細なドキュメントを提供します。API リファレンスのバージョンは、サポートされているプログラミング言語ごとに提供されます。

各モジュールのリファレンスマテリアルは、次のセクションに分かれています。

- 概要: の概念や例など AWS CDK、 のサービスを使用する際に知っておく必要がある入門資料。
- 構築: 1 つ以上の具体的な AWS リソースを表すライブラリクラス。これらは「キュレーション」(L2) リソースまたはパターン (L3 リソース) で、高レベルのインターフェイスと正常なデフォルトを提供します。
- クラス: モジュール内のコンストラクトで使用される機能を提供する非コンストラクトクラス。
- 構造体: プロパティ (コンストラクトのprops引数) やオプションなどの複合値の構造を定義するデータ構造 (属性バンドル) 。
- インターフェイス: 名前がすべて「I」で始まるインターフェイスは、対応するコンストラクトまたは他のクラスの絶対最小機能を定義します。CDK は、コンストラクトインターフェイスを使用して、 AWS CDK アプリの外部で定義され、 などのメソッドによって参照される AWS リソースを表しますBucket.fromBucketArn()。
- 列挙型: 特定のコンストラクトパラメータの指定に使用する名前付き値のコレクション。列挙値を使用すると、CDK は合成中にこれらの値の有効性を確認できます。
- CloudFormation リソース： 名前が「Cfn」で始まるこれらの L1 コンストラクトは、CloudFormation 仕様で定義されているリソースを正確に表します。CDK リリースごとに、その仕様から自動的に生成されます。各 L2 または L3 コンストラクトは、1 つ以上の CloudFormationリソースをカプセル化します。
- CloudFormation プロパティタイプ: 各 CloudFormation リソースのプロパティを定義する名前付き値のコレクション。

コンストラクトクラスと比較したインターフェイス

では、プログラミングの概念としてインターフェイスに精通していても、明確ではない可能性のある特定の方法でインターフェイス AWS CDK を使用します。

は、 などのメソッドを使用した CDK アプリケーション外部で定義されたリソースの使用 AWS CDK をサポートしますBucket.fromBucketArn()。外部リソースは変更できず、 Bucket クラスなどを使用して CDK アプリで定義されたリソースで利用できるすべての機能がない可能性があります。次に、インターフェイスは、外部 AWS リソースを含む特定のリソースタイプについて CDK で使用できる最低限の機能を表します。

CDK アプリでリソースをインスタンス化するときは、常に などの具体的なクラスを使用する必要がありますBucket。独自のコンストラクトのいずれかで受け入れる引数のタイプを指定するときは、外部リソースを処理する準備ができIBucketている場合 (つまり、変更する必要はありません)

など、インターフェイスタイプを使用します。CDK 定義のコンストラクトが必要な場合は、使用できる最も一般的なタイプを指定します。

一部のインターフェイスは、コンストラクトではなく、特定のクラスに関連付けられたプロパティまたはオプションバンドルの最小バージョンです。このようなインターフェイスは、親クラスに渡す引数を受け入れるようにサブクラスする場合に役立ちます。追加のプロパティが 1 つ以上必要な場合は、このインターフェイスまたはより具体的なタイプを実装または取得する必要があります。

> **Note**
>
> でサポートされている一部のプログラミング言語には、インターフェイス機能 AWS CDK がありません。これらの言語では、インターフェイスは通常のクラスにすぎません。名前で識別できます。名前は、最初の「I」のパターンに続いて、他のコンストラクトの名前 (例: `IBucket`) が続きます。同じルールが適用されます。

依存関係の管理

AWS CDK アプリケーションまたはライブラリの依存関係は、パッケージ管理ツールを使用して管理されます。これらのツールは通常、プログラミング言語で使用されます。

通常、言語 AWS CDK の標準または公式のパッケージ管理ツールがある場合は、 でサポートされます。それ以外の場合、 AWS CDK は言語の最も人気のある言語または広くサポートされている言語をサポートします。特にサポートされているツールで作業する場合は、他のツールを使用することもできます。ただし、他のツールの公式サポートは限られています。

では、次のパッケージマネージャー AWS CDK がサポートされています。

言語	サポートされているパッケージ管理ツール
TypeScript/JavaScript	NPM (ノードパッケージマネージャー) または Yarn
Python	PIP (Python 用パッケージインストーラ)
Java	Maven
C#	NuGet

言語	サポートされているパッケージ管理ツール
Go	Go モジュール

`cdk init` コマンドを使用して新しいプロジェクトを作成すると AWS CDK CLI、CDK コアライブラリと安定したコンストラクトの依存関係が自動的に指定されます。

サポートされているプログラミング言語の依存関係の管理の詳細については、以下を参照してください。

- での依存関係の管理 TypeScript.
- での依存関係の管理 JavaScript.
- での依存関係の管理 Python.
- での依存関係の管理 Java.
- での依存関係の管理 C#.
- での依存関係の管理 Go.

AWS CDKTypeScript と他の言語の比較

TypeScript は、 AWS CDK アプリケーションの開発でサポートされている最初の言語です。したがって、大量の CDK コード例が で記述されています TypeScript。別の言語で開発している場合は、選択した言語 TypeScript と比較して での AWS CDK コードの実装方法を比較すると役立つ場合があります。これは、ドキュメント全体で例を使用するのに役立ちます。

モジュールのインポート

TypeScript/JavaScript

TypeScript では、名前空間全体、または名前空間からの個々のオブジェクトのインポートがサポートされています。各名前空間には、特定の AWS サービスで使用するコンストラクトやその他のクラスが含まれます。

```
// Import main CDK library as cdk
import * as cdk from 'aws-cdk-lib';   // ES6 import preferred in TS
const cdk = require('aws-cdk-lib');   // Node.js require() preferred in JS

// Import specific core CDK classes
```

```
import { Stack, App } from 'aws-cdk-lib';
const { Stack, App } = require('aws-cdk-lib');

// Import AWS S3 namespace as s3 into current namespace
import { aws_s3 as s3 } from 'aws-cdk-lib';   // TypeScript
const s3 = require('aws-cdk-lib/aws-s3');     // JavaScript

// Having imported cdk already as above, this is also valid
const s3 = cdk.aws_s3;

// Now use s3 to access the S3 types
const bucket = s3.Bucket(...);

// Selective import of s3.Bucket
import { Bucket } from 'aws-cdk-lib/aws-s3';        // TypeScript
const { Bucket } = require('aws-cdk-lib/aws-s3');   // JavaScript

// Now use Bucket to instantiate an S3 bucket
const bucket = Bucket(...);
```

Python

と同様に TypeScript、Python は名前空間モジュールのインポートと選択的インポートをサポートしています。Python の名前空間は aws_cdk.xxx のようになります。xxx は Amazon S3 の s3 などの AWS サービス名を表します。(これらの例では Amazon S3 を使用しています)。

```
# Import main CDK library as cdk
import aws_cdk as cdk

# Selective import of specific core classes
from aws_cdk import Stack, App

# Import entire module as s3 into current namespace
import aws_cdk.aws_s3 as s3

# s3 can now be used to access classes it contains
bucket = s3.Bucket(...)

# Selective import of s3.Bucket into current namespace
from aws_cdk.s3 import Bucket

# Bucket can now be used to instantiate a bucket
```

```
bucket = Bucket(...)
```

Java

Java のインポートは、 TypeScriptのインポートとは動作が異なります。各インポートステートメントは、特定のパッケージから単一のクラス名、またはそのパッケージで定義されているすべてのクラスをインポートします (を使用*) 。クラスには、インポートされている場合はクラス名自体、またはそのパッケージを含む修飾クラス名を使用してアクセスできます。

ライブラリは AWS 、 コンストラクトライブラリsoftware.amazon.awscdk.services.xxxの のように名前が付けられています (メインライブラリは ですsoftware.amazon.awscdk) 。 AWS CDK パッケージの Maven グループ ID は ですsoftware.amazon.awscdk。

```
// Make certain core classes available
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.App;

// Make all Amazon S3 construct library classes available
import software.amazon.awscdk.services.s3.*;

// Make only Bucket and EventType classes available
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that
 name
// does not conflict with another class)
Bucket bucket = Bucket.Builder.create(...).build();

// We can always use the qualified name of a class (including its package) even
 without an
// import directive
software.amazon.awscdk.services.s3.Bucket bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();

// Java 10 or later can use var keyword to avoid typing the type twice
var bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();
```

C#

C# では、 using ディレクティブを使用してタイプをインポートします。2 つのスタイルがあります。1 つの では、プレーン名を使用して、指定された名前空間のすべてのタイプにアクセスできます。もう 1 つは、エイリアスを使用して名前空間自体を参照できます。

パッケージの名前は AWS 、 構成ライブラリパッケージAmazon.CDK.AWS.xxxの のような名前です。(コアモジュールは ですAmazon.CDK。)

```
// Make CDK base classes available under cdk
using cdk = Amazon.CDK;

// Make all Amazon S3 construct library classes available
using Amazon.CDK.AWS.S3;

// Now we can access any S3 type using its name
var bucket = new Bucket(...);

// Import the S3 namespace under an alias
using s3 = Amazon.CDK.AWS.S3;

// Now we can access an S3 type through the namespace alias
var bucket = new s3.Bucket(...);

// We can always use the qualified name of a type (including its namespace) even
 without a
// using directive
var bucket = new Amazon.CDK.AWS.S3.Bucket(...)
```

Go

各 AWS Construct Library モジュールは Go パッケージとして提供されます。

```
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

// now instantiate a bucket
bucket := awss3.NewBucket(...)
```

```
// use aliases for brevity/clarity
import (
    cdk "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    s3  "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

bucket := s3.NewBucket(...)
```

コンストラクトのインスタンス化

AWS CDK コンストラクトクラスの名前は、サポートされているすべての言語で同じです。ほとんどの言語では、 newキーワードを使用してクラスをインスタンス化しています (Python と Go は使用しません)。また、ほとんどの言語では、 キーワードは現在のインスタンスthisを参照します。(Python は 規則selfに従って を使用します)。現在のインスタンスへの参照を、作成するすべてのコンストラクトの scope パラメータとして渡す必要があります。

AWS CDK コンストラクトの 3 番目の引数は です。これはprops、コンストラクトの構築に必要な属性を含むオブジェクトです。この引数はオプションでもかまいませんが、必要な場合は、サポートされている言語がそれをイディオマティックな方法で処理します。属性の名前は、言語の標準的な命名パターンにも適合します。

TypeScript/JavaScript

```
// Instantiate default Bucket
const bucket = new s3.Bucket(this, 'MyBucket');

// Instantiate Bucket with bucketName and versioned properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
   versioned: true,
});

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  websiteRedirect: {host: 'aws.amazon.com'}});
```

Python

Python は、クラスのインスタンス化時にnewキーワードを使用しません。properties 引数はキーワード引数を使用して表され、引数は を使用して名前が付けられますsnake_case。

props 値がそれ自体が属性のバンドルである場合、サブプロパティのキーワード引数を受け入れる プロパティにちなんだ という名前のクラスによって表されます。

Python では、現在のインスタンスは、規則selfによって名前が付けられた最初の引数として メソッドに渡されます。

```
# Instantiate default Bucket
bucket = s3.Bucket(self, "MyBucket")

# Instantiate Bucket with bucket_name and versioned properties
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket", versioned=true)

# Instantiate Bucket with website_redirect, which has its own sub-properties
bucket = s3.Bucket(self, "MyBucket", website_redirect=s3.WebsiteRedirect(
            host_name="aws.amazon.com"))
```

Java

Java では、props 引数は という名前のクラスで表されます XxxxProps (たとえば、Bucketコンストラクトのprops BucketPropsの場合は)。props 引数は、ビルダーパターンを使用して構築します。

各XxxxPropsクラスにはビルダーがあります。また、次の例に示すように、1 つのステップで props と コンストラクトを構築する各コンストラクトに便利なビルダーもあります。

Props の名前は、 を使用して TypeScriptと同じですcamelCase。

```
// Instantiate default Bucket
Bucket bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with bucketName and versioned properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .bucketName("my-bucket").versioned(true)
                      .build();

# Instantiate Bucket with websiteRedirect, which has its own sub-properties
```

```
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
                      .websiteRedirect(new websiteRedirect.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

C# では、props は という名前のクラスへのオブジェクトイニシャライザを使用して指定されます XxxxProps (たとえば、Bucketコンストラクトのprops BucketPropsの場合) 。

Props の名前は、 を使用する場合を除き TypeScript、 と似ていますPascalCase。

コンストラクトをインスタンス化するときに varキーワードを使用する方が便利なため、クラス名を 2 回入力する必要はありません。ただし、ローカルコードスタイルガイドは異なる場合があります。

```
// Instantiate default Bucket
var bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with BucketName and Versioned properties
var bucket =  Bucket(self, "MyBucket", new BucketProps {
                      BucketName = "my-bucket",
                      Versioned  = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
var bucket = Bucket(self, "MyBucket", new BucketProps {
                      WebsiteRedirect = new WebsiteRedirect {
                              HostName = "aws.amazon.com"
                      }});
```

Go

Go でコンストラクトを作成するには、 関数を呼び出します。NewXxxxxxここで、 Xxxxxxxはコンストラクトの名前です。コンストラクトのプロパティは構造体として定義されます。

Go では、数値、ブール値、文字列などの値を含むすべてのコンストラクトパラメータがポインタです。などの便利な関数jsii.Stringを使用して、これらのポインタを作成します。

```
// Instantiate default Bucket
bucket := awss3.NewBucket(stack, jsii.String("MyBucket"), nil)

// Instantiate Bucket with BucketName and Versioned properties
```

```
bucket1 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 BucketName: jsii.String("my-bucket"),
 Versioned:  jsii.Bool(true),
})

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
bucket2 := awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
 WebsiteRedirect: &awss3.RedirectTarget{
  HostName: jsii.String("aws.amazon.com"),
 }})
```

メンバーへのアクセス

コンストラクトやその他の AWS CDK クラスの属性やプロパティを参照し、これらの値を例えば他のコンストラクトを構築するための入力として使用するのが一般的です。前述の メソッドの命名の違いは、ここでも適用されます。さらに、Java では、メンバーに直接アクセスすることはできません。代わりに、ゲッターメソッドが提供されます。

TypeScript/JavaScript

名前は ですcamelCase。

```
bucket.bucketArn
```

Python

名前は ですsnake_case。

```
bucket.bucket_arn
```

Java

プロパティごとに getter メソッドが用意されています。これらの名前は ですcamelCase。

```
bucket.getBucketArn()
```

C#

名前は ですPascalCase。

```
bucket.BucketArn
```

Go

名前は ですPascalCase。

```
bucket.BucketArn
```

列挙定数

列挙定数はクラスに限定され、すべての言語 (別名) にアンダースコアを含む大文字の名前がありますSCREAMING_SNAKE_CASE。クラス名でも Go を除くサポートされているすべての言語で同じ大文字と小文字が使用されるため、修飾列挙型名もこれらの言語で同じです。

```
s3.BucketEncryption.KMS_MANAGED
```

Go では、列挙定数はモジュール名前空間の属性であり、次のように記述されます。

```
awss3.BucketEncryption_KMS_MANAGED
```

オブジェクトインターフェイス

AWS CDK は TypeScript 、オブジェクトインターフェイスを使用して、クラスが予想されるメソッドとプロパティのセットを実装していることを示します。オブジェクトインターフェイスの名前は で始まるため、オブジェクトインターフェイスを認識できますI。具体的なクラスは、implementsキーワードを使用して実装するインターフェイスを示します。

TypeScript/JavaScript

> **Note**
>
> JavaScript にはインターフェイス機能はありません。implements キーワードとその後に続くクラス名は無視してかまいません。

```
import { IAspect, IConstruct } from 'aws-cdk-lib';
```

```
class MyAspect implements IAspect {
  public visit(node: IConstruct) {
    console.log('Visited', node.node.path);
  }
}
```

Python

Python にはインターフェイス機能はありません。ただし、 では AWS CDK 、クラスを で修飾することで、インターフェイスの実装を示すことができます`@jsii.implements(interface)`。

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
  def visit(self, node: IConstruct) -> None:
    print("Visited", node.node.path)
```

Java

```
import software.amazon.awscdk.IAspect;
import software.amazon.awscdk.IConstruct;

public class MyAspect implements IAspect {
    public void visit(IConstruct node) {
        System.out.format("Visited %s", node.getNode().getPath());
    }
}
```

C#

```
using Amazon.CDK;

public class MyAspect : IAspect
{
    public void Visit(IConstruct node)
    {
        System.Console.WriteLine($"Visited ${node.Node.Path}");
    }
```

```
}
```

Go

Go 構造体は、実装するインターフェイスを明示的に宣言する必要はありません。Go コンパイラは、 構造で使用できるメソッドとプロパティに基づいて実装を決定します。例えば、次のコードでは、 は コンストラクトを使用するVisitメソッドを提供するため、 IAspectインターフェイスMyAspectを実装しています。

```
type MyAspect struct {
}

func (a MyAspect) Visit(node constructs.IConstruct) {
 fmt.Println("Visited", *node.Node().Path())
}
```

AWS CDK での の使用 TypeScript

TypeScript は、 で完全にサポートされているクライアント言語 AWS Cloud Development Kit (AWS CDK) であり、安定していると見なされます。 AWS CDK で を操作するには、Microsoft の TypeScript コンパイラ (`tsc`) 、Node.js、Node Package Manager () など、使い慣れたツール TypeScript を使用しますnpm。このガイドの例では NPM を使用していますが、必要に応じて Yarn を使用することもできます。Construct Library AWS で構成されるモジュールは、NPM リポジトリ npmjs.org を介して配布されます。

任意のエディタまたは IDE を使用できます。多くの AWS CDK デベロッパーは、Visual Studio Code (または同等のオープンソースVSCodium) を使用します。これは、 を優れたサポートを提供します TypeScript。

トピック

- の開始方法 TypeScript
- 「プロジェクトの作成」
- ローカル tscと の使用 cdk
- AWS コンストラクトライブラリモジュールの管理
- での依存関係の管理 TypeScript
- AWS CDK での idioms TypeScript
- 構築、合成、デプロイ

の開始方法 TypeScript

を使用するには AWS CDK、 AWS アカウントと認証情報が必要であり、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

また、 TypeScript それ自体 (バージョン 3.8 以降) も必要です。まだ持っていない場合は、 を使用してインストールできますnpm。

```
npm install -g typescript
```

> **Note**
>
> アクセス許可エラーが発生し、システムで管理者アクセス権がある場合は、 を試してくださいsudo npm install -g typescript。

通常の を TypeScript 最新の状態に保つnpm update -g typescript。

> **Note**
>
> サードパーティー言語の廃止: 言語バージョンは、ベンダーまたはコミュニティによって共有される EOL (サポート終了) までのみサポートされ、事前通知により変更される可能性があります。

「プロジェクトの作成」

空のディレクトリcdk initで を呼び出して、新しい AWS CDK プロジェクトを作成します。--language オプションを使用して、 を指定しますtypescript。

```
mkdir my-project
cd my-project
cdk init app --language typescript
```

プロジェクトを作成すると、aws-cdk-libモジュールとその依存関係もインストールされます。

cdk init は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換さ

れます。ただし、名前は TypeScript 識別子の形式に従う必要があります。例えば、数字で開始したり、スペースを含めたりしないでください。

ローカル tscと の使用 cdk

ほとんどの場合、このガイドでは、 TypeScript と CDK Toolkit をグローバルにインストールし (npm install -g typescript aws-cdk)、提供されているコマンド例 (などcdk synth) がこの前提に従うことを前提としています。このアプローチにより、両方のコンポーネントを最新の状態に保つことが容易になります。両方のコンポーネントが下位互換性に対して厳格なアプローチを採用するため、常に最新バージョンを使用するリスクは一般的にほとんどありません。

一部の チームでは、 TypeScript コンパイラや CDK Toolkit などのツールを含め、各プロジェクト内のすべての依存関係を指定したいと考えています。この方法により、これらのコンポーネントを特定のバージョンに固定し、チーム (および CI/CD 環境) のすべてのデベロッパーがそれらのバージョンを正確に使用できるようになります。これにより、変更の原因がなくなり、ビルドとデプロイの一貫性と繰り返し性が向上します。

CDK には TypeScript 、プロジェクトテンプレートの に TypeScript と CDK Toolkit の両方の依存関係が含まれているためpackage.json、このアプローチを使用する場合は、プロジェクトを変更する必要はありません。必要なのは、アプリケーションの構築とコマンドの発行に少し異なるcdkコマンドを使用することだけです。

操作	グローバルツールを使用する	ローカルツールを使用する
プロジェクトを初期化する	`cdk init --language typescript`	`npx aws-cdk init --language typescript`
ビルド	`tsc`	`npm run build`
CDK Toolkit コマンドを実行する	`cdk ...`	`npm run cdk ...` or `npx aws-cdk ...`

npx aws-cdk は、現在のプロジェクトにローカルにインストールされた CDK Toolkit のバージョンが存在する場合は、グローバルインストールにフォールバックします。グローバルインストールが存在しない場合、 は CDK Toolkit の一時コピーnpxをダウンロードして実行します。@ 構文を使用して任意のバージョンの CDK Toolkit を指定できます。 は をnpx aws-cdk@2.0 --version出力します2.0.0。

> **Tip**
>
> ローカル CDK Toolkit インストールで cdk コマンドを使用できるように、エイリアスを設定します。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

AWS コンストラクトライブラリモジュールの管理

Node Package Manager (npm) を使用して、アプリケーションで使用する Construct Library AWS モジュールと、必要なその他のパッケージをインストールして更新します。(npm必要に応じて yarnの代わりに を使用できます。) は、それらのモジュールの依存関係も自動的にnpmインストールします。

ほとんどの AWS CDK コンストラクトは、 という名前のメイン CDK パッケージにあります。これはaws-cdk-lib、 によって作成された新しいプロジェクトのデフォルトの依存関係ですcdk init。「実験 AWS 」 上位レベルのコンストラクトがまだ開発中のライブラリモジュールの構築は、 のように名前が付けられます@aws-cdk/*SERVICE-NAME*-alpha。サービス名には aws- プレフィックスが付いています。モジュールの名前がわからない場合は、NPM で検索します。

> **Note**
>
> CDK API リファレンスにはパッケージ名も表示されます。

例えば、以下のコマンドは の実験モジュールをインストールします AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

一部の サービスの コンストラクトライブラリのサポートは、複数の名前空間にあります。例えば、以外にもaws-route53、Amazon Route 53 名前空間 、aws-route53-targets、aws-route53-patternsおよび が 3 つ追加されていますaws-route53resolver。

プロジェクトの依存関係は で管理されますpackage.json。このファイルを編集して、依存関係の一部またはすべてを特定のバージョンにロックしたり、特定の条件下で新しいバージョンに更新したりできます。で指定したルールに従って、プロジェクトの NPM 依存関係を最新の許可されたバージョンに更新するにはpackage.json :

```
npm update
```

では TypeScript、NPM を使用してインストールする場合と同じ名前でモジュールをコードにインポートします。アプリケーションに AWS CDK クラスをインポートしたり、ライブラリモジュール AWS を構築したりする場合は、次のプラクティスをお勧めします。これらのガイドラインに従うことで、コードを他の AWS CDK アプリケーションと一致させ、理解しやすくなります。

- ではなく、ES6-styleimportディレクティブを使用しますrequire()。
- 通常、個々のクラスを からインポートしますaws-cdk-lib。

  ```
  import { App, Stack } from 'aws-cdk-lib';
  ```

- から多数のクラスが必要な場合はaws-cdk-lib、個々のクラスをインポートcdkする代わりに、の名前空間エイリアスを使用できます。両方を実行しないでください。

  ```
  import * as cdk from 'aws-cdk-lib';
  ```

- 通常、短い名前空間エイリアスを使用して AWS サービスコンストラクトをインポートします。

  ```
  import { aws_s3 as s3 } from 'aws-cdk-lib';
  ```

での依存関係の管理 TypeScript

TypeScript CDK プロジェクトでは、依存関係はプロジェクトのメインディレクトリの package.json ファイルで指定されます。コア AWS CDK モジュールは、 という 1 つのNPMパッケージにありますaws-cdk-lib。

を使用してパッケージをインストールするとnpm install、NPM はパッケージを package.json に記録します。

必要に応じて、NPM の代わりに Yarn を使用できます。ただし、CDK は Yarn 2 plug-and-play のデフォルトモードである Yarn モードをサポートしていません。以下をプロジェクトの `.yarnrc.yml` ファイルに追加して、この機能をオフにします。

```
nodeLinker: node-modules
```

CDK アプリケーション

`cdk init --language typescript` コマンドによって生成される`package.json`ファイルの例を次に示します。

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

デプロイ可能な CDK アプリケーションの場合、 の `dependencies`セクションで を指定`aws-cdk-lib`する必要がありま`package.json`す。キャレット (^) バージョン番号指定子を使用すると、同じ

メジャーバージョン内にある限り、指定したバージョンよりも新しいバージョンを受け入れるように指定できます。

実験的なコンストラクトの場合は、変更される可能性のある APIs を持つアルファコンストラクトライブラリモジュールの正確なバージョンを指定します。これらのモジュールの新しいバージョンでは、アプリが破損する API の変更が発生する可能性があるため、^ または ~ を使用しないでください。

の devDependenciesセクションで、アプリケーションのテストに必要なライブラリとツールのバージョン (jestテストフレームワークなど) を指定しますpackage.json。オプションで、^ を使用して、新しい互換性のあるバージョンが許容可能であることを指定します。

サードパーティーのコンストラクトライブラリ

コンストラクトライブラリを開発している場合は、次のpackage.jsonサンプルファイルに示すように、 セクションpeerDependenciesと devDependenciesセクションを組み合わせて依存関係を指定します。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

ではpeerDependencies、キャレット (^) を使用して、ライブラリが動作aws-cdk-libする の最小バージョンを指定します。これにより、さまざまな CDK バージョンとのライブラリの互換性が最大化されます。変更される可能性のある APIs を含む alpha construct ライブラリモジュールの正確なバージョンを指定します。peerDependencies を使用すると、ツリー内のすべての CDK ライブラリのコピーが 1 node_modules つだけになります。

でdevDependencies、テストに必要なツールとライブラリを指定します。オプションで ^ を使用して、それ以降の互換性のあるバージョンが許容できることを示します。ライブラリの互換性をアドバタイズする aws-cdk-libおよびその他の CDK パッケージの最小バージョンを正確に (^ または ~ なし) 指定します。この方法により、テストがそれらのバージョンに対して実行されるようになります。これにより、新しいバージョンでのみ見つかった機能を誤って使用すると、テストでその機能をキャッチできます。

> **Warning**
>
> peerDependencies は NPM 7 以降によってのみ自動的にインストールされます。NPM 6 以前を使用している場合、または Yarn を使用している場合は、依存関係の依存関係を に含める必要がありますdevDependencies。そうしないと、インストールされず、未解決のピア依存関係に関する警告が表示されます。

依存関係のインストールと更新

次のコマンドを実行して、プロジェクトの依存関係をインストールします。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

インストールされているモジュールを更新するには、前述の npm installおよび yarn upgrade コマンドを使用できます。どちらのコマンドも、 のパッケージnode_modulesを、 のルールを満たす最新バージョンに更新しますpackage.json。ただし、それらpackage.json自体は更新されません。新しい最小バージョンを設定することもできます。でパッケージをホストする場合 GitHub、 を自動

的に更新するように Dependabot のバージョン更新を設定できますpackage.json。または、npm-check-updates を使用します。

> **Important**
>
> 設計上、依存関係をインストールまたは更新する場合、NPM と Yarn は、 で指定された要件を満たすすべてのパッケージの最新バージョンを選択しますpackage.json。これらのバージョンが (誤ってまたは意図的に) 壊れるリスクは常に存在します。プロジェクトの依存関係を更新した後、徹底的にテストします。

AWS CDK での idioms TypeScript

プロンプト

すべての AWS Construct Library クラスは、コンストラクトが定義されているスコープ (コンストラクトツリーの親) 、ID 、props の 3 つの引数を使用してインスタンス化されます。引数プロパティは、コンストラクトが作成する AWS リソースの設定に使用するキーと値のペアのバンドルです。他のクラスやメソッドも引数に「属性のバンドル」パターンを使用します。

では TypeScript、 の形状propsは、必須およびオプションの引数とそのタイプを示すインターフェイスを使用して定義されます。このようなインターフェイスは、引props数の種類ごとに定義され、通常は単一のコンストラクトまたはメソッドに固有です。例えば、バケットコンストラクト (内aws-cdk-lib/aws-s3 module) は、 BucketPropsインターフェイスに準拠するprops引数を指定します。

プロパティがそれ自体がオブジェクトである場合、例えば の websiteRedirect プロパティBucketPropsの場合、そのオブジェクトには、その形状が準拠する必要がある独自のインターフェイスがあります。この場合は ですRedirectTarget。

AWS Construct Library クラスをサブクラスする (または、props のような引数を取るメソッドを上書きする) 場合は、既存のインターフェイスから継承して、コードに必要な新しいpropsを指定する新しいインターフェイスを作成できます。親クラスまたは基本メソッドを呼び出す場合、通常、受信した props 引数全体を渡すことができます。これは、 オブジェクトで指定された属性は無視されますが、インターフェイスで指定されていないためです。

の将来のリリースでは、独自のプロパティに使用した名前で新しいプロパティが同時に追加 AWS CDK される可能性があります。継承チェーンで受け取った値を渡すと、予期しない動作が発生する

可能性があります。プロパティを削除または に設定して、受信したプロパティの浅いコピーを渡す方が安全ですundefined。例:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

または、プロパティに名前を付けて、プロパティがコンストラクトに属していることを明確にします。このようにして、将来の AWS CDK リリースでプロパティと衝突する可能性はほとんどありません。それらが多数ある場合は、適切に名前が付けられた単一の オブジェクトを使用してそれらを保持します。

欠落した値

オブジェクト (props など) の欠損値は、 undefinedの値を持ちます TypeScript。言語のバージョン 3.7 では、これらの値の操作を簡素化する演算子が導入されました。これにより、未定義の値に達したときに、デフォルトと「短い回路」チェーンを簡単に指定できます。これらの機能の詳細については、TypeScript 「3.7 リリースノート」、特に「オプションの連鎖と Nullish Coalescing」の 2 つの機能を参照してください。

構築、合成、デプロイ

通常、アプリケーションを構築して実行するときは、プロジェクトのルートディレクトリにいる必要があります。

Node.js TypeScript を直接実行することはできません。代わりに、コンパイラ JavaScript を使用して TypeScriptアプリケーションを に変換しますtsc。その後、結果の JavaScript コードが実行されます。

は、アプリの実行に必要なたびに、これ AWS CDK を自動的に行います。ただし、エラーを確認してテストを実行するには、手動でコンパイルすると便利です。 TypeScript アプリを手動でコンパイルするには、 を発行しますnpm run build。視聴モードに入るnpm run watchと問題が発生することもあります。この TypeScriptモードでは、ソースファイルに変更を保存するたびに、コンパイラが自動的にアプリを再構築します。

AWS CDK アプリケーションで定義されたスタックは、合成して個別にデプロイすることも、以下のコマンドを使用してまとめてデプロイすることもできます。通常、プロジェクトを発行するときは、プロジェクトのメインディレクトリにいる必要があります。

- cdk synth: AWS CDK アプリケーションの 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。

- `cdk deploy`: AWS CDK アプリケーション内の 1 つ以上のスタックによって定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

また、ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。そうしないと、シェルは AWS CDK Toolkit に渡す前に、現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> デプロイする前にスタックを明示的に合成する必要はありません。 はこのステップ`cdk deploy`を実行して、最新のコードがデプロイされることを確認します。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

AWS CDK での の使用 JavaScript

JavaScript は、 で完全にサポートされているクライアント言語 AWS CDK であり、安定していると見なされます。 AWS Cloud Development Kit (AWS CDK) で を操作するには、Node.js や Node Package Manager (npm) など、使い慣れたツール JavaScript を使用します。このガイドの例では NPM を使用していますが、必要に応じて Yarn を使用することもできます。 AWS コンストラクトライブラリで構成されるモジュールは、NPM リポジトリ npmjs.org を介して配布されます。

任意の工ディタまたは IDE を使用できます。多くの AWS CDK デベロッパーは、Visual Studio Code (または同等のオープンソースVSCodium) を使用します。これは、 をうまくサポートしています JavaScript。

トピック

JavaScript の開始方法

を使用するには AWS CDK、 AWS アカウントと認証情報が必要です。また、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

JavaScript AWS CDK アプリケーションでは、これら以外の追加の前提条件は必要ありません。

> **Note**
>
> サードパーティーの言語の廃止: 言語バージョンは、ベンダーまたはコミュニティによって共有される EOL (サポート終了) までのみサポートされ、事前通知により変更される可能性があります。

「プロジェクトの作成」

空のディレクトリcdk initで を呼び出して、新しい AWS CDK プロジェクトを作成します。--language オプションを使用して、 を指定しますjavascript。

```
mkdir my-project
cd my-project
cdk init app --language javascript
```

プロジェクトを作成すると、aws-cdk-libモジュールとその依存関係もインストールされます。

`cdk init` は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換されます。ただし、名前は識別子の形式に従う必要があります JavaScript 。例えば、数字で開始したり、スペースを含めたりしないでください。

ローカル の使用 cdk

ほとんどの場合、このガイドでは CDK Toolkit をグローバルにインストールし (`npm install -g aws-cdk`)、提供されているコマンド例 (など`cdk synth`) がこの前提に従うことを前提としています。このアプローチにより、CDK Toolkit を最新の状態に保つことが容易になり、CDK は下位互換性に対して厳格なアプローチを採用するため、常に最新バージョンを使用するリスクは一般的にほとんどありません。

一部の チームでは、CDK Toolkit などのツールを含め、各プロジェクト内のすべての依存関係を指定したいと考えています。この方法により、このようなコンポーネントを特定のバージョンに固定し、チーム (および CI/CD 環境) のすべてのデベロッパーがそれらのバージョンを正確に使用できるようになります。これにより、変更の原因がなくなり、ビルドとデプロイの一貫性と繰り返し性が向上します。

CDK には、 JavaScript プロジェクトテンプレートの に CDK Toolkit の依存関係が含まれているため`package.json`、このアプローチを使用する場合は、プロジェクトを変更する必要はありません。必要なのは、アプリケーションの構築とコマンドの発行に少し異なる`cdk`コマンドを使用することだけです。

操作	グローバル CDK ツールキットを使用する	ローカル CDK ツールキットを使用する
プロジェクトを初期化する	`cdk init --language javascript`	`npx aws-cdk init --language javascript`
CDK Toolkit コマンドを実行する	`cdk ...`	`npm ## cdk ...` or `npx aws-cdk ...`

`npx aws-cdk` は、現在のプロジェクトにローカルにインストールされた CDK Toolkit のバージョンが存在する場合は、グローバルインストールにフォールバックします。グローバルインストールが存在しない場合、 は CDK Toolkit の一時コピー`npx`をダウンロードして実行します。`@` 構文を使用して任意のバージョンの CDK Toolkit を指定できます。 は を`npx aws-cdk@1.120 --version`出力します`1.120.0`。

> **Tip**
>
> ローカル CDK Toolkit インストールで cdk コマンドを使用できるように、エイリアスを設定します。
>
> macOS/Linux
>
> ```
> alias cdk="npx aws-cdk"
> ```
>
> Windows
>
> ```
> doskey cdk=npx aws-cdk $*
> ```

AWS コンストラクトライブラリモジュールの管理

Node Package Manager (npm) を使用して、アプリケーションで使用する Construct Library AWS モジュールと、必要なその他のパッケージをインストールして更新します。(npm必要に応じて yarnの代わりに を使用できます。) は、それらのモジュールの依存関係npmも自動的にインストールします。

ほとんどの AWS CDK コンストラクトは、 という名前のメイン CDK パッケージにあります。これはaws-cdk-lib、 によって作成された新しいプロジェクトのデフォルトの依存関係ですcdk init。「実験 AWS 」 上位レベルのコンストラクトがまだ開発中のライブラリモジュールの構築は、 のように名前が付けられますaws-cdk-lib/*SERVICE-NAME*-alpha。サービス名には aws- プレフィックスが付いています。モジュールの名前がわからない場合は、NPM で検索します。

> **Note**
>
> CDK API リファレンスにはパッケージ名も表示されます。

例えば、以下のコマンドは の実験モジュールをインストールします AWS CodeStar。

```
npm install @aws-cdk/aws-codestar-alpha
```

一部の サービスの コンストラクトライブラリのサポートは、複数の名前空間にあります。例えば、以外にもaws-route53、Amazon Route 53 名前空間 、aws-route53-targets、aws-route53-patternsおよび が 3 つ追加されていますaws-route53resolver。

プロジェクトの依存関係は で管理されますpackage.json。このファイルを編集して、依存関係の一部またはすべてを特定のバージョンにロックしたり、特定の条件下で新しいバージョンに更新したりできます。で指定したルールに従って、プロジェクトの NPM 依存関係を最新の許可されたバージョンに更新するにはpackage.json :

```
npm update
```

では JavaScript、NPM を使用してインストールする場合と同じ名前でモジュールをコードにインポートします。アプリケーションに AWS CDK クラスをインポートしたり、ライブラリモジュール AWS を構築したりする場合は、次のプラクティスをお勧めします。これらのガイドラインに従うことで、コードを他の AWS CDK アプリケーションと一致させ、理解しやすくなります。

- ES6-styleimportディレクティブではなくrequire()、 を使用します。Node.js の古いバージョンは ES6 インポートをサポートしていないため、古い構文を使用すると互換性が増します。(ES6 インポートを実際に使用する場合は、esm を使用して、プロジェクトがサポートされているすべてのバージョンの Node.js と互換性があることを確認してください)。
- 通常、個々のクラスを からインポートしますaws-cdk-lib。

  ```
  const { App, Stack } = require('aws-cdk-lib');
  ```
- から多数のクラスが必要な場合はaws-cdk-lib、個々のクラスをインポートcdkする代わりに、の名前空間エイリアスを使用できます。両方を実行しないでください。

  ```
  const cdk = require('aws-cdk-lib');
  ```
- 通常、短い名前空間エイリアスを使用して AWS 構成ライブラリをインポートします。

  ```
  const { s3 } = require('aws-cdk-lib/aws-s3');
  ```

での依存関係の管理 JavaScript

JavaScript CDK プロジェクトでは、依存関係はプロジェクトのメインディレクトリの package.json ファイルで指定されます。コア AWS CDK モジュールは、 という 1 つのNPMパッケージにありますaws-cdk-lib。

を使用してパッケージをインストールするとnpm install、NPM はパッケージを `package.json` に記録します。

必要に応じて、NPM の代わりに Yarn を使用できます。ただし、CDK は Yarn 2 plug-and-play のデフォルトモードである Yarn モードをサポートしていません。以下をプロジェクトの `.yarnrc.yml` ファイルに追加して、この機能をオフにします。

```
nodeLinker: node-modules
```

CDK アプリケーション

コマンドによって生成されるpackage.jsonファイルの例を次に示しますcdk init --language typescript。に対して生成されるファイルは類似 JavaScript しており、 TypeScript関連のエントリがない場合に限ります。

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

デプロイ可能な CDK アプリケーションの場合、 の dependenciesセクションで を指定aws-cdk-libする必要がありますpackage.json。キャレット (^) バージョン番号指定子を使用すると、同じメジャーバージョン内にある限り、指定したバージョンよりも新しいバージョンを受け入れるように指定できます。

実験的なコンストラクトの場合は、変更される可能性のある APIs を持つアルファコンストラクトライブラリモジュールの正確なバージョンを指定します。これらのモジュールの新しいバージョンでは、アプリが破損する API の変更が発生する可能性があるため、^ または ~ を使用しないでください。

の devDependenciesセクションで、アプリケーションのテストに必要なライブラリとツールのバージョン (jestテストフレームワークなど) を指定しますpackage.json。オプションで、^ を使用して、新しい互換性のあるバージョンが許容可能であることを指定します。

サードパーティーのコンストラクトライブラリ

コンストラクトライブラリを開発している場合は、次のpackage.jsonサンプルファイルに示すように、 セクションpeerDependenciesと devDependenciesセクションを組み合わせて依存関係を指定します。

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

ではpeerDependencies、キャレット (^) を使用して、ライブラリが動作aws-cdk-libする の最小バージョンを指定します。これにより、ライブラリとさまざまな CDK バージョンの互換性が最大化されます。変更される可能性のある APIs を含むアルファコンストラクトライブラリモジュールの

正確なバージョンを指定します。peerDependencies を使用すると、ツリー内のすべての CDK ライブラリのコピーが 1 node_modules つだけになります。

でdevDependencies、テストに必要なツールとライブラリを指定します。オプションで ^ を使用して、それ以降の互換性のあるバージョンが許容できることを示します。ライブラリの互換性をアドバタイズする aws-cdk-libおよびその他の CDK パッケージの最小バージョンを正確に (^ または ~ なし) 指定します。この方法により、テストがそれらのバージョンに対して実行されるようになります。これにより、新しいバージョンでのみ見つかった機能を誤って使用すると、テストでその機能が検出される可能性があります。

> **Warning**
>
> peerDependencies は NPM 7 以降によってのみ自動的にインストールされます。NPM 6 以前を使用している場合、または Yarn を使用している場合は、依存関係の依存関係を に含める必要がありますdevDependencies。そうしないと、インストールされず、未解決のピア依存関係に関する警告が表示されます。

依存関係のインストールと更新

次のコマンドを実行して、プロジェクトの依存関係をインストールします。

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

インストールされているモジュールを更新するには、前述の npm installおよび yarn upgrade コマンドを使用できます。どちらのコマンドも、 のパッケージnode_modulesを のルールを満たす最新バージョンに更新しますpackage.json。ただし、それらpackage.json自体は更新されません。新しい最小バージョンを設定することもできます。でパッケージをホストする場合 GitHub、 を自動的に更新するように Dependabot のバージョン更新を設定できますpackage.json。または、npm-check-updates を使用します。

> **Important**
>
> 設計上、依存関係をインストールまたは更新すると、NPM と Yarn は、 で指定された要件を満たすすべてのパッケージの最新バージョンを選択しますpackage.json。これらのバージョンが (誤ってまたは意図的に) 壊れるリスクは常に存在します。プロジェクトの依存関係を更新した後、徹底的にテストします。

AWS CDK での idioms JavaScript

プロンプト

すべての AWS Construct Library クラスは、3 つの引数を使用してインスタンス化されます。コンストラクトが定義されているスコープ (コンストラクトツリーの親) 、ID 、props は、コンストラクトが作成する AWS リソースの設定に使用するキーと値のペアのバンドルです。他のクラスやメソッドも引数に「属性のバンドル」パターンを使用します。

JavaScript オートコンプリートが適切な IDE またはエディタを使用すると、プロパティ名のスペルミスを回避できます。コンストラクトが encryptionKeysプロパティを期待していて、スペルが の場合encryptionkeys、コンストラクトをインスタンス化するときに、意図した値を渡していません。これにより、 プロパティが必要な場合は合成時にエラーが発生したり、オプションの場合はプロパティがサイレントに無視されたりする可能性があります。後者の場合、上書きしようとしているデフォルトの動作が発生することがあります。ここで特に注意してください。

AWS Construct Library クラスをサブクラスする場合 (または、props のような引数を取るメソッドを上書きする場合) 、独自の使用のために追加のプロパティを受け入れることができます。これらの値は、親クラスまたはオーバーライドされたメソッドでは無視されます。このコードではアクセスされないため、通常、受信したすべてのプロパティを渡すことができます。

の将来のリリースでは、独自のプロパティに使用した名前で新しいプロパティが同時に追加 AWS CDK される可能性があります。継承チェーンで受け取った値を渡すと、予期しない動作が発生する

可能性があります。プロパティを削除または に設定して、受信したプロパティの浅いコピーを渡す方が安全ですundefined。例:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

または、プロパティに名前を付けて、プロパティがコンストラクトに属していることを明確にします。このようにして、将来の AWS CDK リリースでプロパティと衝突する可能性はほとんどありません。それらが多数ある場合は、適切に名前が付けられた単一の オブジェクトを使用してそれらを保持します。

欠落した値

オブジェクトの欠損値 (などprops) には、 の値undefinedがあります JavaScript。これらの処理には、通常の手法が適用されます。例えば、未定義の値のプロパティにアクセスするための一般的なイディオムは次のとおりです。

```
// a may be undefined, but if it is not, it may have an attribute b
// c is undefined if a is undefined, OR if a doesn't have an attribute b
let c = a && a.b;
```

ただし、 以外の「falsy」値がいくつかaある場合はundefined、テストをより明示的なものにすることをお勧めします。ここでは、 nullと undefinedが両方を同時にテストするのと等しいという事実を活用します。

```
let c = a == null ? a : a.b;
```

> **Tip**
>
> Node.js 14.0 以降では、未定義の値の処理を簡素化できる新しい演算子がサポートされています。詳細については、「オプションの連鎖提案と nullish 結合提案」を参照してください。

合成とデプロイ

AWS CDK アプリケーションで定義されたスタックは、合成して個別にデプロイすることも、以下のコマンドを使用してまとめてデプロイすることもできます。通常、プロジェクトを発行するときは、プロジェクトのメインディレクトリにいる必要があります。

- `cdk synth`: AWS CDK アプリケーションの 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。
- `cdk deploy`: AWS CDK アプリケーション内の 1 つ以上のスタックによって定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

また、ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。そうしないと、シェルは AWS CDK Toolkit に渡す前に、現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> デプロイする前にスタックを明示的に合成する必要はありません。 はこのステップ`cdk deploy`を実行して、最新のコードがデプロイされることを確認します。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

で TypeScript の例の使用 JavaScript

TypeScript は、 の開発に使用する言語であり AWS CDK、アプリケーションの開発でサポートされる最初の言語であるため、利用可能な AWS CDK コード例は で記述されています TypeScript。これらのコード例は、 JavaScript デベロッパーにとって適切なリソースです。コードの TypeScript固有の部分を削除するだけで済みます。

TypeScript スニペットでは、新しい ECMAScript `import`と `export`キーワードを使用して他のモジュールからオブジェクトをインポートし、現在のモジュールの外部で使用できるようにオブジェク

トを宣言することがよくあります。Node.js は、これらのキーワードのサポートを最新のリリースで開始しました。使用している Node.js のバージョン (またはサポートしたい) によっては、古い構文を使用するようにインポートとエクスポートを書き換えることができます。

インポートは、 `require()`関数への呼び出しに置き換えることができます。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Bucket, BucketPolicy } from 'aws-cdk-lib/aws-s3';
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { Bucket, BucketPolicy } = require('aws-cdk-lib/aws-s3');
```

エクスポートは`module.exports`オブジェクトに割り当てることができます。

TypeScript

```
export class Stack1 extends cdk.Stack {
  // ...
}

export class Stack2 extends cdk.Stack {
  // ...
}
```

JavaScript

```
class Stack1 extends cdk.Stack {
  // ...
}

class Stack2 extends cdk.Stack {
  // ...
}

module.exports = { Stack1, Stack2 }
```

> **Note**
>
> 古いスタイルのインポートとエクスポートを使用する代わりに、 esmモジュールを使用します。

インポートとエクスポートがソートされたら、実際のコードを調べることができます。一般的に使用される次の TypeScript 機能を使用できます。

- タイプ注釈
- インターフェイス定義
- タイプ変換/キャスト
- アクセス修飾子

変数、クラスメンバー、関数パラメータ、および関数の戻り値の型には、型注釈を指定できます。変数、パラメータ、およびメンバーの場合、タイプはコロンと タイプの識別子に従って指定します。関数の戻り値は関数の署名に従い、コロンと 型で構成されます。

型注釈付きコードを に変換するには JavaScript、コロン と 型を削除します。クラスメンバーは、に何らかの値が必要です JavaScript。 にタイプアノテーションのみundefinedがある場合は、 に設定します TypeScript。

TypeScript

```
var encrypted: boolean = true;

class myStack extends cdk.Stack {
    bucket: s3.Bucket;
    // ...
}

function makeEnv(account: string, region: string) : object {
    // ...
}
```

JavaScript

```
var encrypted = true;
```

```
class myStack extends cdk.Stack {
    bucket = undefined;
    // ...
}

function makeEnv(account, region) {
    // ...
}
```

では TypeScript、インターフェイスを使用して、必須プロパティとオプションプロパティのバンドル、およびそのタイプに名前を付けます。その後、インターフェイス名を型注釈として使用できます。例えば、関数の引数として使用するオブジェクトには、適切な型の必要なプロパティがある TypeScript ことを確認してください。

```
interface myFuncProps {
    code: lambda.Code,
    handler?: string
}
```

JavaScript にはインターフェイス機能がないため、タイプアノテーションを削除したら、インターフェイス宣言を完全に削除します。

関数またはメソッドが汎用型 (などobject) を返すが、その値をより汎用型のインターフェイスの一部ではないプロパティまたはメソッドにアクセスするために、より具体的な子型として扱う場合は、 を使用して値をキャスト TypeScript し、asその後にタイプまたはインターフェイス name. JavaScript doesn がサポート (または必要) するため、 asと次の識別子を削除します。あまりまれなキャスト構文では、タイプ名を角括弧 で囲む必要があります<LikeThis>。これらのキャストも削除する必要があります。

最後に、 はクラスのメンバーprivateのアクセス修飾子 public、protected、および TypeScript をサポートします。のすべてのクラスメンバー JavaScript はパブリックです。これらの修飾子は、表示される場所に関係なく削除するだけです。

これらの TypeScript 機能を特定して削除する方法を知ることは、 に短い TypeScriptスニペットを適応させる上で大きな道のりです JavaScript。ただし、他の TypeScript 機能を使用する可能性が高いため、より長い TypeScript 例をこの方法で変換することは実用的でない場合があります。このような場合は、 を Sucrase することをお勧めします。コードが未定義の変数を使用する場合など、Supcrase は苦情tscを発しません。構文的に有効な場合、いくつかの例外を除いて、Scrase は

それを に変換できます JavaScript。これにより、単独では実行できないスニペットを変換する場合に特に役立ちます。

への移行 TypeScript

多くの JavaScript デベロッパーは、プロジェクトが大きくなり、複雑になるにつれて に移行TypeScriptします。 TypeScript は JavaScriptのスーパーセットです。すべての JavaScript コードは有効な TypeScript コードであるため、コードに変更を加える必要はなく、サポートされている AWS CDK 言語でもあります。タイプ注釈やその他の TypeScript 機能はオプションであり、値を見つけると AWS CDK アプリに追加できます。 TypeScript また、 では、オプションの連鎖や nullish 結合などの JavaScript 新機能を確定前に早期にアクセスでき、Node.js をアップグレードする必要はありません。

TypeScriptの「シェープベース」インターフェイスは、 オブジェクト内の必須プロパティとオプションプロパティ (およびそのタイプ) のバンドルを定義し、コードの記述中に一般的なミスを検出し、IDE が堅牢なオートコンプリートやその他のリアルタイムのコーディングアドバイスを簡単に提供できるようにします。

でコーディング TypeScript するには、 TypeScript コンパイラ を使用してアプリケーションをコンパイルするという追加のステップが必要ですtsc。一般的な AWS CDK アプリケーションの場合、コンパイルには最大で数秒かかります。

既存の JavaScript AWS CDK アプリケーションを に移行する最も簡単な方法は TypeScript 、 を使用して新しい TypeScript プロジェクトを作成しcdk init app --language typescript、ソースファイル (および AWS Lambda 関数ソースコードなどのアセットなど、その他の必要なファイル) を新しいプロジェクトにコピーすることです。 JavaScript ファイルの名前を で終わるように変更.tsし、 で開発を開始します TypeScript。

Python AWS CDK での の操作

Python は、 で完全にサポートされているクライアント言語であり AWS Cloud Development Kit (AWS CDK) 、安定していると見なされます。Python AWS CDK で を操作するには、標準の Python 実装 (CPython)、 を使用した仮想環境virtualenv、Python パッケージインストーラ など、使い慣れたツールを使用しますpip。 AWS コンストラクトライブラリを構成するモジュールは、pypi.org 経由で配布されます。の Python バージョンでは AWS CDK 、Python スタイルの識別子 (snake_caseメソッド名など) を使用します。

任意のエディタまたは IDE を使用できます。多くの AWS CDK デベロッパーは、Visual Studio Code (またはオープンソースの同等の VSCodium) を使用します。これは、公式の拡張機能 を介して

Python を良好にサポートしています。開始するには、Python に含まれている IDLE 工ディタで十分です。の AWS CDK Python モジュールにはタイプヒントがあります。これは、型検証をサポートするリンティングツールまたは IDE に役立ちます。

トピック

- Python の開始方法
- 「プロジェクトの作成」
- AWS コンストラクトライブラリモジュールの管理
- での依存関係の管理 Python
- AWS CDK Python の idioms
- 合成とデプロイ

Python の開始方法

を使用するには AWS CDK、 AWS アカウントと認証情報が必要で、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

Python AWS CDK アプリケーションには Python 3.6 以降が必要です。まだインストールしていない場合は、python.org からオペレーティングシステムの互換性のあるバージョンをダウンロードします。Linux を実行する場合、システムに互換性のあるバージョンが付属しているか、ディストリビューションのパッケージマネージャー (yum、 aptなど) を使用してインストールできます。Mac ユーザーは、macOS 用の Linux スタイルのパッケージマネージャーである Homebrew に興味があるかもしれません。

> **Note**
>
> サードパーティー言語の非推奨: 言語バージョンは、ベンダーまたはコミュニティによって共有される EOL (サポート終了) までのみサポートされ、事前通知により変更される可能性があります。

Python パッケージインストーラー、pip、仮想環境マネージャー virtualenvも必要です。互換性のある Python バージョンの Windows インストールには、これらのツールが含まれています。Linux では、 pipと はパッケージマネージャーで個別のパッケージとして提供virtualenvされる場合があります。または、次のコマンドを使用してインストールすることもできます。

```
python -m ensurepip --upgrade
```

```
python -m pip install --upgrade pip
python -m pip install --upgrade virtualenv
```

アクセス許可エラーが発生した場合は、上記のコマンドを `--user`フラグで実行してモジュールがユーザーディレクトリにインストールされるようにするか、 `sudo`を使用してモジュールをシステム全体でインストールするアクセス許可を取得します。

> **Note**
>
> Linux ディストリビューションでは、Python 3`python3.x` の実行ファイル名を使用することが一般的であり、Python 2.x のインストール`python`を参照しています。一部の Distros には、Python 3 を参照するように`python`コマンドにインストールできるオプションのパッケージがあります。これを行わない場合は、プロジェクトのメインディレクトリ`cdk.json`で を編集することで、アプリケーションの実行に使用するコマンドを調整できます。

> **Note**
>
> Windows では、実行可能ファイル >Windows 用の Python ランチャーを使用して py Python (および pip) を呼び出すことができます。 https://docs.python.org/3/using/windows.html#launcher特に、ランチャーを使用すると、使用するインストール済みバージョンの Python を簡単に指定できます。
> コマンドライン`python`で と入力すると、Windows ストアから Python のインストールに関するメッセージが表示される場合、Python の Windows バージョンをインストールした後でも、Windows の「アプリ実行エイリアス管理設定パネル」を開き、Python の 2 つの App Installer エントリをオフにします。

「プロジェクトの作成」

空のディレクトリで を呼び出して`cdk init`、新しい AWS CDK プロジェクトを作成します。`--language` オプションを使用して を指定します`python`。

```
mkdir my-project
cd my-project
cdk init app --language python
```

`cdk init` は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換されます。ただし、名前は Python 識別子の形式に従う必要があります。例えば、数字で始まる、またはスペースを含むべきではありません。

新しいプロジェクトで作業するには、その仮想環境を有効にします。これにより、プロジェクトの依存関係をグローバルにインストールするのではなく、プロジェクトフォルダにローカルにインストールできます。

```
source .venv/bin/activate
```

> **Note**
>
> これは、仮想環境をアクティブにする Mac/Linux コマンドとして認識されるかもしれません。Python テンプレートには、同じコマンドを Windows で使用できるようにするバッチファイル、`source.bat` が含まれています。従来の Windows コマンドである も`.\venv\Scripts\activate`機能します。
> CDK Toolkit v1.70.0 以前を使用して AWS CDK プロジェクトを初期化した場合、仮想環境はではなく `.env` ディレクトリにあります`.venv`。

> **Important**
>
> 作業を開始するたびに、プロジェクトの仮想環境をアクティブ化します。そうしないと、そこにインストールされたモジュールにアクセスできなくなり、インストールしたモジュールは Python グローバルモジュールディレクトリに移動します (または、アクセス許可エラーが発生します) 。

仮想環境を初めてアクティブ化したら、アプリケーションの標準依存関係をインストールします。

```
python -m pip install -r requirements.txt
```

AWS コンストラクトライブラリモジュールの管理

Python パッケージインストーラー pipを使用して、アプリケーションで使用する AWS コンストラクトライブラリモジュール、および必要な他のパッケージをインストールおよび更新します。 pip は、

それらのモジュールの依存関係も自動的にインストールします。システムがスタンドアロンコマンドpipとして認識しない場合は、次のように Python モジュールpipとして を呼び出します。

```
python -m pip PIP-COMMAND
```

ほとんどの AWS CDK コンストラクトは にありますaws-cdk-lib。実験モジュールは、 のような別のモジュールにありますaws-cdk.*SERVICE-NAME*.alpha。サービス名には aws プレフィックスが含まれます。モジュールの名前がわからない場合は、PyPI で検索します。例えば、以下のコマンドは AWS CodeStar ライブラリをインストールします。

```
python -m pip install aws-cdk.aws-codestar-alpha
```

一部のサービスのコンストラクトは複数の名前空間にあります。例えば、 以外にもaws-cdk.aws-route53、、、 という名前の Amazon Route 53 名前空間が aws-route53-targets3 つ追加されていますaws-route53-patternsaws-route53resolver。

> **Note**
>
> CDK API リファレンスの Python エディションには、パッケージ名も表示されます。

Construct Library モジュールを AWS Python コードにインポートするために使用される名前は次のようになります。

```
import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lambda_
```

アプリケーションに AWS CDK クラスと AWS コンストラクトライブラリモジュールをインポートするときは、次のプラクティスをお勧めします。これらのガイドラインに従うことで、コードが他の AWS CDK アプリケーションと一貫性を持ち、理解しやすくなります。

- 通常、最上位の から個々のクラスをインポートしますaws_cdk。

  ```
  from aws_cdk import App, Construct
  ```

- から多数のクラスが必要な場合はaws_cdk、個々のクラスをインポートcdkする代わりに、 の名前空間エイリアスを使用できます。両方は避けてください。

  ```
  import aws_cdk as cdk
  ```

- 通常、短い名前空間エイリアスを使用して AWS ライブラリの作成 をインポートします。

```
import aws_cdk.aws_s3 as s3
```

モジュールをインストールしたら、プロジェクトの依存関係を一覧表示するプロジェクトの requirements.txt ファイルを更新します。これを行うのは、 を使用するのではなく、手動で行うのが最善ですpip freeze。 は、Python 仮想環境にインストールされているすべてのモジュールの最新バージョンをpip freezeキャプチャします。これは、プロジェクトをバンドルして他の場所で実行する場合に役立ちます。

ただし、通常、 requirements.txtは最上位の依存関係 (アプリケーションが直接依存するモジュール) のみをリストし、それらのライブラリの依存関係をリストしないでください。この戦略により、依存関係の更新が簡単になります。

を編集requirements.txtしてアップグレードを許可できます。==前述のバージョン番号を に置き換えて、互換性のある上位バージョンへのアップグレード~=を許可するか、バージョン要件を完全に削除して、利用可能な最新バージョンのモジュールを指定します。

アップグレードを許可するように適切にrequirements.txt編集された で、このコマンドを実行して、プロジェクトにインストールされているモジュールをいつでもアップグレードします。

```
pip install --upgrade -r requirements.txt
```

での依存関係の管理 Python

Python では、依存関係を指定するには、requirements.txtアプリケーションまたはコンストラクトライブラリsetup.pyに を入力します。その後、依存関係は PIP ツールで管理されます。PIP は、次のいずれかの方法で呼び出されます。

```
pip command options
python -m pip command options
```

python -m pip 呼び出しはほとんどのシステムで機能します。 pipでは、PIP の実行可能ファイルがシステムパス上にある必要があります。pip が機能しない場合は、 に置き換えてみてくださいpython -m pip。

cdk init --language python コマンドは、新しいプロジェクトの仮想環境を作成します。これにより、各プロジェクトに独自のバージョンの依存関係と基本的なrequirements.txtファイルを持たせる

ことができます。プロジェクトの使用を開始するsource .venv/bin/activateたびに を実行して、この仮想環境をアクティブ化する必要があります。Windows では、.\venv\Scripts\activate代わりに を実行します。

CDK アプリケーション

次は、requirements.txt ファイルの例です。PIP には依存関係ロック機能がないため、次に示すように、== 演算子を使用してすべての依存関係の正確なバージョンを指定することをお勧めします。

```
aws-cdk-lib==2.14.0
aws-cdk.aws-appsync-alpha==2.10.0a0
```

でモジュールをインストールpip installしても、自動的に に追加されませんrequirements.txt。自分で行う必要があります。依存関係の新しいバージョンにアップグレードする場合は、 でそのバージョン番号を編集しますrequirements.txt。

を作成または編集した後にプロジェクトの依存関係をインストールまたは更新するにはrequirements.txt、以下を実行します。

```
python -m pip install -r requirements.txt
```

> **Tip**
>
> pip freeze コマンドは、インストールされているすべての依存関係のバージョンを、テキストファイルに書き込むことができる形式で出力します。これは、 の要件ファイルとして使用できますpip install -r。このファイルは、すべての依存関係 (推移的な依存関係を含む) をテストした正確なバージョンに固定するのに便利です。後でパッケージをアップグレードする際の問題を回避するには、 freeze.txt (ではなく) など、別の ファイルを使用しますrequirements.txt。次に、プロジェクトの依存関係をアップグレードするときに再生成します。

サードパーティーのコンストラクトライブラリ

ライブラリでは、依存関係は で指定されるためsetup.py、パッケージがアプリケーションによって消費されると、推移的な依存関係が自動的にダウンロードされます。それ以外の場合、

パッケージを使用するすべてのアプリケーションは、依存関係を にコピーする必要がありますrequirements.txt。以下に例setup.pyを示します。

```
from setuptools import setup

setup(
  name='my-package',
  version='0.0.1',
  install_requires=[
    'aws-cdk-lib==2.14.0',
  ],
  ...
)
```

開発用にパッケージを操作するには、仮想環境を作成またはアクティブ化し、次のコマンドを実行します。

```
python -m pip install -e .
```

PIP は推移的な依存関係を自動的にインストールしますが、インストールできるパッケージのコピーは 1 つだけです。依存関係ツリーで最も高いバージョンが選択されます。アプリケーションには、インストールされるパッケージのバージョンが常に最後の単語になります。

AWS CDK Python の idioms

言語の競合

Python では、 lambdaは言語キーワードであるため、 AWS Lambda コンストラクトライブラリモジュールまたは Lambda 関数の名前として使用することはできません。このような競合の Python の規則では、lambda_変数名に のように末尾のアンダースコアを使用します。

慣例により、 AWS CDK コンストラクトの 2 番目の引数は という名前になりますid。独自のスタックとコンストラクトを記述するときは、Python 組み込み関数 をパラメータid「シャドウ」を呼び出しid()、オブジェクトの一意の識別子を返します。この関数はあまり使用されませんが、コンストラクトで必要になった場合は、 など、 引数の名前を変更しますconstruct_id。

引数とプロパティ

すべての AWS Construct Library クラスは、3 つの引数を使用してインスタンス化されます。つまり、コンストラクトが定義されているスコープ (コンストラクトツリー内の親) 、id 、props です。

これは、コンストラクトが作成するリソースの設定に使用するキーと値のペアのバンドルです。他のクラスやメソッドでは、引数に「属性のバンドル」パターンも使用します。

scope と id は、常にキーワード引数ではなく位置引数として渡す必要があります。これは、コンストラクトが scope または id という名前のプロパティを受け入れると名前が変わるためです。

Python では、props はキーワード引数として表現されます。引数にネストされたデータ構造が含まれている場合、これらはインスタンス化時に独自のキーワード引数を取るクラスを使用して表現されます。同じパターンが、構造化引数を取る他のメソッド呼び出しに適用されます。

例えば、Amazon S3 バケットの `add_lifecycle_rule`メソッドでは、 `transitions`プロパティは`Transition`インスタンスのリストです。

```
bucket.add_lifecycle_rule(
  transitions=[
    Transition(
      storage_class=StorageClass.GLACIER,
      transition_after=Duration.days(10)
    )
  ]
)
```

クラスを拡張したり、メソッドを上書きしたりする場合、親クラスでは理解できない独自の目的で追加の引数を受け入れることをお勧めします。この場合、**kwargsidiom の使用には関係ない引数を受け入れ、キーワードのみの引数を使用して関心のある引数を受け入れる必要があります。親のコンストラクタまたはオーバーライドされたメソッドを呼び出すときは、予期している引数のみを渡します (多くの場合、 のみ`**kwargs`)。親クラスまたはメソッドが想定しない引数を渡すと、エラーが発生します。

```
class MyConstruct(Construct):
    def __init__(self, id, *, MyProperty=42, **kwargs):
        super().__init__(self, id, **kwargs)
        # ...
```

の将来のリリースでは、独自のプロパティに使用した名前の新しいプロパティが同時に追加 AWS CDK される可能性があります。これにより、コンストラクトまたはメソッドのユーザーに技術的な問題が発生することはありません (プロパティが「チェーンを上回らないため、親クラスまたはオーバーライドされたメソッドは単にデフォルト値を使用するため)。ただし、混乱を引き起こす可能性があります。この潜在的な問題は、プロパティに名前を付けることで回避できます。これにより、プ

ロパティはコンストラクトに明確に属します。新しいプロパティが多数ある場合は、適切に名前が付けられたクラスにバンドルし、単一のキーワード引数として渡します。

欠落した値

AWS CDK は Noneを使用して、欠落している値または未定義の値を表します。を使用するときは**kwargs、 プロパティが指定されていない場合、ディクショナリの get()メソッドを使用してデフォルト値を指定します。を使用するとkwargs[...]、欠損値が増えるためKeyError、 を使用しないでください。

```
encrypted = kwargs.get("encrypted")         # None if no property "encrypted" exists
encrypted = kwargs.get("encrypted", False)  # specify default of False if property is
 missing
```

一部の AWS CDK メソッド (ランタイムコンテキスト値tryGetContext()の取得など) は を返す場合がありNone、明示的に確認する必要があります。

インターフェイスの使用

Python には、他の言語と同様にインターフェイス機能はありませんが、同様の抽象ベースクラスがあります。(インターフェイスに慣れていない場合、Wikipedia には という優れた概要があります)。これは TypeScript、 が実装されている言語 AWS CDK であり、インターフェイスを提供し、コンストラクトやその他の AWS CDK オブジェクトには、特定のクラスから継承するのではなく、特定のインターフェイスに準拠するオブジェクトが必要になることがよくあります。そのため、 は JSII レイヤーの一部として独自のインターフェイス機能 AWS CDK を提供します。

クラスが特定のインターフェイスを実装していることを示すには、 デ@jsii.implementsコレータを使用できます。

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
    def visit(self, node: IConstruct) -> None:
        print("Visited", node.node.path)
```

タイプミス

Python は動的型付けを使用します。すべての変数は任意の型の値を参照できます。パラメータと戻り値は 型で注釈を付けることができますが、これらは「ヒント」であり、強制されません。つま

り、Python では、誤ったタイプの値を AWS CDK コンストラクトに渡すのは簡単です。静的に型指定された言語の場合と同様に、ビルド中に型エラーが発生する代わりに、JSII レイヤー (Python と AWS CDKの TypeScript コアの間で変換されます) が予期しない型に対応できない場合にランタイムエラーが発生することがあります。

経験上、Python プログラマーが行うタイプエラーは、これらのカテゴリに分類される傾向があります。

- コンストラクトがコンテナ (Python リストまたはディクショナリ) を期待する単一の値を渡す、またはその逆。
- レイヤー 1 (`CfnXxxxxx`) コンストラクトに関連付けられたタイプの値を L2 または L3 コンストラクトに、またはその逆に渡す。

AWS CDK Python モジュールには型注釈が含まれているため、それらをサポートするツールを使用して型をサポートできます。など、これらをサポートする IDE を使用していない場合はPyCharm、MyPyタイプ検証をビルドプロセスのステップとして呼び出すことができます。タイプ関連のエラーのエラーメッセージを改善できるランタイムタイプチェッカーもあります。

合成とデプロイ

AWS CDK アプリケーションで定義されたスタックは、以下のコマンドを使用して個別に、または一緒に合成およびデプロイできます。通常、発行するときは、プロジェクトのメインディレクトリにある必要があります。

- `cdk synth`: AWS CDK アプリケーション内の 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。
- `cdk deploy`: AWS CDK アプリケーション内の 1 つ以上のスタックで定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。それ以外の場

合、シェルは AWS CDK Toolkit に渡される前に、現在のディレクトリ内のファイルの名前に拡張しようとすることがあります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> スタックをデプロイする前に明示的に合成する必要はありません。 はこのステップcdk deployを実行して、最新のコードがデプロイされるようにします。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

Java AWS CDK での の使用

Java は、 で完全にサポートされているクライアント言語 AWS CDK であり、安定していると見なされます。JDK (Oracle の 、または Amazon Corretto などの OpenJDK ディストリビューション) や Apache Maven など、使い慣れたツールを使用して Java で AWS CDK アプリケーションを開発できます。

は Java 8 以降 AWS CDK をサポートしています。ただし、可能な最新バージョンを使用することをお勧めします。これは、言語のそれ以降のバージョンには、 AWS CDK アプリケーションの開発に特に便利な改善が含まれているためです。例えば、Java 9 では `Map.of()`メソッドが導入されています (でオブジェクトリテラルとして書き込まれるハッシュマップを宣言する便利な方法 TypeScript) 。Java 10 では、 `var`キーワードを使用したローカル型推論が導入されています。

> **Note**
>
> このデベロッパーガイドのほとんどのコード例は、Java 8 で動作します。いくつかの例ではを使用していますMap.of()。これらの例には、Java 9 が必要であることを示すコメントが含まれています。

任意のテキストエディタ、または Maven プロジェクトを読み取ることができる Java IDE を使用して、 AWS CDK アプリケーションを操作できます。このガイドでは Eclipse のヒントを提供していま

すが、IntelliJ IDEA やその他の IDEs は Maven プロジェクトをインポートでき NetBeans、Java での AWS CDK アプリケーション開発に使用できます。

Java 以外の JVM ホスト言語 (Kotlin、Groovy、Clojure、Scala など) で AWS CDK アプリケーションを記述することは可能ですが、エクスペリエンスは特にイオマティックではない場合があり、これらの言語をサポートすることはできません。

トピック

Java の開始方法

を使用するには AWS CDK、 AWS アカウントと認証情報が必要であり、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

Java AWS CDK アプリケーションには Java 8 (v1.8) 以降が必要です。Amazon Corretto をお勧めしますが、任意の OpenJDK ディストリビューションまたは Oracle の JDK を使用できます。Apache Maven 3.5 以降も必要です。Gradle などのツールを使用することもできますが、 AWS CDK Toolkit によって生成されるアプリケーションスケルトンは Maven プロジェクトです。

> **Note**
>
> サードパーティー言語の廃止: 言語バージョンは、ベンダーまたはコミュニティによって共有される EOL (サポート終了) までのみサポートされ、事前通知により変更される可能性があります。

「プロジェクトの作成」

空のディレクトリ`cdk init`で を呼び出して、新しい AWS CDK プロジェクトを作成します。`--language` オプションを使用して、 を指定します`java`。

```
mkdir my-project
cd my-project
cdk init app --language java
```

cdk init は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換されます。ただし、名前は Java 識別子の形式に従う必要があります。例えば、数字で開始したり、スペースを含めたりしないでください。

結果として得られるプロジェクトには、software.amazon.awscdkMaven パッケージへの参照が含まれています。この関数とその依存関係は Maven によって自動的にインストールされます。

IDE を使用している場合は、プロジェクトを開くかインポートできます。例えば、Eclipse では、ファイル > インポート > Maven > 既存の Maven プロジェクト を選択します。プロジェクト設定が Java 8 (1.8) を使用するように設定されていることを確認します。

AWS コンストラクトライブラリモジュールの管理

Maven を使用して、グループ にある AWS Construct Library パッケージをインストールしますsoftware.amazon.awscdk。ほとんどのコンストラクトはアーティファクト にありaws-cdk-lib、デフォルトで新しい Java プロジェクトに追加されます。上位レベルの CDK サポートがまだ開発中のサービスのモジュールは、サービス名の短縮バージョン (なし AWS または Amazon プレフィックス) で名前が付けられた個別の「実験的な」パッケージにあります。Maven Central リポジトリを検索して、すべての AWS CDK および AWS コンストラクトモジュールライブラリの名前を検索します。

> **Note**
>
> CDK API リファレンスの Java エディションには、パッケージ名も表示されます。

一部のサービスの AWS 「コンストラクトライブラリ」のサポートは、複数の名前空間にあります。例えば、Amazon Route 53 には、、software.amazon.awscdk.route53、route53-patterns、 route53resolverに分割された機能がありますroute53-targets。

メイン AWS CDK パッケージは Java コードで としてインポートされますsoftware.amazon.awscdk。Construct Library AWS のさまざまなサービスのモジュールは、 の下にライブsoftware.amazon.awscdk.servicesし、Maven パッ

ケージ名と同様に名前が付けられます。例えば、Amazon S3 モジュールの名前空間は `software.amazon.awscdk.services.s3` です。

Java ソースファイルごとに使用する AWS Construct Library クラスごとに個別の Java `import` ステートメントを作成し、ワイルドカードインポートを避けることをお勧めします。`import` ステートメントを使用せずに、タイプの完全修飾名 (名前空間を含む) をいつでも使用できます。

アプリケーションが実験的なパッケージに依存している場合は、プロジェクトの `pom.xml` を編集し、`<dependencies>` コンテナに新しい `<dependency>` 要素を追加します。例えば、次の `<dependency>` 要素は CodeStar 実験コンストラクトライブラリモジュールを指定します。

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>codestar-alpha</artifactId>
    <version>2.0.0-alpha.10</version>
</dependency>
```

> **Tip**
>
> Java IDE を使用する場合、Maven の依存関係を管理する機能がある可能性があります。ただし、IDE の機能が手動で行う操作と一致していることを確認する場合を除き、`pom.xml` 直接編集することをお勧めします。

での依存関係の管理 Java

Java では、依存関係は `pom.xml` で指定され、Maven を使用してインストールされます。`<dependencies>` コンテナには、各パッケージの `<dependency>` 要素が含まれています。以下に、一般的な CDK Java アプリの `pom.xml` のセクションを示します。

```
<dependencies>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>aws-cdk-lib</artifactId>
        <version>2.14.0</version>
    </dependency>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>appsync-alpha</artifactId>
```

```
        <version>2.10.0-alpha.0</version>
    </dependency>
</dependencies>
```

> **Tip**
>
> 多くの Java IDEs には Maven サポートとビジュアルpom.xml工ディタが統合されており、依存関係の管理に便利な場合があります。

Maven は依存関係ロックをサポートしていません。でバージョン範囲を指定することは可能ですがpom.xml、ビルドを繰り返し可能に保つには、常に正確なバージョンを使用することをお勧めします。

Maven は推移的な依存関係を自動的にインストールしますが、インストールできる各パッケージのコピーは 1 つだけです。POM ツリーで最も高いバージョンが選択されます。アプリケーションには、インストールされるパッケージのバージョンで常に最後の単語が含まれます。

Maven は、プロジェクトを構築 (mvn compile) またはパッケージ化する (mvn package) たびに、依存関係を自動的にインストールまたは更新します。CDK Toolkit は実行するたびにこれを自動的に行うため、一般的に Maven を手動で呼び出す必要はありません。

AWS CDK Java での idioms

プロンプト

すべての AWS Construct Library クラスは、3 つの引数を使用してインスタンス化されます。コンストラクトが定義されているスコープ (コンストラクトツリーの親) 、ID 、props は、コンストラクトが作成するリソースの設定に使用するキーと値のペアのバンドルです。他のクラスやメソッドも引数に「属性のバンドル」パターンを使用します。

Java では、props は Builder パターン を使用して表現されます。各コンストラクトタイプには、対応するプロパティタイプがあります。例えば、Bucketコンストラクト (Amazon S3 バケットを表す) は、 のインスタンスをプロパッシュするために を受け取りますBucketProps。

BucketProps クラス (すべての AWS Construct Library props クラスなど) には、 という内部クラスがありますBuilder。BucketProps.Builder タイプは、BucketPropsインスタンスのさまざまなプロパティを設定するメソッドを提供します。各メソッドはBuilderインスタンスを返すた

め、メソッド呼び出しを連鎖して複数のプロパティを設定できます。チェーンの最後に、 を呼び出しbuild()て実際にBucketPropsオブジェクトを生成します。

```
Bucket bucket = new Bucket(this, "MyBucket", new BucketProps.Builder()
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build());
```

コンストラクトと、props のようなオブジェクトを最終的な引数として受け取る他のクラスは、ショートカットを提供します。クラスにはBuilder、1 つのステップでインスタンス化し、その props オブジェクトをインスタンス化する独自の があります。これにより、 BucketPropsと の両方を明示的にインスタンス化 (例えば) Bucketする必要はなく、props タイプにインポートする必要もありません。

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build();
```

既存のコンストラクトから独自のコンストラクトを取得する場合、追加のプロパティを受け入れたい場合があります。これらのビルダーパターンに従うことをお勧めします。ただし、これはコンストラクトクラスをサブクラスするほどシンプルではありません。2 つの新しいBuilderクラスの移動部分を自分で指定する必要があります。コンストラクトに 1 つ以上の追加の引数を受け入れるだけで済みます。引数がオプションの場合は、追加のコンストラクタを指定する必要があります。

一般的な構造

一部の APIs、 AWS CDK は JavaScript 配列または型指定されていないオブジェクトをメソッドへの入力として使用します。(例えば、 AWS CodeBuildの BuildSpec.fromObject()メソッドを参照してください。) Java では、これらのオブジェクトは として表されますjava.util.Map<String, Object>。値がすべて文字列である場合は、 を使用できますMap<String, String>。

Java では、他の言語のようにコンテナのリテラルを記述する方法はありません。Java 9 以降では、java.util.Map.of()を使用して、これらの呼び出しの 1 つとインラインで最大 10 個のエントリのマップを簡単に定義できます。

```
java.util.Map.of(
    "base-directory", "dist",
```

```
    "files", "LambdaStack.template.json"
)
```

10 個を超えるエントリを持つマップを作成するには、 を使用しますjava.util.Map.ofEntries()。

Java 8 を使用している場合は、次のような独自のメソッドを指定できます。

JavaScript 配列は Java List<String>では List<Object>または として表されます。メソッドjava.util.Arrays.asListは、短い を定義するのに便利ですList。

```
List<String> cmds = Arrays.asList("cd lambda", "npm install", "npm install typescript")
```

欠落した値

Java では、props などの AWS CDK オブジェクトの欠損値は で表されますnull。値を使用する前にnull、 値に 値が含まれていることを明示的にテストする必要があります。Java には、他の言語と同様に NULL 値を処理するのに役立つ「構文結合」はありません。Apache ObjectUtilのdefaultIfNullと は、状況によってはfirstNonNull便利です。または、独自の静的ヘルパーメソッドを記述して、潜在的に NULL 値の処理とコードの読み取りを容易にします。

構築、合成、デプロイ

は、アプリケーションを実行する前に AWS CDK 自動的にコンパイルします。ただし、エラーをチェックしてテストを実行するために、アプリケーションを手動で構築すると便利です。これは、IDE (例えば、Eclipse で Control-B キーを押します) で実行するか、プロジェクトのルートディレクトリにあるコマンドプロンプトmvn compileで発行します。

コマンドプロンプトmvn testで を実行して、作成したテストを実行します。

AWS CDK アプリケーションで定義されたスタックは、合成して個別にデプロイすることも、以下のコマンドを使用してまとめてデプロイすることもできます。通常、プロジェクトを発行するときは、プロジェクトのメインディレクトリにいる必要があります。

- cdk synth: AWS CDK アプリケーションの 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。
- cdk deploy: AWS CDK アプリケーション内の 1 つ以上のスタックによって定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

また、ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。そうしないと、シェルは AWS CDK Toolkit に渡す前に、現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> デプロイする前にスタックを明示的に合成する必要はありません。 はこのステップcdk deployを実行して、最新のコードがデプロイされることを確認します。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

C# AWS CDK での の操作

.NET は、 用に完全にサポートされているクライアント言語であり AWS CDK 、安定していると見なされます。C# は、例とサポートを提供する主要な .NET 言語です。Visual Basic や F# など、他の .NET 言語の AWS CDK アプリケーションを作成することを選択できますが、これらの言語を CDK で使用するためのサポート AWS は限られています。

Visual Studio、Visual Studio Code、 `dotnet` コマンド、 NuGet パッケージマネージャーなどの使い慣れたツールを使用して、C# で AWS CDK アプリケーションを開発できます。 AWS コンストラクトライブラリで構成されるモジュールは、nuget.org 経由で配布されます。

Windows で Visual Studio 2019 (任意のエディション) を使用して、C# で AWS CDK アプリケーションを開発することをお勧めします。

トピック

C# の開始方法

を使用するには AWS CDK、 AWS アカウントと認証情報が必要です。また、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

C# AWS CDK アプリケーションには .NET Core v3.1 以降が必要です。こちらは から入手できます。

.NET ツールチェーンにはdotnet、.NET アプリケーションを構築および実行し、 NuGet パッケージを管理するためのコマンドラインツールである が含まれています。Visual Studio で主に作業する場合でも、このコマンドはバッチ操作や AWS 構成ライブラリパッケージのインストールに役立ちます。

「プロジェクトの作成」

空のディレクトリcdk initで を呼び出して、新しい AWS CDK プロジェクトを作成します。--language オプションを使用して、 を指定しますcsharp。

```
mkdir my-project
cd my-project
cdk init app --language csharp
```

cdk init は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換されます。ただし、名前は C# 識別子の形式に従う必要があります。例えば、数字で開始したり、スペースを含めたりしないでください。

結果のプロジェクトには、 Amazon.CDK.Lib NuGet パッケージへの参照が含まれます。とその依存関係は によって自動的にインストールされます NuGet。

AWS コンストラクトライブラリモジュールの管理

.NET エコシステムは NuGet パッケージマネージャーを使用します。コアクラスとすべての安定したサービスコンストラクトを含むメイン CDK パッケージは、 ですAmazon.CDK.Lib。新しい機能が開発中である実験モジュールは、 のように名前が付けられAmazon.CDK.AWS.*SERVICE-NAME*.Alpha、サービス名は AWS または Amazon プレフィックスのない短縮名です。例えば、AWS IoT モジュールの NuGet パッケージ名は ですAmazon.CDK.AWS.IoT.Alpha。必要なパッケージが見つからない場合は、Nuget.org を検索します。

> **Note**
>
> CDK API リファレンスの .NET エディションには、パッケージ名も表示されます。

一部のサービスの AWS 「コンストラクトライブラリ」のサポートは、複数のモジュールにあります。例えば、 には という名前の 2 番目のモジュール AWS IoT がありますAmazon.CDK.AWS.IoT.Actions.Alpha。

ほとんどの AWS CDK アプリで必要となる AWS CDKのメインモジュールは、C# コードで としてインポートされますAmazon.CDK。 AWS コンストラクトライブラリのさまざまなサービスのモジュールは、 にありますAmazon.CDK.AWS。例えば、Amazon S3 モジュールの名前空間は ですAmazon.CDK.AWS.S3。

CDK コアコンストラクトと、各 C# ソースファイルで使用する AWS サービスごとに C# usingディレクティブを記述することをお勧めします。名前の競合を解決するには、名前空間またはタイプにエイリアスを使用する方が便利な場合があります。using ステートメントを使用せずに、タイプの名前 (名前空間を含む) をいつでも使用できます。

での依存関係の管理 C#

C# AWS CDK apps では、依存関係の管理に を使用します NuGet。 NuGet には、主に同等のインターフェイスが 4 つあります。ニーズと作業スタイルに合ったものを使用してください。Paket MyGetや などの互換性のあるツールを使用したり、.csprojファイルを直接編集したりすることもできます。

NuGet では、依存関係のバージョン範囲を指定することはできません。すべての依存関係は特定のバージョンに固定されます。

依存関係を更新すると、Visual Studio は NuGet を使用して、次回ビルド時に各パッケージの指定されたバージョンを取得します。Visual Studio を使用していない場合は、 dotnet restore コマンドを使用して依存関係を更新します。

プロジェクトファイルの直接編集

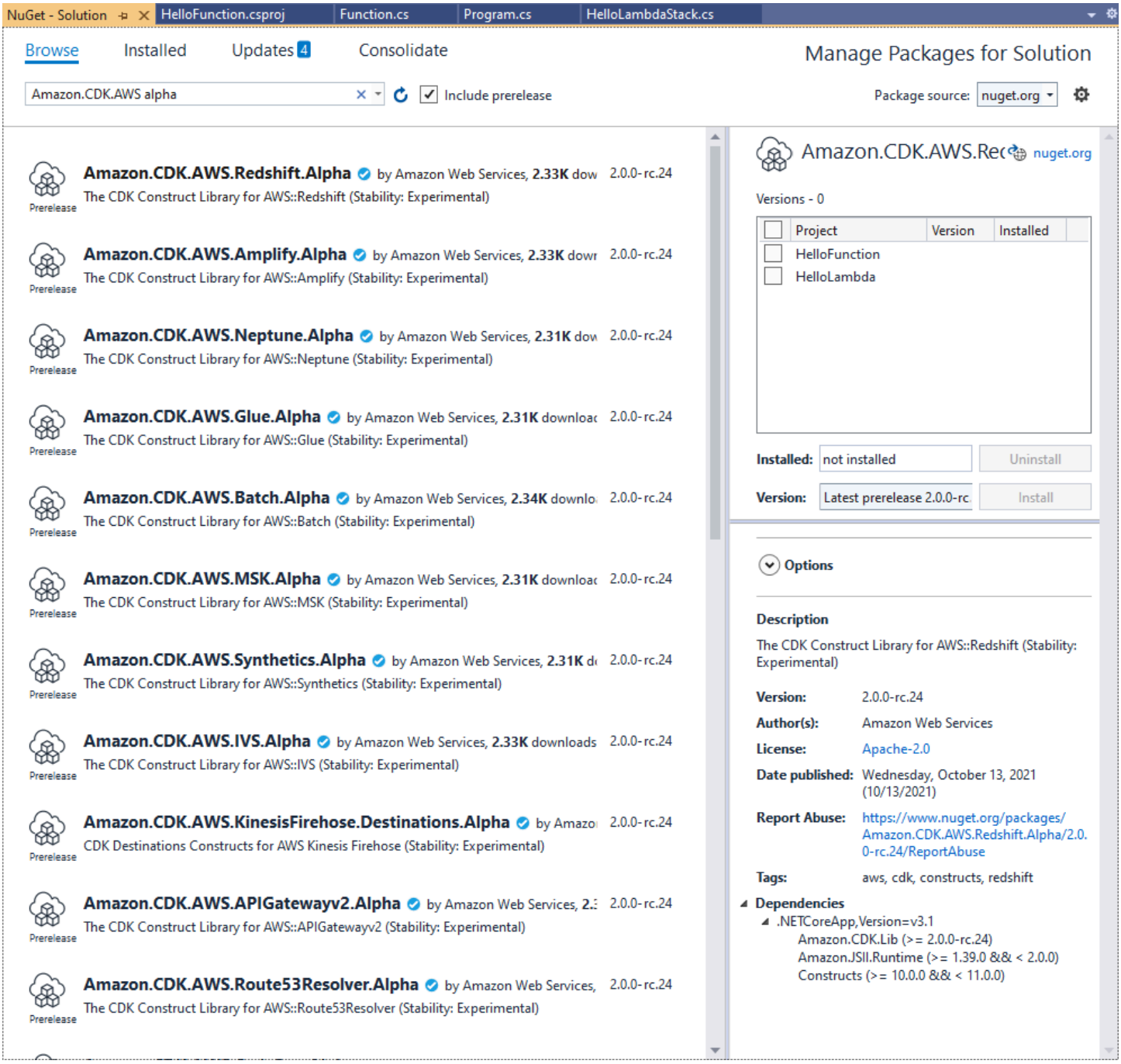
プロジェクトの `.csproj` ファイルには、依存関係を`<PackageReference`要素としてリストする`<ItemGroup>`コンテナが含まれています。

```
<ItemGroup>
    <PackageReference Include="Amazon.CDK.Lib" Version="2.14.0" />
    <PackageReference Include="Constructs" Version="%constructs-version%" />
</ItemGroup>
```

Visual Studio NuGet GUI

Visual Studio の NuGet ツールには、ツール > NuGet パッケージマネージャー > ソリューションの NuGet パッケージの管理からアクセスできます。参照タブを使用して、インストールする AWS 構成ライブラリパッケージを検索します。モジュールのプレリリースバージョンを含む目的のバージョンを選択し、開いているプロジェクトに追加できます。

> **Note**
>
> すべての AWS Construct Library モジュールは「実験的」と見なされ (「」を参照the section called “バージョニング”)、 でプレリリースとしてフラグが付けられ NuGet 、`alpha`名前のサフィックスが付けられます。



更新ページを見て、パッケージの新しいバージョンをインストールします。

NuGet コンソール

NuGet コンソールは、Visual Studio プロジェクトのコンテキスト NuGet で動作する PowerShell ベースの インターフェイスです。Visual Studio で開くには、ツール > NuGet パッケージマネージャー > パッケージマネージャーコンソール を選択します。このツールの使用の詳細について

は、Visual Studio の「パッケージマネージャーコンソールを使用したパッケージのインストールと管理」を参照してください。

dotnet コマンド

dotnet コマンドは、Visual Studio C# プロジェクトを操作するための主要なコマンドラインツールです。Windows コマンドプロンプトから呼び出すことができます。多くの機能の中で、 dotnetは Visual Studio プロジェクトに依存関係を追加できます NuGet。

Visual Studio プロジェクト (.csproj) ファイルと同じディレクトリにいると仮定して、次のようなコマンドを実行してパッケージをインストールします。メイン CDK ライブラリはプロジェクトの作成時に含まれているため、実験的なモジュールを明示的にインストールするだけで済みます。実験モジュールでは、明示的なバージョン番号を指定する必要があります。

```
dotnet add package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

別のディレクトリからコマンドを発行できます。これを行うには、 addキーワードの後に、プロジェクトファイル、またはそれを含むディレクトリへのパスを含めます。次の例では、プロジェクトのメインディレクトリにいることを前提 AWS CDK としています。

```
dotnet add src/PROJECT-DIR package Amazon.CDK.AWS.IoT.Alpha -v VERSION-NUMBER
```

パッケージの特定のバージョンをインストールするには、 -vフラグと目的のバージョンを含めます。

パッケージを更新するには、インストールに使用したのと同じdotnet addコマンドを発行します。実験的なモジュールの場合は、ここでも明示的なバージョン番号を指定する必要があります。

dotnet コマンドを使用したパッケージの管理の詳細については、「dotnet CLI を使用したパッケージのインストールと管理」を参照してください。

nuget コマンド

nuget コマンドラインツールは NuGet パッケージをインストールおよび更新できます。ただし、Visual Studio プロジェクトの設定方法は、プロジェクトcdk initの設定方法とは異なります。(技術的な詳細: はPackages.configプロジェクトとnuget連携し、 はより新しいスタイルのPackageReferenceプロジェクトcdk initを作成します。)

によって作成された AWS CDK プロジェクトで nuget ツールを使用することはお勧めしませんcdk init。別のタイプのプロジェクトを使用していて、 を使用する場合はnuget、 NuGet CLI リファレンスを参照してください。

AWS CDK C# のイディオム

プロンプト

すべての AWS Construct Library クラスは、3 つの引数を使用してインスタンス化されます。コンストラクトが定義されているスコープ (コンストラクトツリー内の親) 、ID 、props は、コンストラクトが作成するリソースの設定に使用するキーと値のペアのバンドルです。他のクラスやメソッドも引数に「属性のバンドル」パターンを使用します。

C# では、props は props 型を使用して表現されます。イディオマティック C# 方式では、オブジェクトイニシャライザを使用してさまざまなプロパティを設定できます。ここでは、 Bucketコンストラクトを使用して Amazon S3 バケットを作成します。対応するプロパティタイプは ですBucketProps。

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
    Versioned = true
});
```

> **Tip**
>
> パッケージAmazon.JSII.Analyzersをプロジェクトに追加して、Visual Studio 内のプロパティ定義で必要な値を確認します。

クラスを拡張したり、メソッドを上書きしたりする場合、親クラスでは理解できない独自の目的で追加のプロップスを受け入れることをお勧めします。これを行うには、適切なプロパティタイプをサブクラス化し、新しい属性を追加します。

```
// extend BucketProps for use with MimeBucket
class MimeBucketProps : BucketProps {
    public string MimeType { get; set; }
}

// hypothetical bucket that enforces MIME type of objects inside it
class MimeBucket : Bucket {
```

```
     public MimeBucket( readonly Construct scope, readonly string id, readonly
 MimeBucketProps props=null) : base(scope, id, props) {
         // ...
     }
}

// instantiate our MimeBucket class
var bucket = new MimeBucket(this, "MyBucket", new MimeBucketProps {
    Versioned = true,
    MimeType = "image/jpeg"
});
```

親クラスのイニシャライザまたはオーバーライドされたメソッドを呼び出す場合、通常、受信したプロパを渡すことができます。新しいタイプは親と互換性があり、追加した追加のプロップは無視されます。

の将来のリリースでは、独自のプロパティに使用した名前で新しいプロパティが同時に追加 AWS CDK される可能性があります。これによりコンストラクトやメソッドを使用する技術的な問題が発生することはありません (プロパティが「チェーン上」に渡されないため、親クラスや上書きされたメソッドは単にデフォルト値を使用するため)。ただし、コンストラクトのユーザーに混乱が生じる可能性があります。プロパティに名前を付けることで、この潜在的な問題を回避し、コンストラクトに明確に属させることができます。新しいプロパティが多数ある場合は、適切に名前が付けられたクラスにバンドルし、単一のプロパティとして渡します。

一般的な構造

一部の APIs、 AWS CDK は JavaScript 配列または型指定されていないオブジェクトをメソッドへの入力として使用します。(例えば、 AWS CodeBuildの [BuildSpec.fromObject()](#)メソッドを参照してください。) C# では、これらのオブジェクトは として表されますSystem.Collections.Generic.Dictionary<String, Object>。値がすべて文字列の場合、Dictionary<String, String>. JavaScript arrays は C# の object[]または string[]配列タイプとして表現できます。

> **Tip**
>
> これらの特定のディクショナリタイプを簡単に操作できるように、短い エイリアスを定義できます。
>
> ```
> using StringDict = System.Collections.Generic.Dictionary<string, string>;
> ```

```
using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
```

欠落した値

C# では、props などの AWS CDK オブジェクトの欠損値は で表されますnull。null 条件付きメンバーアクセス演算子?.と null 結合演算子??は、これらの値を操作するのに役立ちます。

```
// mimeType is null if props is null or if props.MimeType is null
string mimeType = props?.MimeType;

// mimeType defaults to text/plain. either props or props.MimeType can be null
string MimeType = props?.MimeType ?? "text/plain";
```

構築、合成、デプロイ

は、アプリケーションを実行する前に AWS CDK 自動的にコンパイルします。ただし、エラーを確認してテストを実行するために、アプリケーションを手動で構築すると便利です。これを行うには、Visual Studio で F6 を押すか、コマンドラインdotnet build srcから を発行します。ここで、 srcは Visual Studio Solution (.sln) ファイルを含むプロジェクトディレクトリ内のディレクトリです。

AWS CDK アプリケーションで定義されたスタックは、合成して個別にデプロイすることも、以下のコマンドを使用してまとめてデプロイすることもできます。通常、プロジェクトを発行するときは、プロジェクトのメインディレクトリにいる必要があります。

- cdk synth: AWS CDK アプリケーションの 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。
- cdk deploy: AWS CDK アプリケーション内の 1 つ以上のスタックによって定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                # app defines single stack
cdk deploy Happy Grumpy  # app defines two or more stacks; two are deployed
```

また、ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。そうしないと、シェルは AWS CDK Toolkit に渡す前に、現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> デプロイする前にスタックを明示的に合成する必要はありません。 はこのステップcdk deployを実行して、最新のコードがデプロイされることを確認します。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

Go AWS CDK での の使用

Go は、 で完全にサポートされているクライアント言語であり AWS Cloud Development Kit (AWS CDK) 、安定していると見なされます。Go AWS CDK で を操作するには、使い慣れたツールを使用します。Go バージョンの AWS CDK でも、Go スタイルの識別子が使用されます。

CDK がサポートする他の言語とは異なり、Go は従来のオブジェクト指向プログラミング言語ではありません。Go は、他の言語が継承を利用することが多いコンポジションを使用します。可能な限りイディオマティックな Go アプローチを採用しようとしましたが、CDK が異なる場所があります。

このトピックでは、Go AWS CDK で を使用する際のガイダンスを提供します。のシンプルな Go プロジェクトのチュートリアルについては、お知らせブログ記事を参照してください AWS CDK。

トピック

- Go の開始方法
- 「プロジェクトの作成」
- AWS コンストラクトライブラリモジュールの管理
- での依存関係の管理 Go
- AWS CDK Go のイディオム
- 構築、合成、デプロイ

Go の開始方法

を使用するには AWS CDK、 AWS アカウントと認証情報が必要であり、Node.js と AWS CDK Toolkit がインストールされている必要があります。の開始方法 AWS CDK を参照してください。

の Go バインディングは、標準の Go ツールチェーン v1.18 以降 AWS CDK を使用します。選択したエディタを使用できます。

> **Note**
>
> サードパーティー言語の廃止: 言語バージョンは、ベンダーまたはコミュニティによって共有される EOL (サポート終了) までのみサポートされ、事前通知により変更される可能性があります。

「プロジェクトの作成」

空のディレクトリcdk initで を呼び出して、新しい AWS CDK プロジェクトを作成します。--language オプションを使用して、 を指定しますgo。

```
mkdir my-project
cd my-project
cdk init app --language go
```

cdk init は、プロジェクトフォルダの名前を使用して、クラス、サブフォルダ、ファイルなど、プロジェクトのさまざまな要素に名前を付けます。フォルダ名のハイフンはアンダースコアに変換されます。ただし、名前は Go 識別子の形式に従う必要があります。例えば、数字で開始したり、スペースを含めたりしないでください。

結果として得られるプロジェクトには、 のコア AWS CDK Go モジュール への参照が含まれていますgithub.com/aws/aws-cdk-go/awscdk/v2go.mod。このモジュールやその他の必要なモジュールをインストールgo getする際の問題。

AWS コンストラクトライブラリモジュールの管理

ほとんどの AWS CDK ドキュメントや例では、「モジュール」という単語は、 AWS サービスごとに 1 AWS つ以上の Construct Library モジュールを指すためによく使用されます。これは、用語のイディオマティック Go の使用とは異なります。CDK コンストラクトライブラリは、個々のコンスト

ラクトライブラリモジュールを持つ 1 つの Go モジュールで提供され、そのモジュール内の Go パッケージとして提供されるさまざまな AWS サービスをサポートします。

一部のサービスの AWS 「コンストラクトライブラリのサポートは、複数のコンストラクトライブラリモジュール (Go パッケージ) にあります。例えば、Amazon Route 53 には、、、awsroute53patterns、 という名前のメインawsroute53パッケージに加えてawsroute53resolver、3 つの Construct Library モジュールがありますawsroute53targets。

ほとんどの AWS CDK アプリで必要となる AWS CDKのコアパッケージは、Go コードに としてインポートされますgithub.com/aws/aws-cdk-go/awscdk/v2。 AWS コンストラクトライブラリ内のさまざまなサービスのパッケージは、 にありますgithub.com/aws/aws-cdk-go/awscdk/v2。例えば、Amazon S3 モジュールの名前空間は ですgithub.com/aws/aws-cdk-go/awscdk/v2/awss3。

```
import (
        "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
        // ...
)
```

アプリで使用するサービスの 構成ライブラリモジュール (Go パッケージ) をインポートしたら、 などを使用してそのモジュールの構成にアクセスしますawss3.Bucket。

での依存関係の管理 Go

Go では、依存関係のバージョンは で定義されますgo.mod。デフォルトgo.modは、ここに示すようなものです。

```
module my-package

go 1.16

require (
  github.com/aws/aws-cdk-go/awscdk/v2 v2.16.0
  github.com/aws/constructs-go/constructs/v10 v10.0.5
  github.com/aws/jsii-runtime-go v1.29.0
)
```

パッケージ名 (Go 並列のモジュール) は URL によって指定され、必要なバージョン番号が追加されます。Go のモジュールシステムはバージョン範囲をサポートしていません。

必要なモジュールをすべてインストールして を更新するには、 go get コマンドを発行しますgo.mod。依存関係で利用可能な更新のリストを表示するには、 を発行しますgo list -m -u all。

AWS CDK Go のイディオム

フィールド名とメソッド名

フィールド名とメソッド名は TypeScript、CDK のオリジン言語である でキャメルケース (likeThis) を使用します。Go では、これらは Go の規則に従います。また、パスカル文字 () も使用しますLikeThis。

クリーンアップ

main メソッドで、 を使用して CDK アプリがそれ自体の後にクリーンアップするdefer jsii.Close()ことを確認します。

欠損値とポインタ変換

Go では、プロパティバンドルなどの AWS CDK オブジェクトの欠損値は で表されますnil。Go には NULL 可能な型がありません。含めることができる型nilはポインタのみです。値をオプションにするには、プリミティブ型であっても、すべての CDK プロパティ、引数、戻り値はポインターです。これは必須の値にもオプションの値にも適用されるため、後で必須値がオプションになった場合、タイプに重大な変更を加える必要はありません。

リテラル値または式を渡すときは、次のヘルパー関数を使用して値へのポインタを作成します。

- jsii.String
- jsii.Number
- jsii.Bool
- jsii.Time

一貫性を保つため、独自のコンストラクトを定義する場合も同様にポインターを使用することをお勧めします。例えば、コンストラクトの を文字列へのポインターではなく文字列idとして受け取る方が便利です。

プリミティブ AWS CDK 値や複合型などのオプションの値を扱う場合は、ポインターを明示的にテストして、それらを使用するnil前にポインターがないことを確認する必要があります。Go には、他の言語と同様に、空または欠落した値を処理するのに役立つ「構文的な欠陥」はありません。ただ

し、プロパティバンドルや同様の構造で必須の値が存在することが保証されているため (それ以外の場合は構成が失敗します) 、これらの値を nilでチェックする必要はありません。

コンストラクトと Props

1 つ以上の AWS リソースとそれに関連する属性を表すコンストラクトは、インターフェイスとして Go で表されます。例えば、 awss3.Bucketはインターフェイスです。すべてのコンストラクトには、対応するインターフェイスを実装する構造体awss3.NewBucketを返す などのファクトリ関数があります。

すべてのファクトリ関数は 3 つの引数を取ります。コンストラクトscopeが定義されている (コンストラクトツリー内の親) id、、および propsは、コンストラクトが作成するリソースの設定に使用するキーと値のペアのバンドルです。「属性のバンドル」パターンは、 の他の場所でも使用されます AWS CDK。

Go では、props はコンストラクトごとに特定の構造体タイプで表されます。例えば、 は 型の props 引数awss3.Bucketを取りますawss3.BucketProps。構造体リテラルを使用して props 引数を書き込みます。

```
var bucket = awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
    Versioned: jsii.Bool(true),
})
```

一般的な構造

場合によっては、 AWS CDK は JavaScript 配列または型指定されていないオブジェクトをメソッドへの入力として使用します。 (例えば、 AWS CodeBuildの BuildSpec.fromObject()メソッドを参照してください。) Go では、これらのオブジェクトはそれぞれスライスと空のインターフェイスとして表されます。

CDK には、プリミティブ型を含むスライスを構築jsii.Stringsするための などの変分ヘルパー関数が用意されています。

```
jsii.Strings("One", "Two", "Three")
```

カスタムコンストラクトの開発

Go では、通常、既存のコンストラクトを拡張するよりも、新しいコンストラクトを作成する方が簡単です。まず、新しい構造体タイプを定義し、拡張のようなセマンティクスが必要な場合は、1 つ以

上の既存のタイプを匿名で埋め込みます。追加する新機能のメソッドと、必要なデータを保持するために必要なフィールドを記述します。コンストラクトに必要な場合は、props インターフェイスを定義します。最後に、ファクトリー関数を作成してNewMyConstruct()、コンストラクトのインスタンスを返します。

既存のコンストラクトのデフォルト値を変更したり、インスタンス化時に単純な動作を追加したりするだけでは、そのすべては必要ありません。代わりに、「拡張」するコンストラクトのファクトリ関数を呼び出すファクトリ関数を記述します。例えば、他の CDK 言語では、タイプを上書きして Amazon S3 バケット内のオブジェクトのタイプを強制するTypedBucketコンストラクトを作成しs3.Bucket、新しいタイプのイニシャライザで、指定されたファイル名拡張子のみをバケットに追加することを許可するバケットポリシーを追加できます。Go では、適切なバケットポリシーを追加NewTypedBucketした s3.Bucket (を使用してインスタンス化されたs3.NewBucket) を返すを簡単に記述する方が簡単です。機能はすでに標準バケットコンストラクトで利用できるため、新しいコンストラクトタイプは必要ありません。新しい「コンストラクト」は、より簡単に設定することができます。

構築、合成、デプロイ

は、アプリケーションを実行する前に AWS CDK 自動的にコンパイルします。ただし、エラーをチェックしてテストを実行するために、アプリケーションを手動で構築すると便利です。これを行うには、プロジェクトのルートディレクトリにあるコマンドプロンプトgo buildで を発行します。

コマンドプロンプトgo testで を実行して、作成したテストを実行します。

AWS CDK アプリケーションで定義されたスタックは、合成して個別にデプロイすることも、以下のコマンドを使用してまとめてデプロイすることもできます。通常、プロジェクトを発行するときは、プロジェクトのメインディレクトリにいる必要があります。

- cdk synth: AWS CDK アプリケーションの 1 つ以上のスタックから AWS CloudFormation テンプレートを合成します。
- cdk deploy: AWS CDK アプリケーション内の 1 つ以上のスタックによって定義されたリソースを にデプロイします AWS。

1 つのコマンドで合成またはデプロイする複数のスタックの名前を指定できます。アプリケーションが 1 つのスタックのみを定義している場合は、指定する必要はありません。

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

また、ワイルドカード * (任意の文字数) と ? (任意の 1 文字) を使用して、パターンでスタックを識別することもできます。ワイルドカードを使用する場合は、パターンを引用符で囲みます。そうしないと、シェルは AWS CDK Toolkit に渡す前に、現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

> **Tip**
>
> デプロイする前にスタックを明示的に合成する必要はありません。 はこのステップcdk deployを実行して、最新のコードがデプロイされることを確認します。

cdk コマンドの完全なドキュメントについては、「」を参照してくださいthe section called “AWS CDK ツールキット”。

AWS CDK v1 から AWS CDK v2 への移行

のバージョン 2 AWS Cloud Development Kit (AWS CDK) は、希望するプログラミング言語でのコードとしてのインフラストラクチャの記述を容易にするように設計されています。このトピックでは、の v1 と v2 の間の変更について説明します AWS CDK。

> **Tip**
>
> AWS CDK v1 でデプロイされたスタックを特定するには、awscdk-v1-stack-f Finder ユーティリティを使用します。

AWS CDK v1 から CDK v2 への主な変更は次のとおりです。

- AWS CDK v2 は、コアライブラリを含む AWS コンストラクトライブラリの安定した部分を 1 つのパッケージ に統合しますaws-cdk-lib。デベロッパーは、使用する個々の AWS サービス用に追加のパッケージをインストールする必要がなくなりました。このシングルパッケージアプローチでは、さまざまな CDK ライブラリパッケージのバージョンを同期する必要もありません。

  で使用できる正確なリソースを表す L1 (CfnXXXX) コンストラクトは常に安定しているとみなされ AWS CloudFormation、 に含まれていますaws-cdk-lib。
- コミュニティと協力して新しい L2 または L3 コンストラクトを開発している実験モジュールは、には含まれていませんaws-cdk-lib。代わりに、個々のパッケージとして配布されます。実験パッケージの名前には、alphaサフィックスとセマンティックバージョン番号が付けられます。セマンティックバージョン番号は、互換性がある AWS コンストラクトライブラリの最初のバージョンと一致し、alphaサフィックスも付きます。コンストラクタは、安定と指定されたaws-cdk-lib後に に移行し、メインコンストラクタライブラリが厳密なセマンティックバージョニングに準拠できるようにします。

  安定性はサービスレベルで指定されます。例えば、この書き込みでは L1 コンストラクトのみを持つ Amazon の 1 つ以上の L2 コンストラクトの作成を開始した場合、それらはまず という名前のモジュールに表示されます@aws-cdk/aws-appflow-alpha。 AppFlow L1 その後、新しいコンストラクトが顧客の基本的なニーズを満たすと思われるaws-cdk-libとき、 に移行します。

  モジュールが安定していると指定され、 に組み込まれるとaws-cdk-lib、次の箇条書きで説明するAPIs が追加されます。 BetaN

各実験モジュールの新しいバージョンは、 のすべてのリリースでリリースされます AWS CDK。ただし、ほとんどの場合、同期しておく必要はありません。aws-cdk-lib または実験的なモジュールはいつでもアップグレードできます。ただし、2 つ以上の関連する実験モジュールが相互に依存している場合、同じバージョンである必要があります。

- 新しい機能が追加されている安定したモジュールの場合、新しい APIs (まったく新しいコンストラクトでも、既存のコンストラクトの新しいメソッドやプロパティでも) Beta1 は作業の進行中にサフィックスを受け取ります。 (重大な変更が必要な場合はBeta3、、 Beta2などによって実行されます。) サフィックスのない API のバージョンは、API が安定していると指定されたときに追加されます。その後、最新の (ベータ版または最終版) を除くすべてのメソッドが廃止されます。

  例えば、コンストラクトgrantPower()に新しいメソッドを追加すると、最初は と表示されますgrantPowerBeta1()。重大な変更が必要な場合 (例えば、新しい必須パラメータまたはプロパティ) grantPowerBeta2()、次のバージョンのメソッドには という名前が付けられます。作業が完了し、API が確定されると、 メソッド grantPower() (サフィックスなし) が追加され、BetaN メソッドは非推奨になります。

  すべてのベータ APIs、次のメジャーバージョン (3.0) リリースまで Construct Library に残り、署名は変更されません。使用した場合は非推奨の警告が表示されるため、できるだけ早く API の最終バージョンに移行する必要があります。ただし、今後の AWS CDK 2.x リリースではアプリケーションが破損することはありません。

- Construct クラスは、 から、関連するタイプとともに別のライブラリ AWS CDK に抽出されています。これは、構成プログラミングモデルを他のドメインに適用する作業をサポートするために行われます。独自のコンストラクトを記述する場合、または関連する APIsを使用する場合は、constructsモジュールを依存関係として宣言し、インポートにわずかな変更を加える必要があります。CDK アプリのライフサイクルへのフックなどの高度な機能を使用している場合は、さらに多くの変更が必要になる場合があります。詳細については、「RFC」を参照してください。
- AWS CDK v1.x とそのコンストラクトライブラリの非推奨プロパティ、メソッド、タイプは CDK v2 API から完全に削除されました。サポートされているほとんどの言語では、これらの APIs v1.x で警告を生成するため、代替 APIsに既に移行している可能性があります。CDK v1.x の非推奨 APIs の完全なリストは、 で入手できます GitHub。
- AWS CDK v1.x の特徴フラグによってゲートされた動作は、CDK v2 ではデフォルトで有効になっています。以前の機能フラグはもう必要ではなく、ほとんどの場合はサポートされません。いくつかの は、非常に特定の状況でも CDK v1 の動作に戻すことができます。詳細については、「the section called “機能フラグの更新”」を参照してください。

- CDK v2 では、 にデプロイする環境は、最新のブートストラップスタックを使用してブートストラップする必要があります。レガシーブートストラップスタック (v1 のデフォルト) はサポートされなくなりました。CDK v2 では、最新のスタックの新しいバージョンがさらに必要になりました。既存の環境をアップグレードするには、環境を再ブートストラップします。最新のブートストラップスタックを使用するために、機能フラグや環境変数を設定する必要がなくなりました。

> **Important**
>
> 最新のブートストラップテンプレートは、 によって暗示されているアクセス許可`--cloudformation-execution-policies`を`--trust`リスト内の任意の AWS アカウントに効果的に付与します。デフォルトでは、これにより、ブートストラップされたアカウントの任意のリソースに対する読み取りと書き込みのアクセス許可が拡張されます。ブートストラップスタックには、使い慣れたポリシーと信頼できるアカウントを設定してください。

新しい前提条件

AWS CDK v2 のほとんどの要件は AWS CDK v1.x の要件と同じです。その他の要件を以下に示します。

- TypeScript デベロッパーには、 TypeScript 3.8 以降が必要です。
- CDK v2 で使用するには、CDK Toolkit の新しいバージョンが必要です。CDK v2 が一般公開されたので、v2 は CDK Toolkit のインストール時のデフォルトバージョンです。CDK v1 プロジェクトとの下位互換性があるため、CDK v1 プロジェクトを作成しない限り、以前のバージョンをインストールしておく必要はありません。アップグレードするには、 を発行します`npm install -g aws-cdk`。

AWS CDK v2 デベロッパープレビューからのアップグレード

CDK v2 デベロッパープレビューを使用している場合、プロジェクトには AWS CDK、 などの のリリース候補バージョンに対する依存関係があります`2.0.0-rc1`。これらを に更新し`2.0.0`、プロジェクトにインストールされているモジュールを更新します。

TypeScript

`npm install` または `yarn install`

JavaScript

```
npm install または yarn install
```

Python

```
python -m pip install -r requirements.txt
```

Java

```
mvn package
```

C#

```
dotnet restore
```

Go

```
go get
```

依存関係を更新したら、 を発行`npm update -g aws-cdk`して CDK Toolkit を リリースバージョンに更新します。

AWS CDK v1 から CDK v2 への移行

アプリケーションを AWS CDK v2 に移行するには、まず の機能フラグを更新します`cdk.json`。次に、記述されているプログラミング言語に応じて、必要に応じてアプリケーションの依存関係とインポートを更新します。

最新の v1 への更新

古いバージョンの AWS CDK v1 から最新バージョンの v2 にアップグレードしたお客様は、1 回のステップで多数表示されます。それは確実に可能ですが、数年にわたる変更 (現時点ではすべての進化テストが同じではない可能性があります) でアップグレードするだけでなく、新しいデフォルトや別のコード組織でバージョン間でアップグレードすることもあります。

最も安全なアップグレードエクスペリエンスを提供し、予期しない変更の原因をより簡単に診断するには、2 つのステップを分けることをお勧めします。まず最新の v1 バージョンにアップグレードしてから、v2 に切り替えることです。

機能フラグの更新

次の v1 機能フラグが存在するcdk.json場合は、 から削除します。これらは AWS CDK v2 でデフォルトでアクティブになっているためです。インフラストラクチャにとって古い影響が重要な場合は、ソースコードを変更する必要があります。詳細については、「」のフラグのリストを参照してください GitHub。

- `@aws-cdk/core:enableStackNameDuplicates`
- `aws-cdk:enableDiffNoFail`
- `@aws-cdk/aws-ecr-assets:dockerIgnoreSupport`
- `@aws-cdk/aws-secretsmanager:parseOwnedSecretName`
- `@aws-cdk/aws-kms:defaultKeyPolicies`
- `@aws-cdk/aws-s3:grantWriteWithoutAcl`
- `@aws-cdk/aws-efs:defaultEncryptionAtRest`

特定の v1 の動作に戻すには、いくつかの AWS CDK v1 機能フラグを に設定できます。完全なリファレンス GitHub についてはfalse、the section called “v1 の動作に戻す”「」または「」のリストを参照してください。

どちらのタイプのフラグでも、 cdk diff コマンドを使用して合成されたテンプレートの変更を調べ、これらのフラグのいずれかの変更がインフラストラクチャに影響するかどうかを確認します。

CDK ツールキットの互換性

CDK v2 には、CDK Toolkit の v2 以降が必要です。このバージョンは CDK v1 アプリと下位互換性があります。したがって、グローバルにインストールされた単一のバージョンの CDK Toolkit を、v1 または v2 のどちらを使用する場合でも、すべての AWS CDK プロジェクトで使用できます。例外は、CDK Toolkit v2 は CDK v2 プロジェクトのみを作成することです。

v1 と v2 の両方の CDK プロジェクトを作成する必要がある場合は、CDK Toolkit v2 をグローバルにインストールしないでください。 (既にインストールされている場合は削除します:)npm remove -g aws-cdk。CDK Toolkit を呼び出すには、 npxを使用して CDK Toolkit の v1 または v2 を必要に応じて実行します。

```
npx aws-cdk@1.x init app --language typescript
npx aws-cdk@2.x init app --language typescript
```

Tip

コマンドラインエイリアスを設定して、 cdkおよび cdk1 コマンドを使用して CDK Toolkit の目的のバージョンを呼び出すことができます。

macOS/Linux

```
alias cdk1="npx aws-cdk@1.x"
alias cdk="npx aws-cdk@2.x"
```

Windows

```
doskey cdk1=npx aws-cdk@1.x $*
doskey cdk=npx aws-cdk@2.x $*
```

依存関係とインポートの更新

アプリケーションの依存関係を更新し、新しいパッケージをインストールします。最後に、コード内のインポートを更新します。

TypeScript

アプリケーション

CDK アプリの場合は、package.json次のように を更新します。v1 スタイルの個々の安定モジュールへの依存関係を削除し、アプリケーションaws-cdk-libに必要な最小バージョンの を設定します (ここでは 2.0.0)。

実験コンストラクトは、名前が で終わる個別のバージョン管理されたパッケージalphaとアルファバージョン番号で提供されます。アルファバージョン番号は、互換性aws-cdk-libがあるの最初のリリースに対応しています。ここでは、v2.0.0-alpha.1 aws-codestarに固定されています。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
```

```
}
```

ライブラリを構築する

コンストラクトライブラリaws-cdk-libの場合は、アプリケーションに必要な の最小バージョン (ここでは 2.0.0) を確立し、package.json次のように更新します。

は、ピア依存関係と開発依存関係の両方としてaws-cdk-lib表示されます。

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

> **Note**
>
> v2 互換ライブラリをリリースするときは、ライブラリのバージョン番号にメジャーバージョンバンプを実行する必要があります。これは、ライブラリコンシューマーにとって重大な変更であるためです。CDK v1 と v2 の両方を 1 つのライブラリでサポートすることはできません。引き続き v1 を使用しているお客様をサポートするには、以前のリリースを並行して維持するか、v2 用の新しいパッケージを作成します。
> AWS CDK v1 の顧客を引き続きサポートする期間はお客様によって異なります。CDK v1 自体のライフサイクルからキューを取得できます。CDK v1 自体は 2022 年 6 月 1 日にメンテナンスを開始し、2023 年 6 月 1 end-of-life 日に開始されます。詳細については、AWS CDK 「メンテナンスポリシー」を参照してください。

ライブラリとアプリの両方

npm install または を実行して、新しい依存関係をインストールしますyarn install。

インポートを変更して、新しいconstructsモジュールConstruct、 Appや などのコアタイプ、および の最上位レベルStackからインポートしaws-cdk-lib、 の名前空間から使用す

るサービス用の安定したコンストラクトライブラリモジュールをインポートしますaws-cdk-lib。

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';                  // core constructs
import { aws_s3 as s3 } from 'aws-cdk-lib';                // stable module
import * as codestar from '@aws-cdk/aws-codestar-alpha';  // experimental module
```

JavaScript

package.json 次のように を更新します。v1 スタイルの個々の安定モジュールへの依存関係を削除し、アプリケーションaws-cdk-libに必要な最小バージョンの を設定します (ここでは 2.0.0)。

実験コンストラクトは、名前が で終わる個別のバージョン管理されたパッケージalphaとアルファバージョン番号で提供されます。アルファバージョン番号は、互換性aws-cdk-libがあるの最初のリリースに対応しています。ここでは、v2.0.0-alpha.1 aws-codestarに固定されています。

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
}
```

npm install または を実行して、新しい依存関係をインストールしますyarn install。

アプリケーションのインポートを変更して、次の操作を行います。

- 新しいconstructsモジュールConstructからのインポート
- App や などのコアタイプを の最上位Stackレベルからインポートする aws-cdk-lib
- の 名前空間から AWS コンストラクトライブラリモジュールをインポートする aws-cdk-lib

```
const { Construct } = require('constructs');
const { App, Stack } = require('aws-cdk-lib');              // core constructs
const s3 = require('aws-cdk-lib').aws_s3;                   // stable module
const codestar = require('@aws-cdk/aws-codestar-alpha');    // experimental module
```

Python

setup.py 次のように requirements.txtまたは install_requiresの定義を更新します。v1 スタイルの個々の安定モジュールへの依存関係を削除します。

実験コンストラクトは、名前が で終わる個別のバージョン管理されたパッケージalphaとアルファバージョン番号で提供されます。アルファバージョン番号は、互換性aws-cdk-libがあるの最初のリリースに対応しています。ここでは、v2.0.0alpha1 aws-codestarに固定されています。

```
install_requires=[
     "aws-cdk-lib>=2.0.0",
     "constructs>=10.0.0",
     "aws-cdk.aws-codestar-alpha>=2.0.0alpha1",
     # ...
],
```

> **Tip**
>
> を使用して、アプリケーションの仮想環境に既にインストールされている他のバージョンの AWS CDK モジュールをアンインストールしますpip uninstall。次に、を使用して新しい依存関係をインストールしますpython -m pip install -r requirements.txt。

アプリケーションのインポートを変更して、次の操作を行います。

- 新しいconstructsモジュールConstructからのインポート
- App や などのコアタイプを の最上位Stackレベルからインポートする aws_cdk
- の 名前空間から AWS 構成ライブラリモジュールをインポートする aws_cdk

```
from constructs import Construct
from aws_cdk import App, Stack                    # core constructs
from aws_cdk import aws_s3 as s3                  # stable module
import aws_cdk.aws_codestar_alpha as codestar     # experimental module

# ...
```

```
class MyConstruct(Construct):
    # ...

class MyStack(Stack):
    # ...

s3.Bucket(...)
```

Java

でpom.xml、安定したモジュールのすべてのsoftware.amazon.awscdk依存関係を削除し、(software.constructsの場合Construct) と の依存関係に置き換えますsoftware.amazon.awscdk。

実験コンストラクトは、名前が で終わる個別のバージョン管理されたパッケージalphaとアルファバージョン番号で提供されます。アルファバージョン番号は、互換性aws-cdk-libがあるの最初のリリースに対応しています。ここでは、v2.0.0-alpha.1 aws-codestarに固定されています。

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0</version>
</dependency><dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>code-star-alpha</artifactId>
    <version>2.0.0-alpha.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
</dependency>
```

を実行して、新しい依存関係をインストールしますmvn package。

コードを変更して、次の操作を行います。

- 新しいsoftware.constructsライブラリConstructからのインポート
- Appからの Stackや などのコアクラスのインポート software.amazon.awscdk
- からのサービスコンストラクトのインポート software.amazon.awscdk.services

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.App;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.codestar.alpha.GitHubRepository;
```

C#

C# CDK アプリケーションの依存関係をアップグレードする最も簡単な方法は、.csprojファイルを手動で編集することです。安定Amazon.CDK.*しているパッケージリファレンスをすべて削除し、 Amazon.CDK.Libおよび Constructsパッケージへのリファレンスに置き換えます。

実験コンストラクトは、名前が で終わる個別のバージョン管理されたパッケージalphaとアルファバージョン番号で提供されます。アルファバージョン番号は、互換性aws-cdk-libがあるの最初のリリースに対応しています。ここでは、v2.0.0-alpha.1 aws-codestarに固定されています。

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0" />
<PackageReference Include="Amazon.CDK.AWS.Codestar.Alpha" Version="2.0.0-alpha.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

を実行して、新しい依存関係をインストールしますdotnet restore。

ソースファイルのインポートを次のように変更します。

```
using Constructs;                   // for Construct class
using Amazon.CDK;                   // for core classes like App and Stack
using Amazon.CDK.AWS.S3;            // for stable constructs like Bucket
using Amazon.CDK.Codestar.Alpha;    // for experimental constructs
```

Go

依存関係go getを最新バージョンに更新し、プロジェクトの .mod ファイルを更新する問題。

デプロイ前に移行したアプリケーションをテストする

スタックをデプロイする前に、 cdk diffを使用してリソースへの予期しない変更がないか確認します。論理 IDs の変更 (リソースの置き換えによる) は想定されません。

予想される変更には以下が含まれますが、これらに限定されません。

- `CDKMetadata` リソースへの変更。
- アセットハッシュを更新しました。
- 新しいスタイルのスタック合成に関連する変更。アプリケーションが v1 でレガシースタックシンセライザーを使用した場合に適用されます。(CDK v2 はレガシースタック合成子をサポートしていません)。
- `CheckBootstrapVersion` ルールの追加。

予期しない変更は、通常、 AWS CDK v2 自体へのアップグレードが原因ではありません。通常、これらは機能フラグによって以前に変更された非推奨の動作の結果です。これは、約 1.85.x より前のバージョンの CDK からアップグレードする兆候です。最新の v1.x リリースへのアップグレードと同じ変更が表示されます。通常、これを解決するには、以下を実行します。

1. アプリケーションを最新の v1.x リリースにアップグレードする
2. 機能フラグを削除する
3. 必要に応じてコードを修正する
4. デプロイ
5. v2 へのアップグレード

> **Note**
>
> アップグレードしたアプリが 2 段階アップグレード後にデプロイできない場合は、問題を報告します。

アプリケーションにスタックをデプロイする準備ができたら、最初にコピーをデプロイしてテストできるようにすることを検討してください。これを行う最も簡単な方法は、別のリージョンにデプロイすることです。ただし、スタックIDs を変更することもできます。テスト後は、必ず を使用してテストコピーを破棄してくださいcdk destroy。

トラブルシューティング

TypeScript **`'from' expected`** インポートの または **`';' expected`** エラー

TypeScript 3.8 以降にアップグレードします。

「cdk bootstrap」を実行する

次のようなエラーが表示された場合：

```
#  MyStack failed: Error: MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not
 found. Has the environment been bootstrapped? Please run 'cdk bootstrap' (see https://
docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
    at CloudFormationDeployments.validateBootstrapStackVersion (.../aws-cdk/lib/api/
cloudformation-deployments.ts:323:13)
    at processTicksAndRejections (internal/process/task_queues.js:97:5)
MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not found. Has the environment
 been bootstrapped? Please run 'cdk bootstrap' (see https://docs.aws.amazon.com/cdk/
latest/guide/bootstrapping.html)
```

AWS CDK v2 には、更新されたブートストラップスタックが必要です。さらに、すべての v2 デプロイにはブートストラップリソースが必要です。(v1 では、ブートストラップなしでシンプルなスタックをデプロイできます) 。詳細については、「the section called “ブートストラッピング”」を参照してください。

v1 スタックの検索

CDK アプリケーションを v1 から v2 に移行する場合、v1 を使用して作成されたデプロイされた AWS CloudFormation スタックを特定できます。これを行うには、次のコマンドを実行します。

```
npx awscdk-v1-stack-finder
```

使用の詳細については、awscdk-v1-stack-f Finder README を参照してください。

既存のリソースと AWS CloudFormation テンプレートを に移行する AWS CDK

CDK 移行機能は のプレビューリリースであり AWS CDK 、変更される可能性があります。

AWS Cloud Development Kit (AWS CDK) コマンドラインインターフェイス (AWS CDK CLI) を使用して、デプロイされた AWS リソース、デプロイされた AWS CloudFormation スタック、およびローカル AWS CloudFormation テンプレートを に移行します AWS CDK。

トピック

移行の仕組み

`cdk migrate` コマンドを使用して AWS CDK CLI、次のソースから移行します。

- デプロイされた AWS リソース。
- デプロイされた AWS CloudFormation スタック。
- ローカル AWS CloudFormation テンプレート。

デプロイされた AWS リソース

AWS CloudFormation スタックに関連付けられていない特定の環境 (AWS アカウント および AWS リージョン) からデプロイされた AWS リソースを移行できます。

は、IaC AWS CDK CLIジェネレーターサービスを利用して AWS 環境内のリソースをスキャンし、リソースの詳細を収集します。 IaC IaC ジェネレーターの詳細については、AWS CloudFormation 「 ユーザーガイド」の「既存のリソースのテンプレートの生成」を参照してください。

リソースの詳細を AWS CDK CLI収集した後、 は、移行されたリソースを含む単一のスタックを含む新しい CDK アプリケーションを作成します。

デプロイされた AWS CloudFormation スタック

1 つの AWS CloudFormation スタックを新しい AWS CDK アプリケーションに移行できます。AWS CDK CLI はスタックのテンプレートを取得し AWS CloudFormation 、新しい CDK アプリを作成します。CDK アプリは、移行されたスタックを含む単一の AWS CloudFormation スタックで構成されます。

ローカル AWS CloudFormation テンプレート

ローカル AWS CloudFormation テンプレートから移行できます。ローカルテンプレートには、デプロイされたリソースが含まれている場合と含まれない場合があります。CLI は、 リソースに 1 AWS CDK つのスタックを含む新しい CDK アプリを作成します。

移行後、CDK アプリケーションを に管理、変更、デプロイ AWS CloudFormation して、 リソースをプロビジョニングまたは更新できます。

CDK 移行の利点

リソースを に移行する AWS CDK のは、これまで手動プロセスであり、開始 AWS CDK するには AWS CloudFormation および との時間と専門知識が必要です。CDK Migrate を使用すると AWS CDK CLI、 は移行作業の大部分をわずかな時間で容易にします。CDK 移行では、 を使用して で新規および既存のアプリケーションの開発と管理 AWS CDK をすぐに開始できます AWS。

考慮事項

一般的な考慮事項

CDK 移行と CDK インポート

`cdk import` コマンドは、デプロイされたリソースを新規または既存の CDK アプリケーションにインポートできます。インポートする場合、各リソースはアプリで L1 コンストラクトとして

手動で定義する必要があります。を使用してcdk import、一度に 1 つ以上のリソースを新規または既存の CDK アプリにインポートすることをお勧めします。詳細については、「スタックへの既存リソースのインポート」を参照してください。

cdk migrate コマンドは、デプロイされたリソース、デプロイされた AWS CloudFormation スタック、またはローカル AWS CloudFormation テンプレートから新しい CDK アプリに移行します。移行中、 は AWS CDK CLI cdk importを使用してリソースを新しい CDK アプリにインポートします。また、 AWS CDK CLIはリソースごとに L1 コンストラクトを生成します。サポートされている移行ソースから新しい AWS CDK アプリケーションにインポートcdk migrateする場合は、 を使用することをお勧めします。

CDK Migrate は L1 コンストラクトのみを作成します

新しく作成された CDK アプリには L1 コンストラクトのみが含まれます。移行後にアプリに上位レベルのコンストラクトを追加できます。

CDK Migrate は、単一のスタックを含む CDK アプリケーションを作成します。

新しく作成された CDK アプリには、1 つのスタックが含まれます。

デプロイされたリソースを移行する場合、移行されたすべてのリソースは新しい CDK アプリの 1 つのスタックに含まれます。

AWS CloudFormation スタックを移行する場合、新しい CDK アプリでは 1 つの AWS CloudFormation スタックのみを 1 つのスタックに移行できます。

アセットの移行

AWS Lambda コードなどのプロジェクトアセットは、新しい CDK アプリに直接移行されません。移行後、アセット値を指定して CDK アプリに含めることができます。

ステートフルリソースの移行

データベースや Amazon Simple Storage Service (Amazon S3) バケットなどのステートフルリソースを移行する場合、新しいリソースを作成する代わりに、ほとんどの場合、既存のリソースを移行する必要があります。

ステートフルリソースを移行して保持するには、次の手順を実行します。

- ステートフルリソースがインポートをサポートしていることを確認します。詳細については、「 ユーザーガイド」の「リソースタイプのサポートAWS CloudFormation 」を参照してください。

- 移行後、新しい CDK アプリで移行されたリソースの論理 ID が、デプロイされたリソースの論理 ID と一致することを確認します。
- AWS CloudFormation スタックから移行する場合は、新しい CDK アプリのスタック名がスタックと一致する AWS CloudFormation ことを確認します。
- 移行したリソースの同じ AWS アカウントと を使用して CDK AWS リージョン アプリケーションをデプロイします。

AWS CloudFormation テンプレートから移行する際の考慮事項

CDK 移行が単一テンプレート移行をサポート

AWS CloudFormation テンプレートを移行するときは、移行するテンプレートを 1 つ選択できます。ネストされたテンプレートはサポートされていません。

組み込み関数を使用したテンプレートの移行

組み込み関数を使用する AWS CloudFormation テンプレートから移行する場合、 AWS CDK CLI は Fn クラスを使用してロジックを CDK アプリに移行しようとします。詳細については、「 API リファレンス」の「クラス Fn」を参照してください。 AWS Cloud Development Kit (AWS CDK)

デプロイされたリソースから移行する際の考慮事項

スキャンの制限

環境のリソースをスキャンする場合、IaC ジェネレーターには、スキャン時に取得できるデータとクォータの制限に特定の制限があります。詳細については、「 AWS CloudFormation ユーザーガイドhttps://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/generate-IaC.html#generate-template-considerations」の「考慮事項」を参照してください。

前提条件

`cdk migrate` コマンドを使用する前に、次の操作を行います。

1. を使用して認証を確立します AWS。手順については、「ステップ 2: プログラムによるアクセスを設定する」を参照してください。
2. AWS CDK CLI をインストールまたはアップグレードします。インストール手順については、「ステップ 3: をインストールする AWS CDKCLI」を参照してください。

CDK 移行の開始方法

開始するには、任意のディレクトリから `cdk migrate` コマンドを実行します AWS CDK CLI。実行する移行のタイプに応じて、必須オプションとオプションを指定します。

で使用できるオプションの完全なリストと説明については、`cdk migrate`「」を参照してくださいcdk migrate。

以下は、提供すべき重要なオプションです。

スタックの名前

唯一の必須オプションは です`--stack-name`。このオプションを使用して、移行後に AWS CDK アプリ内で作成されるスタックの名前を指定します。スタック名は、デプロイ時に AWS CloudFormation スタックの名前としても使用されます。

[言語]

を使用して`--language`、新しい CDK アプリケーションのプログラミング言語を指定します。

AWS アカウントと AWS リージョン

は、デフォルトのソースから AWS アカウントと AWS リージョン 情報 AWS CDK CLIを取得します。詳細については、「ステップ 2: プログラムによるアクセスを設定する」を参照してください。`--account` および `--region`オプションを で使用`cdk migrate`して、他の値を指定できます。

新しい CDK プロジェクトの出力ディレクトリ

デフォルトでは、 AWS CDK CLI は作業ディレクトリに新しい CDK プロジェクトを作成し、 で指定した値を使用してプロジェクトフォルダ`--stack-name`に名前を付けます。同じ名前のフォルダが現在存在する場合、 AWS CDK CLIはそのフォルダを上書きします。

`--output-path` オプションを使用して、新しい CDK プロジェクトフォルダに別の出力パスを指定できます。

移行ソース

移行元のソースを指定するオプションを指定します。

- `--from-path` – ローカル AWS CloudFormation テンプレートから移行します。
- `--from-scan` – AWS アカウントと にデプロイされたリソースから移行します AWS リージョン。

- `--from-stack` – AWS CloudFormation スタックから移行します。

  移行ソースに応じて、`cdk migrate`コマンドをカスタマイズするための追加オプションを指定できます。

AWS CloudFormation スタックからの移行

デプロイされた AWS CloudFormation スタックから移行するには、 `--from-stack`オプションを指定します。デプロイした AWS CloudFormation スタックの名前を で指定します`--stack-name`。以下に例を示します。

```
$ cdk migrate --from-stack --stack-name "myCloudFormationStack"
```

AWS CDK CLI は以下を実行します。

1. デプロイされたスタックの AWS CloudFormation テンプレートを取得します。
2. `cdk init` を実行して新しい CDK アプリケーションを初期化します。
3. 移行したスタックを含む AWS CloudFormation スタックを CDK アプリ内に作成します。

デプロイされた AWS CloudFormation スタックから移行すると、 は AWS CDK CLIデプロイされたリソースIDs とデプロイされた AWS CloudFormation スタック名を、新しい CDK アプリの移行されたリソースとスタックと照合しようとします。

移行後、CDK アプリを正常に管理および変更できます。デプロイすると、 AWS CloudFormation は一致する AWS CloudFormation スタック名により、デプロイを AWS CloudFormation スタック更新として識別します。一致する論理 IDs を持つリソースが更新されます。デプロイの詳細については、「」を参照してくださいCDK アプリケーションの管理とデプロイ。

AWS CloudFormation テンプレートからの移行

CDK 移行では、 `JSON`または でフォーマットされた AWS CloudFormation テンプレートからの移行がサポートされていますYAML。

ローカル AWS CloudFormation テンプレートから移行するには、 `--from-path`オプションを使用してローカルテンプレートへのパスを指定します。また、必要な`--stack-name`オプションを指定する必要があります。以下に例を示します。

```
$ cdk migrate --from-path "./template.json" --stack-name "myCloudFormationStack"
```

AWS CDK CLI は以下を実行します。

1. ローカル AWS CloudFormation テンプレートを取得します。
2. `cdk init` を実行して新しい CDK アプリケーションを初期化します。
3. 移行した AWS CloudFormation テンプレートを含むスタックを CDK アプリ内に作成します。

移行後、CDK アプリを正常に管理および変更できます。デプロイには、次のオプションがあります。

- AWS CloudFormation スタックの更新 — ローカル AWS CloudFormation テンプレートが以前にデプロイされていた場合は、デプロイされた AWS CloudFormation スタックを更新できます。
- 新しい AWS CloudFormation スタックをデプロイする – ローカル AWS CloudFormation テンプレートが一度もデプロイされなかった場合、または以前にデプロイしたテンプレートから新しいスタックを作成する場合は、新しい AWS CloudFormation スタックをデプロイできます。

AWS SAM テンプレートからの移行

AWS Serverless Application Model （ AWS SAM) テンプレートから移行するには、まず AWS CloudFormation テンプレートに変換するか、 をデプロイして AWS CloudFormation スタックを作成する必要があります。

AWS SAM テンプレートを に変換するには AWS CloudFormation、 `sam validate --debug` コマンドを使用できます AWS SAM CLI。このコマンドを実行する前に`lint`、 `samconfig.toml` ファイル`false`で を に設定する必要がある場合があります。

AWS CloudFormation スタックに変換するには、 を使用して AWS SAM テンプレートをデプロイします AWS SAM CLI。次に、デプロイされたスタックから移行します。

デプロイされたリソースからの移行

デプロイされた AWS リソースから移行するには、 `--from-scan`オプションを指定します。また、必要な`--stack-name`オプションを指定する必要があります。以下に例を示します。

```
$ cdk migrate --from-scan --stack-name "myCloudFormationStack"
```

AWS CDK CLI は以下を実行します。

1. アカウントをスキャンしてリソースとプロパティの詳細を確認する – AWS CDK は IaC ジェネレータCLIーを使用してアカウントをスキャンし、詳細を収集します。
2. テンプレートの生成 AWS CloudFormation – スキャン後、 AWS CDK CLI は IaC ジェネレーターを使用して AWS CloudFormation テンプレートを作成します。
3. 新しい CDK アプリケーションを初期化してテンプレートを移行する – AWS CDK は、新しい AWS CDK アプリケーションを初期化`cdk init`するためにCLI実行し、 AWS CloudFormation テンプレートを 1 つのスタックとして CDK アプリケーションに移行します。

フィルターを使用する

デフォルトでは、 AWS CDK CLI は AWS 環境全体をスキャンし、IaC ジェネレーターの最大クォータ制限までリソースを移行します。で AWS CDK CLIフィルターを指定して、リソースをアカウントから新しい CDK アプリに移行する基準を指定できます。詳細については、「`--filter`」を参照してください。

IaC ジェネレーターによるリソースのスキャン

アカウントのリソース数によっては、スキャンに数分かかる場合があります。スキャンプロセス中に進行状況バーが表示されます。

サポートされているリソースタイプ

は、IaC AWS CDK CLIジェネレーターでサポートされているリソースを移行します。詳細なリストについては、「 ユーザーガイド」の「リソースタイプのサポートAWS CloudFormation 」を参照してください。

書き込み専用プロパティの解決

サポートされているリソースには、書き込み専用プロパティが含まれているものがあります。これらのプロパティは、 プロパティを設定するために に書き込むことができますが、IaC ジェネレーターが読み取ることも、 値を取得 AWS CloudFormation することもできません。例えば、データベースパスワードの指定に使用されるプロパティは、セキュリティ上の理由から書き込み専用である場合があります。

移行中にリソースをスキャンする場合、IaC ジェネレーターは書き込み専用プロパティを含む可能性のあるリソースを検出し、次のいずれかのタイプに分類します。

- MUTUALLY_EXCLUSIVE_PROPERTIES – これらは、交換可能で同様の目的を果たす特定のリソースの書き込み専用プロパティです。リソースを設定するには、相互に排他的なプロパティの 1 つが必要です。例えば、 AWS::Lambda::Functionリソースの S3Bucket、ImageUri、および ZipFileプロパティは、相互に排他的な書き込み専用プロパティです。これらのいずれかを使用して関数アセットを指定できますが、いずれかを使用する必要があります。
- MUTUALLY_EXCLUSIVE_TYPES – これらは、複数の設定タイプを受け入れる書き込み専用プロパティとして必要です。例えば、 AWS::ApiGateway::RestApiリソースの Bodyプロパティは、オブジェクトまたは文字列タイプを受け入れます。
- UNSUPPORTED_PROPERTIES – これらは書き込み専用プロパティであり、他の 2 つのカテゴリには含まれません。オブジェクトの配列を受け入れるオプションプロパティまたは必須プロパティのいずれかです。

書き込み専用プロパティと、デプロイされたリソースをスキャンして AWS CloudFormation テンプレートを作成するときに IaC ジェネレーターがそれらを管理する方法の詳細については、AWS CloudFormation 「 ユーザーガイド」のIaC ジェネレーターと書き込み専用プロパティ」を参照してください。

移行後、新しい CDK アプリで書き込み専用プロパティ値を指定する必要があります。は CDK AWS CDK CLIプロジェクトの ReadMe ファイルに警告セクションを追加して、IaC ジェネレーターによって識別された書き込み専用プロパティを文書化します。以下に例を示します。

```
# Welcome to your CDK TypeScript project
...
## Warnings
### Write-only properties
Write-only properties are resource property values that can be written to but can't be
 read by AWS CloudFormation or CDK Migrate. For more information, see [IaC generator
 and write-only properties](https://docs.aws.amazon.com/AWSCloudFormation/latest/
UserGuide/generate-IaC-write-only-properties.html).

Write-only properties discovered during migration are organized here by resource ID and
 categorized by write-only property type. Resolve write-only properties by providing
 property values in your CDK app. For guidance, see [Resolve write-only properties]
(https://docs.aws.amazon.com/cdk/v2/guide/migrate.html#migrate-resources-writeonly).
### MyLambdaFunction
- **UNSUPPORTED_PROPERTIES**:
  - SnapStart/ApplyOn: Applying SnapStart setting on function resource type.Possible
 values: [PublishedVersions, None]
This property can be replaced with other types
```

```
  - Code/S3ObjectVersion: For versioned objects, the version of the deployment package
 object to use.
This property can be replaced with other exclusive properties
- **MUTUALLY_EXCLUSIVE_PROPERTIES**:
  - Code/S3Bucket: An Amazon S3 bucket in the same AWS Region as your function. The
 bucket can be in a different AWS account.
This property can be replaced with other exclusive properties
  - Code/S3Key: The Amazon S3 key of the deployment package.
This property can be replaced with other exclusive properties
```

- 警告は、関連付けられているリソースの論理 ID を識別する見出しの下に整理されます。
- 警告はタイプ別に分類されます。これらのタイプは IaC ジェネレーターから直接取得されます。

書き込み専用プロパティを解決するには

1. CDK プロジェクトの ReadMeファイルの警告セクションから解決する書き込み専用プロパティを特定します。ここでは、書き込み専用プロパティを含む可能性のある CDK アプリのリソースをメモし、検出された書き込み専用プロパティタイプを特定できます。

   a. の場合MUTUALLY_EXCLUSIVE_PROPERTIES、 AWS CDK アプリで設定する相互に排他的なプロパティを決定します。

   b. ではMUTUALLY_EXCLUSIVE_TYPES、プロパティの設定に使用する受け入れたタイプを決定します。

   c. の場合UNSUPPORTED_PROPERTIES、 プロパティがオプションか必須かを判断します。次に、必要に応じて を設定します。

2. IaC ジェネレーターと書き込み専用プロパティのガイダンスを使用して、警告タイプの意味を参照してください。

3. CDK アプリでは、解決する書き込み専用プロパティ値もアプリの Propsセクションで指定されます。ここに正しい値を入力します。プロパティの説明とガイダンスについては、 AWS CDK API リファレンス を参照してください。

   以下は、解決する 2 つの書き込み専用プロパティを持つ、移行された CDK アプリ内の Propsセクションの例です。

   ```
   export interface MyTestAppStackProps extends cdk.StackProps {
     /**
      * The Amazon S3 key of the deployment package.
      */
   ```

```
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Keym8P82: string;
  /**
   * An Amazon S3 bucket in the same AWS Region as your function. The bucket can be
 in a different AWS account.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Bucketzidw8: string;
}
```

書き込み専用プロパティ値をすべて解決したら、デプロイの準備が整います。

migrate.json ファイル

は、移行中に AWS CDK プロジェクトに `migrate.json` ファイル AWS CDK CLIを作成します。このファイルには、デプロイされたリソースのリファレンス情報が含まれています。CDK アプリケーションを初めてデプロイすると、 AWS CDK CLI はこのファイルを使用してデプロイされたリソースを参照し、リソースを新しい AWS CloudFormation スタックに関連付けて、ファイルを削除します。

CDK アプリケーションの管理とデプロイ

に移行する場合 AWS CDK、新しい CDK アプリはすぐにデプロイできない場合があります。このトピックでは、新しい CDK アプリケーションの管理とデプロイ時に考慮すべきアクション項目について説明します。

デプロイの準備

デプロイする前に、CDK アプリを準備する必要があります。

アプリを合成する

`cdk synth` コマンドを使用して、CDK アプリケーションのスタックを テンプレートに AWS CloudFormation 合成します。

デプロイされた AWS CloudFormation スタックまたはテンプレートから移行した場合は、合成されたテンプレートと移行されたテンプレートを比較して、リソースとプロパティの値を確認できます。

`cdk synth` の詳細については、「スタックの合成」を参照してください。

差分を実行する

デプロイされた AWS CloudFormation スタックから移行した場合は、cdk diff コマンドを使用して、新しい CDK アプリケーションのスタックと比較できます。

cdk 差分の詳細については、「」を参照してくださいスタックの比較。

環境のブートストラップ

AWS 環境から初めてデプロイする場合は、 `cdk bootstrap`を使用して環境を準備します。詳細については、「ブートストラッピング」を参照してください。

CDK アプリをデプロイする

CDK アプリをデプロイすると、 AWS CDK CLI は AWS CloudFormation サービスを利用してリソースをプロビジョニングします。リソースは CDK アプリの 1 つのスタックにバンドルされ、1 つの AWS CloudFormation スタックとしてデプロイされます。

移行元に応じて、 をデプロイして新しい AWS CloudFormation スタックを作成するか、既存の AWS CloudFormation スタックを更新できます。

デプロイして新しい AWS CloudFormation スタックを作成する

デプロイされたリソースから移行した場合、 AWS CDK CLIはデプロイ時に新しい AWS CloudFormation スタックを自動的に作成します。デプロイされたリソースは新しい AWS CloudFormation スタックに含まれます。

一度もデプロイされなかったローカル AWS CloudFormation テンプレートから移行した場合、AWS CDK CLIはデプロイ時に新しい AWS CloudFormation スタックを自動的に作成します。

以前にデプロイされたデプロイ済み AWS CloudFormation スタックまたはローカル AWS CloudFormation テンプレートから移行した場合は、 をデプロイして新しい AWS CloudFormation スタックを作成できます。新しいスタックを作成するには、次の手順を実行します。

- 新しい AWS 環境にデプロイします。これは、別の AWS アカウントを使用するか、別の にデプロイすることで構成されます AWS リージョン。
- 移行したスタックまたはテンプレートの同じ AWS 環境に新しいスタックをデプロイする場合は、CDK アプリケーションのスタック名を新しい値に変更する必要があります。また、CDK アプリでリソースのすべての論理 IDsを変更する必要があります。その後、同じ環境にデプロイして、新しいスタックと新しいリソースを作成できます。

デプロイして既存の AWS CloudFormation スタックを更新する

以前にデプロイされたデプロイ済み AWS CloudFormation スタックまたはローカル AWS CloudFormation テンプレートから移行した場合は、 をデプロイして既存の AWS CloudFormation スタックを更新できます。

CDK アプリのスタック名がデプロイされたスタックの AWS CloudFormation スタック名と一致していることを検証し、同じ AWS 環境にデプロイします。

のセキュリティ認証情報を設定する AWS CDKCLI

を使用してローカル環境でアプリケーション AWS Cloud Development Kit (AWS CDK) を開発する場合、主に コマンドラインインターフェイス (AWS CDK CLI) を使用して AWS CDK とやり取りし、CDK スタック AWS をデプロイおよび管理します。CDK を使用して CLIを操作するには AWS、ローカルマシンでセキュリティ認証情報を設定する必要があります。これにより、 AWS 自分が誰であり、どのようなアクセス許可を持っているかがわかります。

セキュリティ認証情報の詳細については、「IAM ユーザーガイド」のAWS 「 セキュリティ認証情報」を参照してください。

トピック

前提条件

AWS または を初めて使用する場合は AWS CDK、このセクションの前提条件を完了してください。

AWS アカウント と管理ユーザーを作成する

セキュリティ認証情報による認証方法を決定する前に、ユーザーまたは組織には AWS アカウントと管理ユーザーが必要です。詳細については、「IAM ユーザーガイド」の「IAM のセットアップ」を参照してください。

IAM は、 AWS コンソール、 AWS Command Line Interface （ AWS CLI ）、または関連する のアプリケーションインターフェイス (APIs) など、さまざまな方法で管理できますSDKs。CDK で IAM を使用する場合はCLI、主に を使用してセキュリティ認証情報 AWS CLI を設定および管理します。詳細については、「IAM ユーザーガイド」の「 (AWS Command Line InterfaceCLI ）」および「 Software Development Kits (SDKs ）」を参照してください。

ユーザーの作成および管理方法を決定する

AWS アカウント および管理ユーザーを作成したら、ユーザーの最小特権アクセス許可の概念に従って、ユーザーを管理する方法を決定する必要があります。これは IAM のベストプラクティスの推奨

事項です。ユーザーを管理するさまざまな方法については、IAM ユーザーガイドのAWS 「ID 管理の概要: ユーザー」を参照してください。

組織にユーザーの管理方法がある場合は、そのガイダンスに従ってください。それ以外の場合は、AWS IAM Identity Center を使用してユーザーを作成および管理することをお勧めします。IAM Identity Center を使用すると、一元管理されたサービスから AWS アカウント、ユーザー、およびアクセス許可を管理できます。また、 での認証に短期的な認証情報をユーザーに提供することもできます AWS。入門については、「 ユーザーガイド」の「IAM Identity Center とはAWS IAM Identity Center 」を参照してください。

ユーザーの作成を開始したら、ローカルマシンでセキュリティ認証情報を設定する必要があります。ユーザーは を使用してこれ AWS CLI を行うことができます。

のインストール AWS CLI

ユーザーとして、 を使用してローカルマシンで設定ファイルと認証情報ファイル AWS CLI を作成および管理します。これらのファイルは、CDK で使用できるセキュリティ認証情報を保存、管理、生成するために使用されますCLI。

をインストールするには AWS CLI、「 ユーザーガイド」の「 の最新バージョンのインストールまたは更新 AWS CLIAWS Command Line Interface 」を参照してください。

これらのファイルの詳細については、「 ユーザーガイド」の「設定と認証情報ファイルの設定AWS Command Line Interface 」を参照してください。

AWS CDK CLI のインストール

を使用して CDK Node Package ManagerをインストールしますCLI。ほとんどの場合、以下を実行して、最新バージョンをグローバルにインストールすることをお勧めします。

```
$ npm install -g aws-cdk
```

セキュリティ認証情報の設定方法

セキュリティ認証情報の設定方法は、ユーザーまたは組織がユーザーを管理する方法によって異なります。を使用して CDK のセキュリティ認証情報 AWS CLI を設定および管理することをお勧めしますCLI。これには、 などの AWS CLI コマンドを使用してローカルマシンでセキュリティ認証情報`aws configure`を設定することが含まれます。ただし、 `config`および `credentials` ファイルを手動で更新したり、環境変数を設定したり、別の方法を使用することもできます。

を使用してセキュリティ認証情報を設定する際のガイダンスと AWS CLI、さまざまな方法を使用する場合の設定と認証情報の優先順位については、AWS Command Line Interface 「 ユーザーガイド」の「認証とアクセス認証情報」を参照してください。CDK は、 と同じ設定と認証情報の優先順位CLIに従います AWS CLI。`--profile` コマンドラインオプションは、環境変数よりも優先されます。`AWS_PROFILE` と の両方の`CDK_DEFAULT_PROFILE`環境変数を設定している場合、`AWS_PROFILE`環境変数が優先されます。

セキュリティ認証情報などの基本設定をすばやく設定する場合は、「 ユーザーガイド」の「 のセットアップ AWS CLIAWS Command Line Interface 」を参照してください。

ローカルマシンでセキュリティ認証情報を設定したら、CDK CLI を使用して とやり取りできます AWS。ユーザーの管理方法で CDK を使用するCLI方法の詳細については、以下を参照してください。

- IAM Identity Center ユーザーの CDK CLI セキュリティ認証情報の設定と管理.

追加情報

にサインインするさまざまな方法については AWS、使用するユーザーのタイプに応じて、AWS「サインインユーザーガイド」の「サインインとは」を参照してください。 AWS

を含む および ツールを使用する際 AWS SDKsのリファレンス情報については、「 AWSSDKsおよび ツールリファレンスガイド AWS CLI」を参照してください。

IAM Identity Center ユーザーの CDK CLI セキュリティ認証情報の設定と管理

で作成および管理されているユーザーは、 AWS CDK コマンドラインインターフェイス () の短期認証情報を生成 AWS IAM Identity Center できますAWS CDK CLI。

前提条件を含む CDK のセキュリティ認証情報の概要についてはCLI、「」を参照してくださいのセキュリティ認証情報を設定する AWS CDKCLI。

トピック

- IAM Identity Center を使用する AWS CLI ように を設定する
- AWS CLI を使用して短期認証情報を取得する
- 短期認証情報を手動で設定する

- CDK で短期認証情報を使用するCLI
- 例: AWS CLI を使用して、IAM Identity Center ユーザーとして短期的なセキュリティ認証情報を設定する

IAM Identity Center を使用する AWS CLI ように を設定する

IAM Identity Center で認証 AWS CLI するように を設定できます。これは、IAM Identity Center ユーザーのセキュリティ認証情報を設定する際に推奨されるアプローチです。短期認証情報の生成に必要な IAM Identity Center 情報は、ローカルマシンの `config` ファイルに保存されます。このファイルを設定するには、 AWS CLI `aws configure sso` を使用します。詳細については、「 ユーザーガイド」の AWS 「IAM Identity Center AWS CLI を使用するように AWS Command Line Interface を設定する」を参照してください。

これらの手順を完了すると、`config`ファイルには少なくとも 1 つの名前付きプロファイル と が必要です`sso-session`。この情報を使用して、プロファイルの短期認証情報をリクエストおよび更新できます。

AWS CLI を使用して短期認証情報を取得する

コマンドを使用して AWS CLI `aws sso login`、プロファイルの短期認証情報を取得できます。`aws sso login` コマンドを使用してプロファイルを切り替えることもできます。短期認証情報の取得とプロファイルの切り替えの手順については、「 ユーザーガイド」の「IAM Identity Center の名前付きプロファイルを使用するAWS Command Line Interface 」を参照してください。

短期認証情報を手動で設定する

を使用する代わりに AWS CLI、 AWS アクセスポータルから短期認証情報を取得し、`credentials` ファイルと `config` ファイルを手動で更新してセキュリティ認証情報を設定できます。手順については、「 ユーザーガイド」の「短期認証情報による認証AWS Command Line Interface 」を参照してください。

CDK で短期認証情報を使用するCLI

CDK は、 で生成するか、 AWS CLI ファイル`credentials`と `config` ファイルで手動で設定する短期認証情報CLIを使用します。短期認証情報を設定したら、CDK CLI を使用して とやり取りできます AWS。

CDK を使用して で CDK スタックCLIを操作する場合は AWS、任意の CDK CLI コマンドで `--profile`オプションを使用して、短期認証情報の生成に使用した名前付きプロファイルを指定しま

す。`default` プロファイルを IAM Identity Center プロファイルとして設定している場合は、プロファイルを指定する必要はありません。

例: AWS CLI を使用して、IAM Identity Center ユーザーとして短期的なセキュリティ認証情報を設定する

この例では、 AWS IAM Identity Center トークンプロバイダー設定でユーザーを認証するように AWS Command Line Interface （ AWS CLI) を設定します。SSO トークンプロバイダー設定により、 は更新された認証トークン AWS CLI を自動的に取得して、 AWS Cloud Development Kit (AWS CDK) コマンドラインインターフェイス () で使用できる短期認証情報を生成できますAWS CDK CLI。

トピック

前提条件

この例では、以下の前提条件が満たされていることを前提としています。

- をセットアップ AWS して開始CLIツールをインストールするために必要な前提条件。詳細については、「前提条件」を参照してください。
- IAM Identity Center は、ユーザーの管理方法として組織によって設定されています。
- IAM Identity Center に少なくとも 1 人のユーザーが作成されています。

ステップ 1: を設定する AWS CLI

このステップの詳細な手順については、「 ユーザーガイド」の「自動認証更新で IAM Identity Center トークンプロバイダーの認証情報 AWS CLI を使用するように AWS Command Line Interface を設定する」を参照してください。

IAM Identity Center の情報を収集するために、組織が提供する AWS アクセスポータルにサインインします。これには、 SSO 開始 URL と SSO リージョン が含まれます。

次に、 コマンドを使用して AWS CLI `aws configure sso`、ローカルマシン`sso-session`で IAM Identity Center プロファイル と を設定します。

```
$ aws configure sso
SSO session name (Recommended): my-sso
SSO start URL [None]: https://my-sso-portal.awsapps.com/start
SSO region [None]: us-east-1
SSO registration scopes [sso:account:access]: <ENTER>
```

は、デフォルトのブラウザを開いて、IAM Identity Center アカウントのログインプロセスを開始 AWS CLI しようとします。 AWS CLI がブラウザを開くことができない場合は、ログインプロセスを手動で開始する手順が表示されます。このプロセスでは、IAM Identity Center セッションを現在の AWS CLI セッションに関連付けます。

セッションを確立すると、 に AWS アカウント 利用可能な AWS CLI が表示されます。

```
There are 2 AWS accounts available to you.
> DeveloperAccount, developer-account-admin@example.com (123456789011)
  ProductionAccount, production-account-admin@example.com (123456789022)
```

矢印キーを使用して を選択しますDeveloperAccount。

次に、 には、選択したアカウントから使用可能な IAM ロール AWS CLI が表示されます。

```
Using the account ID 123456789011
There are 2 roles available to you.
> ReadOnly
  FullAccess
```

矢印キーを使用して を選択しますFullAccess。

次に、 は、プロファイルのデフォルトの出力形式、デフォルトの 、および名前を指定して AWS リージョン、設定を完了するように AWS CLI 促します。

```
CLI default client Region [None]: us-west-2 <ENTER>>
CLI default output format [None]: json <ENTER>
CLI profile name [123456789011_FullAccess]: my-dev-profile <ENTER>
```

は、 で名前付きプロファイルを使用する方法を示す最終メッセージ AWS CLI を表示します AWS CLI。

```
To use this profile, specify the profile name using --profile, as shown:

aws s3 ls --profile my-dev-profile
```

このステップを完了すると、configファイルの形式は次のようになります。

```
[profile my-dev-profile]
sso_session = my-sso
sso_account_id = 123456789011
sso_role_name = fullAccess
region = us-west-2
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://my-sso-portal.awsapps.com/start
sso_registration_scopes = sso:account:access
```

これで、この sso-sessionと名前付きプロファイルを使用してセキュリティ認証情報をリクエストできるようになりました。

ステップ 2: AWS CLI を使用してセキュリティ認証情報を生成する

このステップの詳細な手順については、「 ユーザーガイド」の「IAM Identity Center の名前付きプロファイルを使用するAWS Command Line Interface 」を参照してください。

コマンドを使用して、プロファイルのセキュリティ認証情報を AWS CLI aws sso loginリクエストします。

```
$ aws sso login --profile my-dev-profile
```

はデフォルトのブラウザを開き、IAM ログインを検証 AWS CLI します。現在 IAM Identity Center にサインインしていない場合は、サインインプロセスを完了するように求められます。 AWS CLI がブラウザを開くことができない場合は、認証プロセスを手動で開始する手順が表示されます。

ログインに成功すると、 は IAM Identity Center セッション認証情報を AWS CLI キャッシュします。これらの認証情報には有効期限のタイムスタンプが含まれます。有効期限が切れると、 AWS CLI は IAM Identity Center に再度サインインするようリクエストします。

有効な IAM Identity Center 認証情報を使用して、 はプロファイルで指定された IAM ロールの AWS 認証情報 AWS CLI を安全に取得します。ここから、認証情報で AWS CDK CLIを使用できます。

ステップ 3: CDK を使用するCLI

CDK CLI コマンドでは、 `--profile`オプションを使用して、認証情報を生成した名前付きプロファイルを指定します。認証情報が有効な場合、CDK CLI はコマンドを正常に実行します。以下に例を示します。

```
$ cdk diff --profile my-dev-profile
Stack CdkAppStack
Hold on while we create a read-only change set to get a diff with accurate replacement
 information (use --no-change-set to use a less accurate but faster template-only diff)
Resources
[-] AWS::S3::Bucket myBucket myBucket5AF9C99B destroy

Outputs
[-] Output BucketRegion: {"Value":{"Ref":"AWS::Region"}}


#  Number of stacks with differences: 1
```

認証情報の有効期限が切れると、次のようなエラーメッセージが表示されます。

```
$ cdk diff --profile my-dev-profile
Stack CdkAppStack

Unable to resolve AWS account to use. It must be either configured when you define your
 CDK Stack, or through the environment
```

認証情報を更新するには、 コマンドを使用します AWS CLI `aws sso login`。

```
$ aws sso login --profile my-dev-profile
```

で使用する環境を設定する AWS CDK

AWS 環境は、 で使用する複数の方法で設定できます AWS Cloud Development Kit (AWS CDK)。環境を管理する AWS 最適な方法は、特定の二ーズによって異なります。

アプリケーション内の各 CDK スタックは、スタックのデプロイ先を決定するために、最終的に環境に関連付ける必要があります。

AWS 環境の概要については、「」を参照してください環境。

トピック

- から環境を指定できる場所
- 環境の優先順位 AWS CDK
- 環境を指定するタイミング
- で環境を指定する方法 AWS CDK
- で環境を設定する際の考慮事項 AWS CDK
- 例

から環境を指定できる場所

認証情報と設定ファイルで環境を指定するか、 `Stack` コンストラクトライブラリの コンストラクトの AWS envプロパティを使用して環境を指定できます。

認証情報と設定ファイル

AWS Command Line Interface （ AWS CLI) を使用して、 AWS 環境情報を保存、整理、管理する `credentials`および `config` ファイルを作成できます。これらのファイルの詳細については、「 ユーザーガイド」の「設定ファイルと認証情報ファイルの設定AWS Command Line Interface 」を参照してください。

これらのファイルに保存されている値は、プロファイル によって整理されます。プロファイルとこれらのファイル内のキーと値のペアに名前を付ける方法は、プログラムによるアクセスを設定する方法によって異なります。さまざまな方法の詳細については、「」を参照してくださいのセキュリティ認証情報を設定する AWS CDKCLI。

一般に、 は `credentials` ファイルからの AWS アカウント 情報と `config` ファイルからの AWS リージョン 情報を AWS CDK 解決します。

credentials および config ファイルを設定したら、環境変数を使用して および AWS CDK CLI で使用する環境を指定できます。

スタックコンストラクトの env プロパティ

Stack コンストラクトの envプロパティを使用して、各スタックの環境を指定できます。このプロパティは、使用するアカウントとリージョンを定義します。ハードコードされた値をこのプロパティに渡すことも、CDK によって提供される環境変数を渡すこともできます。

環境変数を渡すには、 AWS_DEFAULT_ACCOUNTおよび AWS_DEFAULT_REGION環境変数を使用します。これらの環境変数は、 credentialsおよび config ファイルから値を渡すことができます。CDK コード内のロジックを使用して、これらの環境変数の値を決定することもできます。

環境の優先順位 AWS CDK

環境を指定する複数の方法を使用する場合、 は次の優先順位 AWS CDK に従います。

1. Stack コンストラクトの envプロパティで指定されたハードコードされた値。
2. AWS_DEFAULT_ACCOUNT および Stack コンストラクトの envプロパティで指定されたAWS_DEFAULT_REGION環境変数。
3. credentials および config ファイルからプロファイルに関連付けられ、 --profileオプションCLIを使用して CDK に渡される環境情報。
4. credentials および config ファイルのdefaultプロファイル。

環境を指定するタイミング

CDK を使用して開発する場合、まず CDK スタックを定義します。このスタックには、 AWS リソースを表すコンストラクトが含まれます。次に、各 CDK スタックを AWS CloudFormation テンプレートに合成します。次に、CloudFormation テンプレートを環境にデプロイします。環境を指定すると、環境情報が適用されるタイミングが決まり、CDK の動作と結果に影響する可能性があります。

テンプレート合成時に環境を指定する

Stack コンストラクトの envプロパティを使用して環境情報を指定すると、環境情報はテンプレート合成に適用されます。cdk synth または を実行すると、環境固有の CloudFormation テンプレートcdk deployが生成されます。

env プロパティ内で環境変数を使用する場合は、CDK CLI コマンドで --profileオプションを使用して、認証情報と設定ファイルから環境情報を含むプロファイルを渡す必要があります。その後、この情報はテンプレート合成に適用され、環境固有のテンプレートが作成されます。

CloudFormation テンプレート内の環境情報は、他の方法よりも優先されます。例えば、 で別の環境を指定するとcdk deploy --profile *profile*、プロファイルは無視されます。

この方法で環境情報を提供すると、CDK アプリ内で環境に依存するコードとロジックを使用できます。つまり、合成されたテンプレートは、合成されたマシン、ユーザー、またはセッションによって異なる可能性があります。このアプローチは、多くの場合、開発中に許容または望ましいですが、本番環境での使用には推奨されません。

スタックデプロイ時に環境を指定する

Stack コンストラクトの envプロパティを使用して環境を指定しない場合、CDK CLIは合成時に環境に依存しない CloudFormation テンプレートを生成します。その後、 を使用してデプロイ先の環境を指定できますcdk deploy --profile *profile*。

環境に依存しないテンプレートをデプロイするときにプロファイルを指定しない場合、CDK CLIはデプロイ時に credentialsおよび config ファイルのdefaultプロファイルから環境値を使用しようとします。

デプロイ時に環境情報が利用できない場合、 は、、stack.region、 などの環境関連の属性を使用してstack.account、デプロイ時に環境情報の解決を試み AWS CloudFormation ますstack.availabilityZones。

環境に依存しないスタックの場合、スタック内のコンストラクトは環境情報を使用できず、環境情報を必要とするロジックを使用できません。例えば、 のようなコードを記述if (stack.region === 'us-east-1')したり、 などの環境情報を必要とするコンストラクトメソッドを使用することはできませんVpc.fromLookup。これらの機能を使用するには、 envプロパティで環境を指定する必要があります。

環境に依存しないスタックの場合、アベイラビリティーゾーンを使用するコンストラクトには 2 つのアベイラビリティーゾーンが表示され、スタックを任意のリージョンにデプロイできます。

で環境を指定する方法 AWS CDK

スタックごとにハードコードされた環境を指定する

Stack コンストラクトの envプロパティを使用して、スタックの AWS 環境値を指定します。以下に例を示します。

TypeScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

JavaScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

Python

```
env_EU = cdk.Environment(account="8373873873", region="eu-west-1")
env_USA = cdk.Environment(account="2383838383", region="us-west-2")

MyFirstStack(app, "first-stack-us", env=env_USA)
MyFirstStack(app, "first-stack-eu", env=env_EU)
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        return Environment.builder()
                .account(account)
                .region(region)
                .build();
```

```
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv("8373873873", "eu-west-1");
        Environment envUSA = makeEnv("2383838383", "us-west-2");

        new MyFirstStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyFirstStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment
    {
        Account = account,
        Region = region
    };
}

var envEU = makeEnv(account: "8373873873", region: "eu-west-1");
var envUSA = makeEnv(account: "2383838383", region: "us-west-2");

new MyFirstStack(app, "first-stack-us", new StackProps { Env=envUSA });
new MyFirstStack(app, "first-stack-eu", new StackProps { Env=envEU });
```

Go

```
env_EU := awscdk.Environment{
 Account: jsii.String("8373873873"),
 Region:  jsii.String("eu-west-1"),
}

env_USA := awscdk.Environment{
 Account: jsii.String("2383838383"),
```

```
 Region:  jsii.String("us-west-2"),
}

MyFirstStack(app, "first-stack-us", &awscdk.StackProps{
 Env: &env_USA,
})

MyFirstStack(app, "first-stack-eu", &awscdk.StackProps{
 Env: &env_EU,
})
```

このアプローチは、本番環境で推奨されます。この方法で環境を明示的に指定することで、スタックが常に特定の環境にデプロイされるようにできます。

環境変数を使用して環境を指定する

AWS CDK には、CDK コードで使用できる CDK_DEFAULT_ACCOUNTと の 2 つの環境変数が用意されていますCDK_DEFAULT_REGION。スタックインスタンスの envプロパティ内でこれらの環境変数を使用すると、CDK CLI--profileオプションを使用して認証情報と設定ファイルから環境情報を渡すことができます。

これらの環境変数を指定する方法の例を次に示します。

TypeScript

Node の process オブジェクトを介して環境変数にアクセスします。

> **Note**
>
> process で使用するDefinitelyTypedモジュールが必要です TypeScript。 は、このモジュールcdk initをインストールします。ただし、追加前に作成されたプロジェクトを使用している場合、または を使用してプロジェクトをセットアップしていない場合は、このモジュールを手動でインストールする必要がありますcdk init。
>
> ```
> npm install @types/node
> ```

```
new MyDevStack(app, 'dev', {
  env: {
```

```
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

Node の process オブジェクトを介して環境変数にアクセスします。

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

Python

os モジュールのenvironディクショナリを使用して環境変数にアクセスします。

```
import os
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ["CDK_DEFAULT_ACCOUNT"],
    region=os.environ["CDK_DEFAULT_REGION"]))
```

Java

を使用してSystem.getenv()環境変数の値を取得します。

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
```

```
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

を使用してSystem.Environment.GetEnvironmentVariable()環境変数の値を取得します。

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

Go

```
import "os"

MyDevStack(app, "dev", &awscdk.StackProps{
 Env: &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
    },
})
```

環境変数を使用して環境を指定することで、同じ CDK スタックを異なる環境の AWS CloudFormation テンプレートに合成できます。つまり、CDK コードを変更しなくても、同じ CDK スタックを異なる AWS 環境にデプロイできます。を実行するときに使用するプロファイルを指定するだけで済みますcdk synth。

このアプローチは、同じスタックを異なる環境にデプロイする場合の開発環境に最適です。ただし、同じ CDK コードが合成されるマシン、ユーザー、またはセッションに応じて異なるテンプレートを合成できるため、本番環境ではこのアプローチはお勧めしません。

CDK を使用して認証情報と設定ファイルから環境を指定するCLI

環境に依存しないテンプレートをデプロイする場合は、任意の CDK CLI コマンドで --profileオプションを使用して、使用するプロファイルを指定します。以下は、 ファイルcredentialsと config ファイルで定義されている prodプロファイルmyStackを使用して、 という名前の CDK スタックをデプロイする例です。

```
$ cdk deploy myStack --profile prod
```

--profile オプションと他の CDK CLI コマンドおよびオプションの詳細については、「」を参照してくださいAWS CDK CLI コマンドリファレンス。

で環境を設定する際の考慮事項 AWS CDK

スタック内でコンストラクトを使用して定義するサービスは、デプロイ先のリージョンをサポートしている必要があります。リージョン AWS のサービス ごとにサポートされている のリストについては、「リージョンAWS 別のサービス」を参照してください。

指定した環境に を使用してスタックデプロイを実行するには、有効な AWS Identity and Access Management (IAM) AWS CDK 認証情報が必要です。

例

CDK スタックから環境に依存しない CloudFormation テンプレートを合成する

この例では、CDK スタックから環境に依存しない CloudFormation テンプレートを作成します。その後、このテンプレートを任意の環境にデプロイできます。

CDK スタックの例を次に示します。このスタックは、Amazon S3 バケットとバケットのリージョンの CloudFormation スタック出力を定義します。この例では、 envは定義されていません。

TypeScript

```
export class CdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the S3 bucket
    const bucket = new s3.Bucket(this, 'myBucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });

    // Create an output for the bucket's Region
    new cdk.CfnOutput(this, 'BucketRegion', {
      value: bucket.env.region,
    });
  }
}
```

JavaScript

```
class CdkAppStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Create the S3 bucket
    const bucket = new s3.Bucket(this, 'myBucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });

    // Create an output for the bucket's Region
    new cdk.CfnOutput(this, 'BucketRegion', {
      value: bucket.env.region,
    });
  }
}
```

Python

```
class CdkAppStack(cdk.Stack):
```

```
    def __init__(self, scope: cdk.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Create the S3 bucket
        bucket = s3.Bucket(self, 'myBucket',
            removal_policy=cdk.RemovalPolicy.DESTROY
        )

        # Create an output for the bucket's Region
        cdk.CfnOutput(self, 'BucketRegion',
            value=bucket.env.region
        )
```

Java

```
public class CdkAppStack extends Stack {

    public CdkAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Create the S3 bucket
        Bucket bucket = Bucket.Builder.create(this, "myBucket")
            .removalPolicy(RemovalPolicy.DESTROY)
            .build();

        // Create an output for the bucket's Region
        CfnOutput.Builder.create(this, "BucketRegion")
            .value(this.getRegion())
            .build();
    }
}
```

C#

```
namespace MyCdkApp
{
    public class CdkAppStack : Stack
    {
        public CdkAppStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            // Create the S3 bucket
```

```
            var bucket = new Bucket(this, "myBucket", new BucketProps
            {
                RemovalPolicy = RemovalPolicy.DESTROY
            });

            // Create an output for the bucket's Region
            new CfnOutput(this, "BucketRegion", new CfnOutputProps
            {
                Value = this.Region
            });
        }
    }
}
```

Go

```
func NewCdkAppStack(scope constructs.Construct, id string, props *CdkAppStackProps)
 awscdk.Stack {
 stack := awscdk.NewStack(scope, &id, &props.StackProps)

 // Create the S3 bucket
 bucket := awss3.NewBucket(stack, jsii.String("myBucket"), &awss3.BucketProps{
  RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
 })

 // Create an output for the bucket's Region
 awscdk.NewCfnOutput(stack, jsii.String("BucketRegion"), &awscdk.CfnOutputProps{
  Value: stack.Region(),
 })

 return stack
}
```

を実行するとcdk synth、CDK はバケットのリージョンの出力値AWS::Regionとして 擬似パラメータを含む CloudFormation テンプレートCLIを生成します。このパラメータはデプロイ時に解決されます。

```
Outputs:
  BucketRegion:
    Value:
      Ref: AWS::Region
```

認証情報と設定ファイルのdevプロファイルで指定された環境にこのスタックをデプロイするには、以下を実行します。

```
$ cdk deploy CdkAppStack --profile dev
```

プロファイルを指定しない場合、CDK CLIは認証情報と設定ファイルのdefaultプロファイルからの環境情報の使用を試みます。

ロジックを使用してテンプレート合成時の環境情報を決定する

この例では、有効な式を使用するようにstackインスタンスの envプロパティを設定します。と の 2 CDK_DEPLOY_ACCOUNT つの追加の環境変数を指定しますCDK_DEPLOY_REGION。これらの環境変数が存在する場合、合成時のデフォルトを上書きできます。

TypeScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

Python

```
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ.get("CDK_DEPLOY_ACCOUNT", os.environ["CDK_DEFAULT_ACCOUNT"]),
    region=os.environ.get("CDK_DEPLOY_REGION", os.environ["CDK_DEFAULT_REGION"])
```

Java

```
public class MyApp {
```

```
    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEPLOY_ACCOUNT") : account;
        region = (region == null) ? System.getenv("CDK_DEPLOY_REGION") : region;
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_ACCOUNT") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_REGION") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}
```

```
new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

Go

```
var account, region string
var b bool

if account, b = os.LookupEnv("CDK_DEPLOY_ACCOUNT"); !b || len(account) == 0 {
 account = os.Getenv("CDK_DEFAULT_ACCOUNT")
}
if region, b = os.LookupEnv("CDK_DEPLOY_REGION"); !b || len(region) == 0 {
 region = os.Getenv("CDK_DEFAULT_REGION")
}

MyDevStack(app, "dev", &awscdk.StackProps{
 Env: &awscdk.Environment{
  Account: &account,
  Region:  &region,
 },
})
```

スタックの環境をこのように宣言すると、短いスクリプトまたはバッチファイルを記述し、コマンドライン引数から変数を設定して、 を呼び出すことができますcdk deploy。次に例を示します。最初の 2 つを超える引数は、コマンドラインオプションまたは引数を指定cdk deployするために に渡されます。

macOS/Linux

```
#!/usr/bin/env bash
if [[ $# -ge 2 ]]; then
    export CDK_DEPLOY_ACCOUNT=$1
    export CDK_DEPLOY_REGION=$2
    shift; shift
    npx cdk deploy "$@"
    exit $?
else
    echo 1>&2 "Provide account and region as first two args."
    echo 1>&2 "Additional args are passed through to cdk deploy."
    exit 1
fi
```

スクリプトを として保存しcdk-deploy-to.sh、 chmod +x cdk-deploy-to.shを実行して実行可能にします。

Windows

```
@findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass
 %~dpn0.ps1 %*
@exit /B %ERRORLEVEL%
if ($args.length -ge 2) {
    $env:CDK_DEPLOY_ACCOUNT, $args = $args
    $env:CDK_DEPLOY_REGION,  $args = $args
    npx cdk deploy $args
    exit $lastExitCode
} else {
    [console]::error.writeline("Provide account and region as first two args.")
    [console]::error.writeline("Additional args are passed through to cdk deploy.")
    exit 1
}
```

スクリプトの Windows バージョンは、macOS /Linux バージョンと同じ機能 PowerShell を提供するために を使用します。 macOS また、コマンドラインから簡単に呼び出せるように、バッチファイルとして実行できるようにする手順も含まれています。として保存する必要がありますcdk-deploy-to.bat。ファイルはcdk-deploy-to.ps1、バッチファイルが呼び出されたときに作成されます。

その後、スクリプトを使用して特定の環境にデプロイする追加のcdk-deploy-toスクリプトを作成できます。以下に例を示します。

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-test.sh
./cdk-deploy-to.sh 123457689 us-east-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-test.bat
cdk-deploy-to 135792469 us-east-1 %*
```

スクリプトを使用して複数の環境にcdk-deploy-toデプロイする例を次に示します。最初のデプロイが失敗すると、プロセスは停止します。

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-prod.sh
./cdk-deploy-to.sh 135792468 us-west-1 "$@" || exit
./cdk-deploy-to.sh 246813579 eu-west-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-prod.bat
cdk-deploy-to 135792469 us-west-1 %* || exit /B
cdk-deploy-to 245813579 eu-west-1 %*
```

で使用する環境をブートストラップする AWS CDK

AWS 環境をブートストラップして、 AWS Cloud Development Kit (AWS CDK) スタックデプロイの準備をします。

- 環境の概要については、「」を参照してください環境。
- ブートストラップの概要については、「」を参照してくださいブートストラッピング。

トピック

環境をブートストラップする方法

AWS CDK コマンドラインインターフェイス (AWS CDK CLI) または任意の AWS CloudFormation デプロイツールを使用して、環境をブートストラップできます。

CDK を使用するCLI

CDK CLI `cdk bootstrap` コマンドを使用して環境をブートストラップできます。これは、ブートストラップに大幅な変更が必要ない場合は推奨される方法です。

任意の作業ディレクトリからのブートストラップ

作業ディレクトリからブートストラップするには、ブートストラップする環境をコマンドライン引数として指定します。以下に例を示します。

```
$ cdk bootstrap aws://123456789012/us-east-1
```

引数を指定する場合、aws://プレフィックスはオプションです。以下は有効です。

```
$ cdk bootstrap 123456789012/us-east-1
```

複数の環境を同時にブートストラップするには、複数の引数を指定します。

```
$ cdk bootstrap aws://123456789012/us-east-1 aws://123456789012/us-east-2
```

CDK プロジェクトの親ディレクトリからのブートストラップ

cdk.json ファイルを含む CDK プロジェクトの親ディレクトリcdk bootstrapから を実行できます。引数として環境を指定しない場合、CDK CLIは configおよび credentials ファイルなどのデフォルトのソースから環境情報、または CDK スタックに指定された環境情報を取得します。

CDK プロジェクトの親ディレクトリからブートストラップする場合、コマンドライン引数から提供される環境が他のソースよりも優先されます。

config および credentials ファイルで指定されている環境をブートストラップするには、--profileオプションを使用します。

```
$ cdk bootstrap --profile prod
```

cdk bootstrap コマンドとサポートされているオプションの詳細については、「」を参照してくださいcdk bootstrap。

任意の AWS CloudFormation ツールを使用する

ブートストラップテンプレートは、aws-cdk GitHubリポジトリからコピーすることも、 cdk bootstrap --show-template コマンドを使用してテンプレートを取得することもできます。次に、任意の AWS CloudFormation ツールを使用してテンプレートを環境にデプロイします。

この方法では、 AWS CloudFormation StackSets または を使用できます AWS Control Tower。 AWS CloudFormation コンソールまたは AWS Command Line Interface () を使用することもできますAWS CLI。テンプレートは、デプロイする前に変更できます。この方法は、より柔軟で大規模なデプロイに適しています。

--show-template オプションを使用してブートストラップテンプレートを取得してローカルマシンに保存する例を次に示します。

macOS/Linux

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

Windows

Windows PowerShell では、テンプレートのエンコードを保持するために を使用する必要があります。

```
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-
template.yaml"
```

CDK を使用してこのテンプレートをデプロイするにはCLI、以下を実行します。

```
$ cdk bootstrap --template bootstrap-template.yaml
```

を使用してテンプレート AWS CLI をデプロイする例を次に示します。

macOS/Linux

```
aws cloudformation create-stack \
  --stack-name CDKToolkit \
  --template-body file://path/to/bootstrap-template.yaml \
  --capabilities CAPABILITY_NAMED_IAM \
  --region us-west-1
```

Windows

```
aws cloudformation create-stack ^
  --stack-name CDKToolkit ^
  --template-body file://path/to/bootstrap-template.yaml ^
  --capabilities CAPABILITY_NAMED_IAM ^
  --region us-west-1
```

を使用して複数の環境 CloudFormation StackSets をブートストラップする方法については、AWS クラウドオペレーションと移行ブログのAWS CDK 「 AWS アカウント を使用するための複数のブートストラップ CloudFormation StackSets」を参照してください。

環境をブートストラップするタイミング

環境にデプロイする前に、各環境をブートストラップする必要があります。ブートストラップされていない環境に CDK スタックをデプロイしようとすると、次のようなエラーが表示されます。

```
$ cdk deploy

#  Synthesis time: 2.02s

 # Deployment failed: Error: BootstrapExampleStack: SSM parameter /cdk-bootstrap/
hnb659fds/version not found. Has the environment been bootstrapped? Please run 'cdk
 bootstrap' (see https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
```

環境を複数回ブートストラップしても問題ありません。環境がすでにブートストラップされている場合、必要に応じてブートストラップスタックがアップグレードされます。そうしないと、何も起こりません。

ブートストラップスタックを更新する

CDK チームは定期的にブートストラップテンプレートを新しいバージョンに更新します。この場合、ブートストラップスタックを更新することをお勧めします。ブートストラッププロセスをカスタマイズしていない場合は、最初に環境をブートストラップするために実行したのと同じ手順に従って、ブートストラップスタックを更新できます。詳細については、「ブートストラップテンプレートのバージョン履歴」を参照してください。

ブートストラップをカスタマイズする

デフォルトのブートストラップテンプレートがニーズに合わない場合は、次の方法で環境へのリソースのブートストラップをカスタマイズできます。

- `cdk bootstrap` コマンドでコマンドラインオプションを使用する – この方法は、コマンドラインオプションでサポートされる小さな特定の変更を行うのに最適です。
- デフォルトのブートストラップテンプレートを変更してデプロイする – この方法は、複雑な変更を行う場合や、ブートストラップ中にプロビジョニングされたリソースの設定を完全に制御したい場合に最適です。

ブートストラップのカスタマイズの詳細については、「」を参照してくださいAWS CDK ブートストラップをカスタマイズする。

CDK パイプラインによるブートストラップ

CDK Pipelines を使用して別のアカウントの環境にデプロイしている場合、次のようなメッセージが表示されます。

```
Policy contains a statement with one or more invalid principals
```

このエラーメッセージは、適切な IAM ロールが他の環境に存在しないことを意味します。最も可能性の高い原因は、環境がブートストラップされていないことです。環境をブートストラップして、もう一度試してください。

ブートストラップスタックの削除からの保護

ブートストラップスタックが削除されると、CDK デプロイをサポートするために環境に最初にプロビジョニングされた AWS リソースも削除されます。これにより、パイプラインは動作しなくなります。この場合、復旧のための一般的な解決策はありません。

環境がブートストラップされたら、環境のブートストラップスタックを削除して再作成しないでください。代わりに、 `cdk bootstrap` コマンドを再度実行して、ブートストラップスタックを新しいバージョンに更新してみてください。

ブートストラップスタックが誤って削除されないように、終了保護を有効にするには、 `cdk bootstrap` コマンドで `--termination-protection`オプションを指定することをお勧めします。新規または既存のブートストラップスタックで終了保護を有効にできます。終了保護を有効にする手順については、「ブートストラップスタック の終了保護を有効にする」を参照してください。

ブートストラップテンプレートのバージョン履歴

ブートストラップテンプレートはバージョニングされ、それ AWS CDK 自体とともに時間の経過とともに進化します。独自のブートストラップテンプレートを指定する場合は、正規のデフォルトテンプレートで最新の状態を維持します。テンプレートが引き続きすべての CDK 機能で動作するようにしたい。

> **Note**
>
> 以前のバージョンのブートストラップテンプレートでは、デフォルトでブートストラップされた AWS KMS key 各環境に が作成されていました。KMS キーの料金が発生しないように

するには、 を使用してこれらの環境を再起動します`--no-bootstrap-customer-key`。現在のデフォルトは KMS キーではないため、これらの料金を回避できます。

このセクションには、各バージョンで行われた変更のリストが含まれています。

[テンプレートのバージョン]	AWS CDK バージョン	変更
1	1.40.0	バケット、キー、リポジトリ、ロールを含むテンプレートの初期バージョン。
2	1.45.0	アセット発行ロールを個別のファイル発行ロールとイメージ発行ロールに分割します。
3	1.46.0	アセットコンシューマーに復号化アクセス許可を追加できるように`FileAssetKeyArn` エクスポートを追加します。
4	1.61.0	AWS KMS アクセス許可が Amazon S3 経由で暗黙的に使用され、 が不要になりました`FileAsetKeyArn` 。SSM `CdkBootstrapVersion` パラメータを追加して、スタック名を知らずにブートストラップスタックバージョンを検証できるようにします。
5	1.87.0	デプロイロールは SSM パラメータを読み取ることができます。

[テンプレートのバージョン]	AWS CDK バージョン	変更
6	1.108.0	デプロイロールとは別にルックアップロールを追加します。
6	1.109.0	デプロイ、ファイル発行、およびイメージ発行ロールに`aws-cdk:bootstrap-role` タグをアタッチします。
7	1.110.0	デプロイロールは、ターゲットアカウントのバケットを直接読み取ることができなくなります。(ただし、このロールは実質的に管理者であり、バケットを読み取り可能にするために常にその AWS CloudFormation アクセス許可を使用できます)。
8	1.114.0	ルックアップロールには、ターゲット環境への完全な読み取り専用アクセス許可があり、`aws-cdk:bootstrap-role` タグもあります。
9	2.1.0	Amazon S3 アセットのアップロードが、一般的に参照される暗号化 SCP によって拒否されないように修正しました。
10	2.4.0	Amazon ECR ScanOnPush がデフォルトで有効になりました。

[テンプレートのバージョン]	AWS CDK バージョン	変更
11	2.18.0	Lambda が Amazon ECR リポジトリからプルして、再起動後も存続できるようにするポリシーを追加しました。
12	2.20.0	実験的な のサポートを追加しましたcdk import。
13	2.25.0	ブートストラップで作成された Amazon ECR リポジトリのコンテナイメージをイミュータブルにします。
14	2.34.0	デフォルトでリポジトリレベルで Amazon ECR イメージスキャンをオフにして、イメージスキャンをサポートしていないブートストラップリージョンを許可します。
15	2.60.0	KMS キーにタグを付けることはできません。
16	2.69.0	Security Hub の検出結果 KMS.2 に対処します。
17	2.72.0	Security Hub の検出結果 ECR.3 に対処します。
18	2.80.0	バージョン 16 に対して行われた変更は、すべてのパーティションで機能しないため、推奨されません。

[テンプレートのバージョン]	AWS CDK バージョン	変更
19	2.106.1	AccessControl プロパティがテンプレートから削除されたバージョン 18 に加えられた変更を元に戻しました。(#27964)
20	2.119.0	AWS CloudFormation デプロイ IAM ロールに `ssm:GetParameters` アクションを追加します。詳細については、「#28336」を参照してください。

レガシーから最新のブートストラップテンプレートへのアップグレード

AWS CDK v1 は、レガシーとモダンの 2 つのブートストラップテンプレートをサポートしました。CDK v2 は最新の テンプレートのみをサポートしています。参考までに、これら 2 つのテンプレートの大まかな違いを次に示します。

機能	レガシー (v1 のみ)	Modern (v1 および v2)
クロスアカウントデプロイ	許可されていません	許可
AWS CloudFormation アクセス許可	現在のユーザーのアクセス許可 (AWS プロファイル、環境変数などによって決定される) を使用してデプロイします。	ブートストラップスタックがプロビジョニングされたときに指定されたアクセス許可を使用してデプロイします (例えば、 を使用`--trust`)
バージョニング	使用可能なブートストラップスタックのバージョンは 1 つだけです	ブートストラップスタックはバージョニングされています。将来のバージョンでは新しいリソースを追加でき、

機能	レガシー (v1 のみ)	Modern (v1 および v2)
		AWS CDK アプリケーションには最小バージョンが必要になる場合があります。
リソース *	Amazon S3 バケット	Amazon S3 バケット
		AWS KMS key
		IAM ロール
		Amazon ECR リポジトリ
		バージョニング用の SSM パラメータ
リソースの命名	自動生成	確定的
バケット暗号化	デフォルトキー	AWS デフォルトでは マネージドキー。カスタマーマネージドキーを使用するようにカスタマイズできます。

* 必要に応じて、ブートストラップテンプレートにリソースを追加します。

レガシーテンプレートを使用してブートストラップされた環境は、再起動によって CDK v2 の最新のテンプレートを使用するようにアップグレードする必要があります。レガシーバケットを削除する前に、環境内のすべての AWS CDK アプリケーションを少なくとも 1 回再デプロイします。

Security Hub の検出結果に対処する

を使用している場合は AWS Security Hub、ブートストラッププロセスによって AWS CDK 作成された一部のリソースで結果が報告される場合があります。Security Hub の検出結果は、精度と安全性を再確認する必要があるリソース設定を見つけるのに役立ちます。これらの特定のリソース設定は AWS Security で確認済みであり、セキュリティ上の問題にはならないと確信しています。

[KMS.2] IAM プリンシパルは、すべての KMS キーで復号アクションを許可する IAM インラインポリシーを使用しないでください

デプロイロール (DeploymentActionRole) は、暗号化されたデータを読み取るアクセス許可を付与します。これは、CDK Pipelines によるクロスアカウントデプロイに必要です。このロールのポリシーは、すべてのデータにアクセス許可を付与するわけではありません。Amazon S3 および から暗号化されたデータを読み取るアクセス許可を付与するのは AWS KMS、それらのリソースがバケットまたはキーポリシーを通じて許可する場合のみです。

以下は、ブートストラップテンプレートからのデプロイロール内のこれら 2 つのステートメントのスニペットです。

```
DeploymentActionRole:
    Type: AWS::IAM::Role
    Properties:
      ...
      Policies:
        - PolicyDocument:
            Statement:
              ...
              - Sid: PipelineCrossAccountArtifactsBucket
                Effect: Allow
                Action:
                  - s3:GetObject*
                  - s3:GetBucket*
                  - s3:List*
                  - s3:Abort*
                  - s3:DeleteObject*
                  - s3:PutObject*
                Resource: "*"
                Condition:
                  StringNotEquals:
                    s3:ResourceAccount:
                      Ref: AWS::AccountId
              - Sid: PipelineCrossAccountArtifactsKey
                Effect: Allow
                Action:
                  - kms:Decrypt
                  - kms:DescribeKey
                  - kms:Encrypt
                  - kms:ReEncrypt*
                  - kms:GenerateDataKey*
```

```
                Resource: "*"
                Condition:
                  StringEquals:
                    kms:ViaService:
                      Fn::Sub: s3.${AWS::Region}.amazonaws.com
              ...
```

Security Hub がこれにフラグを付けるのはなぜですか？

ポリシーには、 `Condition`句と`Resource: *`組み合わせた が含まれます。Security Hub はワイルドカードにフラグ*を付けます。このワイルドカードは、アカウントがブートストラップされた時点で、アー CodePipeline ティファクトバケットの CDK Pipelines によって作成された AWS KMS キーがまだ存在しないため、ARN によってブートストラップテンプレートで参照できないために使用されます。さらに、Security Hub は、このフラグを立てるときに `Condition`句を考慮しません。これは、 AWS KMS キー AWS アカウント と同じ からの`Resource: *`リクエスト`Condition`に制限されます。これらのリクエストは、 AWS KMS キー AWS リージョン と同じ の Amazon S3 から送信する必要があります。

この検出結果を修正する必要がありますか？

ブートストラップテンプレートの AWS KMS キーを過度に許容するように変更していない限り、デプロイロールは必要以上のアクセスを許可しません。したがって、この検出結果を修正する必要はありません。

この検出結果を修正するにはどうすればよいですか？

この検出結果の修正方法は、クロスアカウントデプロイに CDK Pipelines を使用するかどうかによって異なります。

Security Hub の検出結果を修正し、クロスアカウントデプロイに CDK Pipelines を使用するには

1. まだデプロイしていない場合は、 `cdk bootstrap` コマンドを使用して CDK ブートストラップスタックをデプロイします。
2. まだ作成していない場合は、CDK を作成してデプロイしますPipeline。手順については、「CDK Pipelines を使用した継続的インテグレーションとデリバリー (CI/CD)」を参照してください。
3. CodePipeline アーティファクトバケットの AWS KMS キー ARN を取得します。このリソースはパイプラインの作成時に作成されます。

4. CDK ブートストラップテンプレートのコピーを取得して変更します。以下は、 を使用した例です AWS CDK CLI。

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

5. PipelineCrossAccountArtifactsKey ステートメントを ARN 値に置き換えてResource: *、テンプレートを変更します。
6. テンプレートをデプロイしてブートストラップスタックを更新します。CDK を使用した例を次に示しますCLI。

```
$ cdk bootstrap aws://account-id/region --template bootstrap-template.yaml
```

クロスアカウントデプロイに CDK Pipelines を使用していない場合に Security Hub の検出結果を修正するには

1. CDK ブートストラップテンプレートのコピーを取得して変更します。CDK を使用した例を次に示しますCLI。

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

2. テンプレートから PipelineCrossAccountArtifactsBucketおよび PipelineCrossAccountArtifactsKeyステートメントを削除します。
3. テンプレートをデプロイしてブートストラップスタックを更新します。CDK を使用した例を次に示しますCLI。

```
$ cdk bootstrap aws://account-id/region --template bootstrap-template.yaml
```

考慮事項

ブートストラップは環境内のリソースをプロビジョニングするため、これらのリソースを で使用すると AWS 料金が発生する可能性があります AWS CDK。

AWS CDK ブートストラップをカスタマイズする

コマンド AWS CDK ラインインターフェイス (AWS CDK CLI) を使用するか、 AWS Cloud Development Kit (AWS CDK) ブートストラップテンプレートを変更してデプロイすることで、 AWS CloudFormation ブートストラップをカスタマイズできます。

ブートストラップの概要については、「」を参照してくださいブートストラッピング。

トピック

- CDKCLI を使用してブートストラップをカスタマイズする
- デフォルトのブートストラップテンプレートを変更する
- ブートストラップテンプレート契約に従う

CDKCLI を使用してブートストラップをカスタマイズする

CDK を使用してブートストラップをカスタマイズする方法の例を次に示しますCLI。すべての `cdk bootstrap` オプションのリストについては、「cdk bootstrap」を参照してください。

Amazon S3 バケットの名前を上書きする

`--bootstrap-bucket-name` オプションを使用して、デフォルトの Amazon S3 バケット名を上書きします。これには、テンプレート合成の変更が必要になる場合があります。詳細については、「CDK スタック合成をカスタマイズする」を参照してください。

Amazon S3 バケットのサーバー側の暗号化キーを変更する

デフォルトでは、ブートストラップスタックの Amazon S3 バケットは、サーバー側の暗号化に AWS マネージドキーを使用するように設定されています。既存のカスタマーマネージドキーを使用するには、 `--bootstrap-kms-key-id`オプションを使用し、使用する AWS Key Management Service （ AWS KMS) キーの値を指定します。暗号化キーをより詳細に制御したい場合は、カスタマーマネージドキーを使用する`--bootstrap-customer-key`ように を指定します。

が引き受けるデプロイロールに マネージドポリシーをアタッチする AWS CloudFormation

デフォルトでは、スタックは `AdministratorAccess`ポリシーを使用して完全な管理者アクセス許可でデプロイされます。独自の管理ポリシーを使用するには、 `--cloudformation-execution-policies`オプションを使用して、デプロイロールにアタッチする管理ポリシーの ARNs を指定します。

複数のポリシーを指定するには、カンマで区切られた単一の文字列を渡します。

```
$ cdk bootstrap --cloudformation-execution-policies "arn:aws:iam::aws:policy/
AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess"
```

デプロイの失敗を回避するには、指定したポリシーが、ブートストラップされる環境に実行するデプロイに十分であることを確認してください。

ブートストラップスタック内のリソース名に追加される修飾子を変更する

デフォルトでは、 hnb659fds 修飾子はブートストラップスタック内のリソースの物理 ID に追加されます。この値を変更するには、 `--qualifier`オプションを使用します。

この変更は、名前の競合を避けるために、同じ環境で複数のブートストラップスタックをプロビジョニングする場合に便利です。

修飾子の変更は、CDK 自体の自動テスト間の名前の分離を目的としています。 CloudFormation 実行ロールに付与された IAM アクセス許可を非常に正確に絞り込むことができない限り、1 つのアカウントに 2 つの異なるブートストラップスタックを配置しても、アクセス許可分離のメリットはありません。したがって、通常、この値を変更する必要はありません。

修飾子を変更する場合、CDK アプリは変更された値をスタックシンセサイザーに渡す必要があります。詳細については、「CDK スタック合成をカスタマイズする」を参照してください。

ブートストラップスタックにタグを追加する

の形式の `--tags`オプションKEY=VALUEを使用して、ブートストラップスタックに CloudFormation タグを追加します。

ブートストラップされる環境にデプロイ AWS アカウント できる追加の を指定する

`--trust` オプションを使用して、ブートストラップされる環境にデプロイ AWS アカウント できる追加の を指定します。デフォルトでは、ブートストラップを実行するアカウントは常に信頼されます。

このオプションは、別の環境Pipelineから CDK がデプロイされる環境をブートストラップする場合に便利です。

このオプションを使用する場合は、 も指定する必要があります`--cloudformation-execution-policies`。

既存のブートストラップスタックに信頼されたアカウントを追加するには、以前に指定したアカウントを含め、信頼するすべてのアカウントを指定する必要があります。信頼する新しいアカウントのみを指定すると、以前に信頼されたアカウントは削除されます。

以下は、2 つのアカウントを信頼する例です。

```
$ cdk bootstrap aws://123456789012/us-west-2 --trust 234567890123 --
trust 987654321098 --cloudformation-execution-policies arn:aws:iam::aws:policy/
AdministratorAccess
 #  Bootstrapping environment aws://123456789012/us-west-2...
Trusted accounts for deployment: 234567890123, 987654321098
Trusted accounts for lookup: (none)
Execution policies: arn:aws:iam::aws:policy/AdministratorAccess
CDKToolkit: creating CloudFormation changeset...
 #  Environment aws://123456789012/us-west-2 bootstrapped.
```

ブートストラップされる環境内の情報を検索 AWS アカウント できる追加の を指定する

`--trust-for-lookup` オプションを使用して、ブートストラップする環境からコンテキスト情報を検索 AWS アカウント できる を指定します。このオプションは、実際にはそれらのスタックを直接デプロイするアクセス許可を付与せずに、環境にデプロイされるスタックを合成するアクセス許可をアカウントに付与する場合に便利です。

ブートストラップスタックの終了保護を有効にする

ブートストラップスタックが削除されると、環境内で最初にプロビジョニングされた AWS リソースも削除されます。環境をブートストラップした後は、意図的に削除して再作成しないことをお勧めします。代わりに、 `cdk bootstrap` コマンドを再度実行して、ブートストラップスタックを新しいバージョンに更新してみてください。

`--termination-protection` オプションを使用して、ブートストラップスタックの終了保護設定を管理します。終了保護を有効にすると、ブートストラップスタックとそのリソースが誤って削除されるのを防ぐことができます。これは、ブートストラップスタックを誤って削除した場合Pipelines、一般的な復旧オプションがないため、CDK を使用する場合に特に重要です。

終了保護を有効にした後、 AWS CLI または AWS CloudFormation コンソールを使用して検証できます。

終了保護を有効にするには

1. 次のコマンドを実行して、新規または既存のブートストラップスタックで終了保護を有効にします。

```
$ cdk bootstrap --termination-protection
```

2. AWS CLI または CloudFormation コンソールを使用して確認します。以下に示しているのは、 AWS CLIを使用した例です。ブートストラップスタック名を変更した場合は、 をスタック名CDKToolkitに置き換えます。

```
$ aws cloudformation describe-stacks --stack-name CDKToolkit --query
 "Stacks[0].EnableTerminationProtection"
true
```

デフォルトのブートストラップテンプレートを変更する

CDK が提供するよりも多くのカスタマイズが必要な場合はCLI、必要に応じてブートストラップテンプレートを変更できます。次に、テンプレートをデプロイして環境をブートストラップします。

デフォルトのブートストラップテンプレートを変更してデプロイするには

1. `--show-template` オプションを使用してデフォルトのブートストラップテンプレートを取得します。デフォルトでは、CDK CLIはターミナルウィンドウにテンプレートを出力します。CDK CLI コマンドを変更して、テンプレートをローカルマシンに保存できます。以下に例を示します。

```
$ cdk bootstrap --show-template > my-bootstrap-template.yaml
```

2. 必要に応じてブートストラップテンプレートを変更します。変更を加える場合は、ブートストラップテンプレート契約に従う必要があります。ブートストラップテンプレート契約の詳細については、「」を参照してくださいブートストラップテンプレート契約に従う。

   カスタマイズが、後でデフォルトテンプレートcdk bootstrapを使用して実行されているユーザーによって誤って上書きされないようにするには、BootstrapVariantテンプレートパラメータのデフォルト値を変更します。CDK CLIは、現在デプロイされているテンプレートと同じBootstrapVariantか、同じかそれ以上のバージョンを持つテンプレートでのみ、ブートストラップスタックの上書きを許可します。

3. 任意のデプロイ方法を使用して、変更されたテンプレートを AWS CloudFormation デプロイします。CDK を使用する例を次に示しますCLI。

```
$ cdk bootstrap --template my-bootstrap-template.yaml
```

ブートストラップテンプレート契約に従う

ブートストラップをカスタマイズする場合、スタック合成動作をカスタマイズする必要がある場合があります。これにより、合成された CloudFormation テンプレートはブートストラップスタックと互換性が保たれます。詳細については、「CDK スタック合成をカスタマイズする」を参照してください。

スタック合成をカスタマイズする最も簡単な方法は、Stackインスタンスの DefaultStackSynthesizer クラスを変更することです。このクラスで提供できるもの以外のカスタマイズが必要な場合は、 を実装するクラスとして独自のシンセサイザーを記述できます IStackSynthesizer (おそらく から取得) DefaultStackSynthesizer。

ブートストラップをカスタマイズするときは、ブートストラップテンプレート契約に従って との互換性を維持しますDefaultStackSynthesizer。ブートストラップテンプレート契約を超えてブートストラップを変更する場合は、独自のシンセサイザーを作成する必要があります。

バージョニング

ブートストラップテンプレートには、よく知られている名前の Amazon EC2 Systems Manager (SSM) パラメータを作成するリソースと、テンプレートのバージョンを反映する出力が含まれている必要があります。

```
Resources:
  CdkBootstrapVersion:
    Type: AWS::SSM::Parameter
    Properties:
      Type: String
      Name:
        Fn::Sub: '/cdk-bootstrap/${Qualifier}/version'
      Value: 4
Outputs:
  BootstrapVersion:
    Value:
      Fn::GetAtt: [CdkBootstrapVersion, Value]
```

ロール

DefaultStackSynthesizer には、5 つの異なる目的で 5 つの IAM ロールが必要です。デフォルトのロールを使用していない場合は、 DefaultStackSynthesizer オブジェクト内で IAM ロール ARNsを指定する必要があります。ロールは次のとおりです。

- デプロイロールは CDK CLIと が引き受け AWS CodePipeline 、環境にデプロイします。そのAssumeRolePolicyコントロールは、環境にデプロイできるユーザーを制御します。テンプレートには、このロールに必要なアクセス許可が表示されます。
- ルックアップロールは、環境でコンテキストルックアップを実行するCLIために CDK によって引き受けられます。そのAssumeRolePolicyコントロールは、環境にデプロイできるユーザーを制御します。このロールに必要なアクセス許可は、 テンプレートで確認できます。
- ファイル発行ロールとイメージ発行ロールは、CDK CLIと AWS CodeBuild プロジェクトが引き受け、アセットを環境に発行します。これらは、それぞれ Amazon S3 バケットと Amazon ECR リポジトリへの書き込みに使用されます。これらのロールには、これらのリソースへの書き込みアクセスが必要です。
- AWS CloudFormation 実行ロールは に渡され AWS CloudFormation 、実際のデプロイを実行します。そのアクセス許可は、デプロイが実行されるアクセス許可です。アクセス許可は、管理ポリシー ARNs。

出力

CDK CLIでは、ブートストラップスタックに次の CloudFormation 出力が存在する必要があります。

- `BucketName` - ファイルアセットバケットの名前。
- `BucketDomainName` – ドメイン名形式のファイルアセットバケット。
- `BootstrapVersion` – ブートストラップスタックの現在のバージョン。

テンプレート履歴

ブートストラップテンプレートはバージョニングされ、それ AWS CDK 自体で時間の経過とともに進化します。独自のブートストラップテンプレートを指定する場合は、正規のデフォルトテンプレートで最新の状態を維持します。テンプレートが引き続きすべての CDK 機能で動作するようにしたい。詳細については、「ブートストラップテンプレートのバージョン履歴」を参照してください。

CDK スタック合成の設定とカスタマイズ

AWS Cloud Development Kit (AWS CDK) スタックをデプロイする前に、まずスタックを合成する必要があります。スタック合成は、CDK スタックから AWS CloudFormation テンプレートを作成するプロセスです。 CloudFormation テンプレートは、 でリソースをプロビジョニングするためにデプロイしたものです AWS。

トピック

CDK スタック合成を設定する

CDK スタック合成は、`Stack`インスタンスのシンセサイザープロパティを使用して設定できます。このプロパティは、スタックのテンプレートの合成方法を指定します。基本的な例を次に示します。

TypeScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

Python

```
MyStack(self, "MyStack",
```

```
    # stack properties
    synthesizer=DefaultStackSynthesizer(
        # synthesizer properties
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  // stack properties
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    // synthesizer properties
    .build())
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
// stack properties
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        // synthesizer properties
    })
});
```

デフォルトでは、 は synthesizerプロパティ[DefaultStackSynthesizer]に
AWS CDK を使用します。の値を指定しない場合synthesizer、CDK は を使用しま
すDefaultStackSynthesizer。

合成が正しく機能するためには、CDK はデプロイ先の環境内のブートストラップされたリソースについて知っておく必要があります。これにより、合成されたテンプレートのリソースがブートストラップスタックのリソースと正しく相互作用します。

ブートストラップスタックやテンプレートの変更など、ブートストラップを変更していない場合は、スタック合成を変更する必要はありません。シンセサイザープロパティを指定する必要はありません。CDK はデフォルトの DefaultStackSynthesizer クラスを使用して、ブートストラップスタックと適切にやり取りするように CDK スタック合成を設定します。

CDK スタックの合成

CDK スタックを合成するには、 AWS CDK コマンドラインインターフェイス (AWS CDK CLI) cdk synth コマンドを使用します。このコマンドで使用できるオプションなど、このコマンドの詳細については、「」を参照してくださいcdk synthesize。

CDK アプリケーションに 1 つのスタックが含まれている場合、またはすべてのスタックを合成する場合は、CDK スタック名を引数として指定する必要はありません。デフォルトでは、CDK CLIは CDK スタックを テンプレートに AWS CloudFormation 合成します。各スタックのjsonフォーマットされたテンプレートが cdk.out ディレクトリに保存されます。アプリケーションに 1 yaml つのスタックが含まれている場合、フォーマットされたテンプレートが に出力されますstdout。以下に例を示します。

```
$ cdk synth
Resources:
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:H4sIAAAAAAAA/
yXGyQ2AIBAAwFr4wwrEDugAK0DAZEGXhEMfxt6N8TWjQc0SJHNXEz5kseMK99Kdz9xsZGMro/
r43RQK2LHQw6mECKlNp5agFCiWGqKogzoeEezvC612NX1aAAAA
    Metadata:
      aws:cdk:path: CdkAppStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
    ...
```

CDK アプリケーションに複数のスタックが含まれている場合は、スタックの論理 ID を指定して 1 つのスタックを合成できます。以下に例を示します。

```
$ cdk synth MyStackName
```

スタックを合成して を実行しない場合cdk deploy、CDK CLIはデプロイ前にスタックを自動的に合成します。

CDK スタック合成をカスタマイズする

ブートストラップを変更する場合は、CDK スタック合成をカスタマイズする必要がある場合があります。CDK スタック合成をカスタマイズするには、StackインスタンスのdefaultStackSynthesizerプロパティを変更できます。これらのプロパティのいずれも必要

なカスタマイズを提供していない場合は、 を実装するクラスとしてシンセサイザーを記述できますIStackSynthesizer (おそらく から取得されますDefaultStackSynthesizer)。

修飾子を変更する

この修飾子は、ブートストラップ中に作成されたリソースの名前に追加されます。修飾子を変更した場合は、同じ修飾子を使用するように CDK スタック合成をカスタマイズする必要があります。

修飾子を変更するには、修飾子プロパティを使用してシンセサイザをインスタンス化DefaultStackSynthesizerするか、CDK プロジェクトの cdk.json ファイルで修飾子をコンテキストキーとして設定して、 を設定します。

以下は、 qualifierプロパティを使用してシンセサイザーをインスタンス化する例です。

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=DefaultStackSynthesizer(
        qualifier="MYQUALIFIER"
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
```

```
    .qualifier("MYQUALIFIER")
    .build())
  .build();
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        Qualifier = "MYQUALIFIER"
    })
});
```

以下は、 で修飾子をコンテキストキーとして設定する例ですcdk.json。

```
{
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "MYQUALIFIER"
  }
}
```

リソース名を変更する

その他のプロパティはすべて、ブートストラップテンプレート内のリソースの名前DefaultStackSynthesizerに関連しています。ブートストラップテンプレートを変更し、リソース名または命名スキームを変更した場合にのみ、これらのプロパティのいずれかを指定する必要があります。

すべてのプロパティは、特別なプレースホルダー ${Qualifier}、${AWS::Partition}、${AWS::AccountId}、および を受け入れます${AWS::Region}。これらのプレースホルダーは、 qualifierパラメータの値と、スタックの環境の AWS パーティション、アカウント ID、および AWS リージョン 値でそれぞれ置き換えられます。

次の例は、シンセサイザーをインスタンス化しているかのように、 で最も一般的に使用されるプロパティDefaultStackSynthesizerとデフォルト値を示しています。詳細なリストについては、「」を参照してくださいDefaultStackSynthesizerProps。

TypeScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,

})
```

JavaScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,
})
```

Python

```
DefaultStackSynthesizer(
  # Name of the S3 bucket for file assets
  file_assets_bucket_name="cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
  bucket_prefix="",

  # Name of the ECR repository for Docker image assets
  image_assets_repository_name="cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}",

  # ARN of the role assumed by the CLI and Pipeline to deploy here
  deploy_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
  deploy_role_external_id="",

  # ARN of the role used for file asset publishing (assumed from the CLI role)
  file_asset_publishing_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
  file_asset_publishing_external_id="",

  # ARN of the role used for Docker asset publishing (assumed from the CLI role)
  image_asset_publishing_role_arn="arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
  image_asset_publishing_external_id="",

  # ARN of the role passed to CloudFormation to execute the deployments
  cloud_formation_execution_role="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}",

  # ARN of the role used to look up context information in an environment
  lookup_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
  lookup_role_external_id="",

  # Name of the SSM parameter which describes the bootstrap stack version number
  bootstrap_stack_version_ssm_parameter="/cdk-bootstrap/${Qualifier}/version",

  # Add a rule to every template which verifies the required bootstrap stack version
  generate_bootstrap_version_rule=True,
)
```

Java

```
DefaultStackSynthesizer.Builder.create()
  // Name of the S3 bucket for file assets
  .fileAssetsBucketName("cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}")
  .bucketPrefix('')

  // Name of the ECR repository for Docker image assets
  .imageAssetsRepositoryName("cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}")

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  .deployRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}")
  .deployRoleExternalId("")

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  .fileAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}")
  .fileAssetPublishingExternalId("")

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  .imageAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}")
  .imageAssetPublishingExternalId("")

  // ARN of the role passed to CloudFormation to execute the deployments
  .cloudFormationExecutionRole("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}")

  .lookupRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}")
  .lookupRoleExternalId("")

  // Name of the SSM parameter which describes the bootstrap stack version number
  .bootstrapStackVersionSsmParameter("/cdk-bootstrap/${Qualifier}/version")

  // Add a rule to every template which verifies the required bootstrap stack
 version
  .generateBootstrapVersionRule(true)
.build()
```

C#

```
new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
{
    // Name of the S3 bucket for file assets
    FileAssetsBucketName = "cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
    BucketPrefix = "",

    // Name of the ECR repository for Docker image assets
    ImageAssetsRepositoryName = "cdk-${Qualifier}-container-assets-
${AWS::AccountId}-${AWS::Region}",

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    DeployRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
    DeployRoleExternalId = "",

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    FileAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
    FileAssetPublishingExternalId = "",

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    ImageAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
    ImageAssetPublishingExternalId = "",

    // ARN of the role passed to CloudFormation to execute the deployments
    CloudFormationExecutionRole = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-
${AWS::Region}",

    LookupRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
    LookupRoleExternalId = "",

    // Name of the SSM parameter which describes the bootstrap stack version number
    BootstrapStackVersionSsmParameter = "/cdk-bootstrap/${Qualifier}/version",

    // Add a rule to every template which verifies the required bootstrap stack
 version
    GenerateBootstrapVersionRule = true,
```

```
})
```

AWS CDK アプリケーションの開発

AWS Cloud Development Kit (AWS CDK) アプリケーションを開発します。

トピック

コンストラクトライブラリからの AWS コンストラクトのカスタマイズ

エスケープハッチ、未加工の上書き、カスタムリソースを使用して、 AWS コンストラクトライブラリからコンストラクトをカスタマイズします。

トピック

エスケープハッチの使用

AWS コンストラクトライブラリは、さまざまなレベルの抽象化のコンストラクトを提供します。

最上位レベルでは、 AWS CDK アプリケーションとその中のスタックは、クラウドインフラストラクチャ全体、またはその重要なチャンクを抽象化したものです。パラメータ化して、さまざまな環境にデプロイしたり、さまざまなニーズに合わせてデプロイしたりできます。

抽象化は、クラウドアプリケーションを設計および実装するための強力なツールです。 AWS CDK は、抽象化を使用して構築するだけでなく、新しい抽象化を作成するための能力も提供します。既存のオープンソースの L2 および L3 コンストラクトをガイダンスとして使用して、独自の L2 および L3 コンストラクトを構築して、組織のベストプラクティスと提案を反映することができます。

抽象化は完全ではありません。また、優れた抽象化でも可能なすべてのユースケースに対応することはできません。開発中に、ニーズにほぼ適合するコンストラクトが見つかり、小規模または大規模なカスタマイズが必要になる場合があります。

このため、 AWS CDK にはコンストラクトモデルから抜け出す方法があります。これには、下位レベルの抽象化または別のモデルへの完全な移行が含まれます。エスケープハッチを使用すると、AWS CDK パラダイムをエスケープし、ニーズに合った方法でカスタマイズできます。その後、変更を新しいコンストラクトにまとめることで、根本的な複雑さを抽象化し、他のデベロッパーにクリーンな API を提供できます。

エスケープハッチを使用できる状況の例を次に示します。

- AWS サービス機能は を通じて使用できますが AWS CloudFormation、L2 コンストラクトはありません。
- AWS サービス機能は を通じて利用でき AWS CloudFormation、サービスの L2 コンストラクトはありますが、まだ機能を公開していません。L2 コンストラクトは CDK チームによってキュレートされるため、新機能ではすぐには利用できない場合があります。
- この機能は、 ではまだ利用できません AWS CloudFormation 。

  機能が利用可能かどうかを判断するには AWS CloudFormation、AWS 「リソースタイプとプロパティタイプのリファレンス」を参照してください。

L1 コンストラクトのエスケープハッチを開発する

サービスで L2 コンストラクトを使用できない場合は、自動的に生成された L1 コンストラクトを使用できます。これらのリソースは、 `CfnBucket`や などCfn、 で始まる名前で認識できます`CfnRole`。同等の AWS CloudFormation リソースを使用するのとまったく同じ方法でインスタンス化します。

例えば、分析を有効にして低レベルの Amazon S3 バケット L1 をインスタンス化するには、次のように記述します。

TypeScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config',
      // ...
    }
  ]
});
```

JavaScript

```
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config'
      // ...
    }
  ]
});
```

Python

```
s3.CfnBucket(self, "MyBucket",
    analytics_configurations: [
        dict(id="Config",
             # ...
             )
    ]
)
```

Java

```
CfnBucket.Builder.create(this, "MyBucket")
    .analyticsConfigurations(Arrays.asList(java.util.Map.of(    // Java 9 or later
        "id", "Config", // ...
    ))).build();
```

C#

```
new CfnBucket(this, 'MyBucket', new CfnBucketProps {
    AnalyticsConfigurations = new Dictionary<string, string>
    {
        ["id"] = "Config",
        // ...
    }
});
```

対応するCfnXxxクラスを持たないリソースを定義したいというまれなケースがあります。これは、リソース仕様でまだ公開されていない新しい AWS CloudFormation リソースタイプである可能性があります。このような場合は、 cdk.CfnResourceを直接インスタンス化し、リソースタイプとプロパティを指定できます。以下の例ではこれを示しています。

TypeScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config',
        // ...
      }
    ]
  }
});
```

JavaScript

```
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config'
        // ...
      }
```

```
    ]
  }
});
```

Python

```
cdk.CfnResource(self, 'MyBucket',
  type="AWS::S3::Bucket",
  properties=dict(
    # Note the PascalCase here! These are CloudFormation identifiers.
    "AnalyticsConfigurations": [
      {
        "Id": "Config",
        # ...
      }
    ]
  }
)
```

Java

```
CfnResource.Builder.create(this, "MyBucket")
        .type("AWS::S3::Bucket")
        .properties(java.util.Map.of(    // Map.of requires Java 9 or later
            // Note the PascalCase here! These are CloudFormation identifiers
            "AnalyticsConfigurations", Arrays.asList(
                    java.util.Map.of("Id", "Config", // ...
                    ))))
        .build();
```

C#

```
new CfnResource(this, "MyBucket", new CfnResourceProps
{
    Type = "AWS::S3::Bucket",
    Properties = new Dictionary<string, object>
    {   // Note the PascalCase here! These are CloudFormation identifiers
        ["AnalyticsConfigurations"] = new Dictionary<string, string>[]
        {
            new Dictionary<string, string> {
                ["Id"] = "Config"
            }
```

```
        }
    }
});
```

L2 コンストラクトのエスケープハッチを開発する

L2 コンストラクトに機能が欠落している場合、または問題を回避しようとしている場合は、L2 コンストラクトによってカプセル化されている L1 コンストラクトを変更できます。

すべての L2 コンストラクトには、対応する L1 コンストラクトがそれらに含まれます。例えば、高レベルBucketコンストラクトは低レベルCfnBucketコンストラクトをラップします。は AWS CloudFormation リソースに直接CfnBucket対応しているため、 で利用できるすべての機能が公開されます AWS CloudFormation。

L1 コンストラクトにアクセスするための基本的なアプローチは、`construct.node.defaultChild` (Python: `default_child`) を使用し、正しいタイプにキャストし (必要な場合) 、プロパティを変更することです。ここでも、 の例を見てみましょうBucket。

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }
];
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config'
    // ...
```

```
  }
];
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
    {
        "id": "Config",
        # ...
    }
]
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations(
        Arrays.asList(java.util.Map.of(    // Java 9 or later
            "Id", "Config", // ...
        ));
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List<object> {
    new Dictionary<string, string>
    {
        ["Id"] = "Config",
        // ...
    }
};
```

このオブジェクトを使用して、 Metadataや などの AWS CloudFormation オプションを変更することもできますUpdatePolicy。

TypeScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

JavaScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

Python

```
cfn_bucket.cfn_options.metadata = {
    "MetadataKey": "MetadataValue"
}
```

Java

```
cfnBucket.getCfnOptions().setMetadata(java.util.Map.of(    // Java 9+
    "MetadataKey", "Metadatavalue"));
```

C#

```
cfnBucket.CfnOptions.Metadata = new Dictionary<string, object>
{
    ["MetadataKey"] = "Metadatavalue"
};
```

ハッチのエスケープ解除

には、抽象化レベルを上げる機能 AWS CDK も用意されています。抽象化レベルは「アンエスケープ」ハッチと呼ばれます。などの L1 コンストラクトがある場合はCfnBucket、新しい L2 コンストラクト (Bucketこの場合は) を作成して L1 コンストラクトをラップできます。

これは、L1 リソースを作成するが、L2 リソースを必要とするコンストラクトで使用する場合に便利です。また、L1 コンストラクトでは利用できない .grantXxxx() のような便利な方法を使用する場合にも役立ちます。

という名前の L2 クラスで静的メソッドを使用して、より高い抽象化レベルに移行しま
す`.fromCfnXxxx()`。例えば、Amazon S3 バケット`Bucket.fromCfnBucket()`の場合です。L1
リソースは唯一のパラメータです。

TypeScript

```
b1 = new s3.CfnBucket(this, "buck09", { ... });
b2 = s3.Bucket.fromCfnBucket(b1);
```

JavaScript

```
b1 = new s3.CfnBucket(this, "buck09", { ...} );
b2 = s3.Bucket.fromCfnBucket(b1);
```

Python

```
b1 = s3.CfnBucket(self, "buck09", ...)
 b2 = s3.from_cfn_bucket(b1)
```

Java

```
CfnBucket b1 = CfnBucket.Builder.create(this, "buck09")
        // ....
                          .build();
IBucket b2 = Bucket.fromCfnBucket(b1);
```

C#

```
var b1 = new CfnBucket(this, "buck09", new CfnBucketProps { ... });
var v2 = Bucket.FromCfnBucket(b1);
```

L1 コンストラクトから作成された L2 コンストラクトは、リソース名、ARN、またはルックアップ
から作成されたものと同様に、L1 ARNs リソースを参照するプロキシオブジェクトです。これらの
コンストラクトを変更しても、最終的な合成 AWS CloudFormation テンプレートには影響しません
(L1 リソースがあるため、代わりに変更できます)。プロキシオブジェクトの詳細については、「」
を参照してくださいthe section called “AWS アカウント内のリソースの参照”。

混同を避けるため、同じ L1 コンストラクトを参照する複数の L2 コンストラクトを作成しないでく
ださい。 L1 例えば、前のセクション「」の手法`Bucket`を使用して `CfnBucket`から を抽出する場

合、その Bucket.fromCfnBucket()で を呼び出して 2 つ目のBucketインスタンスを作成しないでくださいCfnBucket。実際には期待どおりに機能しますが (合成AWS::S3::Bucketされているもののみ)、コードの保守が困難になります。

Raw オーバーライド

L1 コンストラクトに欠落しているプロパティがある場合は、raw オーバーライドを使用してすべての型付けをバイパスできます。これにより、合成されたプロパティを削除することもできます。

次の例に示すように、いずれかのaddOverrideメソッド (Python: add_override) を使用します。

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild ;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');
```

```
// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Use dot notation to address inside the resource template fragment
cfn_bucket.add_override("Properties.VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_deletion_override("Properties.VersioningConfiguration.Status")

# use index (0 here) to address an element of a list
cfn_bucket.add_override("Properties.Tags.0.Value", "NewValue")
cfn_bucket.add_deletion_override("Properties.Tags.0")

# addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfn_bucket.add_property_override("VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_property_deletion_override("VersioningConfiguration.Status")
cfn_bucket.add_property_override("Tags.0.Value", "NewValue")
cfn_bucket.add_property_deletion_override("Tags.0")
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.addDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.addOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.addDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
```

```
cfnBucket.addPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.addPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.addPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.addPropertyDeletionOverride("Tags.0");
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment
cfnBucket.AddOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.AddOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.AddDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.AddPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.AddPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.AddPropertyDeletionOverride("Tags.0");
```

カスタムリソース

この機能が を介してではなく AWS CloudFormation、直接 API コールを介してのみ使用できない場合は、 AWS CloudFormation カスタムリソースを作成して、必要な API コールを行う必要があります。を使用してカスタムリソース AWS CDK を記述し、通常のコンストラクトインターフェイスにラップできます。コンストラクトのコンシューマーの観点から見ると、エクスペリエンスはネイティブに見えます。

カスタムリソースを構築するには、リソースの CREATE、、UPDATEおよび DELETEライフサイクルイベントに応答する Lambda 関数を記述する必要があります。カスタムリソースが単一の API コールのみを行う必要がある場合は、 の使用を検討してくださいAwsCustomResource。これにより、AWS CloudFormation デプロイ中に任意の SDK 呼び出しを実行できます。それ以外の場合は、独自の Lambda 関数を作成して、完了するために必要な作業を実行する必要があります。

件名が大きすぎて、ここでは完全に説明できませんが、次のリンクから始めてください。

- カスタムリソース
- カスタムリソースの例
- より完全な例については、CDK 標準ライブラリの DnsValidatedCertificate クラスを参照してください。これはカスタムリソースとして実装されます。

環境変数から値を取得する

環境変数の値を取得するには、次のようなコードを使用します。このコードは、環境変数 の値を取得しますMYBUCKET。

TypeScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

JavaScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

Python

```
import os

# Raises KeyError if environment variable doesn't exist
bucket_name = os.environ["MYBUCKET"]

# Sets bucket_name to None if environment variable doesn't exist
bucket_name = os.getenv("MYBUCKET")

# Sets bucket_name to a default if env var doesn't exist
bucket_name = os.getenv("MYBUCKET", "DefaultName")
```

Java

```
// Sets bucketName to null if environment variable doesn't exist
String bucketName = System.getenv("MYBUCKET");

// Sets bucketName to a default if env var doesn't exist
String bucketName = System.getenv("MYBUCKET");
if (bucketName == null) bucketName = "DefaultName";
```

C#

```
using System;

// Sets bucket name to null if environment variable doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET");

// Sets bucket_name to a default if env var doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET") ?? "DefaultName";
```

AWS CloudFormation 値を使用する

で AWS CloudFormation パラメータを使用する方法については、the section called “パラメータ”「」を参照してください AWS CDK。

既存の AWS CloudFormation テンプレート内のリソースへの参照を取得するには、「」を参照してくださいthe section called “ AWS CloudFormation テンプレートをインポートする”。

既存の AWS CloudFormation テンプレートをインポートする

コンストラクトを使用してリソースを L1 `cloudformation-include.CfnInclude`コンストラクトに変換することで、 AWS CloudFormation テンプレートから AWS Cloud Development Kit (AWS CDK) アプリケーションにリソースをインポートします。

インポート後、これらのリソースを AWS CDK コードで最初に定義した場合と同じ方法でアプリで操作できます。これらの L1 コンストラクトは、上位レベルの AWS CDK コンストラクト内でも使用できます。例えば、これにより、L2 アクセス許可付与メソッドを、それらが定義するリソースで使用できます。

`cloudformation-include.CfnInclude` コンストラクトは基本的に、 AWS CloudFormation テンプレート内の任意のリソースに AWS CDK API ラッパーを追加します。この機能を使用して、既

存の AWS CloudFormation テンプレートを一度に 1 つの AWS CDK 1 つの にインポートします。これにより、 AWS CDK コンストラクトを使用して既存のリソースを管理し、高レベルの抽象化の利点を活用できます。この機能を使用して、 AWS CDK コンストラクト API を提供することで AWS CloudFormation 、 テンプレートを AWS CDK デベロッパーに提供することもできます。

> **Note**
>
> AWS CDK v1 には`aws-cdk-lib.CfnInclude`、以前は同じ汎用目的で使用されていた も含まれています。ただし、 の機能の多くは不足していますcloudformation-include.CfnInclude。

トピック

AWS CloudFormation テンプレートのインポート

以下は、このトピックで例を提供するために使用するサンプル AWS CloudFormation テンプレートです。テンプレートをコピーして として保存`my-template.json`し、それに従います。これらの例を実行したら、既存のデプロイ済み AWS CloudFormation テンプレートのいずれかを使用してさらに詳しく調べることができます。コンソールから AWS CloudFormation 取得できます。

```
{
  "Resources": {
    "MyBucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "MyBucket",
      }
    }
  }
}
```

JSON テンプレートまたは YAML テンプレートを使用できます。YAML パーサーは受け入れる内容が若干異なる可能性があるため、JSON が利用可能な場合は推奨されます。

以下は、 を使用してサンプルテンプレートを AWS CDK アプリケーションにインポートする方法の例ですcloudformation-include。テンプレートは CDK スタックのコンテキスト内でインポートされます。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as cfninc from 'aws-cdk-lib/cloudformation-include';
import { Construct } from 'constructs';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const cfninc = require('aws-cdk-lib/cloudformation-include');

class MyStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}

module.exports = { MyStack }
```

Python

```
import aws_cdk as cdk
from aws_cdk import cloudformation_include as cfn_inc
from constructs import Construct

class MyStack(cdk.Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        template = cfn_inc.CfnInclude(self, "Template",
            template_file="my-template.json")
```

Java

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.cloudformation.include.CfnInclude;
import software.constructs.Construct;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        CfnInclude template = CfnInclude.Builder.create(this, "Template")
         .templateFile("my-template.json")
         .build();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using cfnInc = Amazon.CDK.CloudFormation.Include;

namespace MyApp
{
```

```
    public class MyStack : Stack
    {
        internal MyStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            var template = new cfnInc.CfnInclude(this, "Template", new
 cfnInc.CfnIncludeProps
            {
                TemplateFile = "my-template.json"
            });
        }
    }
}
```

デフォルトでは、リソースをインポートすると、リソースの元の論理 ID がテンプレートから保持されます。この動作は、論理 IDs を保持する必要がある への AWS CloudFormation テンプレートのインポートに適しています。 は AWS CDK、インポートされたこれらのリソースを AWS CloudFormation テンプレートから同じリソースとして認識するためにこの情報 AWS CloudFormation を必要とします。

テンプレートの AWS CDK コンストラクトラッパーを開発して他の AWS CDK デベロッパーが使用できるようにする場合は、代わりに に新しいリソース IDs AWS CDK を生成してもらいます。これにより、名前の競合なしでスタックで コンストラクトを複数回使用できます。これを行うには、テンプレートをインポート`false`するときに `preserveLogicalIds`プロパティを に設定します。以下に例を示します。

TypeScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    preserve_logical_ids=False)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .preserveLogicalIds(false)
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfn_inc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    PreserveLogicalIds = false
});
```

インポートされたリソースを AWS CDK アプリの制御下に置くには、 スタックを に追加しますApp。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { MyStack } from '../lib/my-stack';

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { MyStack } = require('../lib/my-stack');

const app = new cdk.App();
new MyStack(app, 'MyStack');
```

Python

```
import aws_cdk as cdk
from mystack.my_stack import MyStack

app = cdk.App()
MyStack(app, "MyStack")
```

Java

```
import software.amazon.awscdk.App;

public class MyApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyStack(app, "MyStack");
    }
}
```

C#

```
using Amazon.CDK;

namespace CdkApp
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyStack(app, "MyStack");
        }
    }
}
```

スタック内の AWS リソースに意図しない変更がないことを確認するには、差分を実行します。cdk diff コマンドを使用して、 AWS CDK固有のメタデータを省略します AWS CDK CLI。以下に例を示します。

```
cdk diff --no-version-reporting --no-path-metadata --no-asset-metadata
```

AWS CloudFormation テンプレートをインポートすると、 AWS CDK アプリケーションはインポートされたリソースの信頼できるソースになります。リソースを変更するには、 AWS CDK アプリで変更し、 cdk deploy コマンドを使用してデプロイします AWS CDK CLI。

インポートされたリソースへのアクセス

サンプルコードtemplateの名前は、インポートされた AWS CloudFormation テンプレートを表します。リソースからリソースにアクセスするには、 オブジェクトの getResource()メソッドを使用します。返されたリソースに特定の種類のリソースとしてアクセスするには、結果を目的のタイプにキャストします。これは Python または では必要ありません JavaScript。以下に例を示します。

TypeScript

```
const cfnBucket = template.getResource('MyBucket') as s3.CfnBucket;
```

JavaScript

```
const cfnBucket = template.getResource('MyBucket');
```

Python

```
cfn_bucket = template.get_resource("MyBucket")
```

Java

```
CfnBucket cfnBucket = (CfnBucket)template.getResource("MyBucket");
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
```

この例では、 cfnBucketは aws-s3.CfnBucket クラスのインスタンスになりました。これは、対応する AWS CloudFormation リソースを表す L1 コンストラクトです。これは、そのタイプの他のリソースと同様に扱うことができます。例えば、 bucket.attrArnプロパティを使用して ARN 値を取得できます。

代わりに L1 CfnBucketリソースを L2 aws-s3.Bucketインスタンスにラップするには、静的メソッド fromBucketArn()、、fromBucketAttributes()または を使用しますfromBucketName()。通常、このfromBucketName()方法は最も便利です。以下に例を示します。

TypeScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

JavaScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

Python

```
bucket = s3.Bucket.from_bucket_name(self, "Bucket", cfn_bucket.ref)
```

Java

```
Bucket bucket = (Bucket)Bucket.fromBucketName(this, "Bucket", cfnBucket.getRef());
```

C#

```
var bucket = (Bucket)Bucket.FromBucketName(this, "Bucket", cfnBucket.Ref);
```

他の L2 コンストラクトは、既存のリソースからコンストラクトを作成するための同様の方法を持っています。

L1 コンストラクトを L2 コンストラクトにラップしても、新しいリソースは作成されません。この例では、2 番目の S3; バケットを作成していません。代わりに、新しいBucketインスタンスは既存の をカプセル化しますCfnBucket。

この例では、 bucketは他の L2 Bucketコンストラクトと同様に動作する L2 コンストラクトになりました。例えば、バケットの便利な grantWrite()メソッドを使用して、 AWS Lambda 関数にバケットへの書き込みアクセスを許可できます。必要な AWS Identity and Access Management (IAM) ポリシーを手動で定義する必要はありません。以下に例を示します。

TypeScript

```
bucket.grantWrite(lambdaFunc);
```

JavaScript

```
bucket.grantWrite(lambdaFunc);
```

Python

```
bucket.grant_write(lambda_func)
```

Java

```
bucket.grantWrite(lambdaFunc);
```

C#

```
bucket.GrantWrite(lambdaFunc);
```

パラメータの置き換え

AWS CloudFormation テンプレートにパラメータが含まれている場合は、 `parameters`プロパティを使用して、インポート時にパラメータをビルド時の値に置き換えることができます。次の例では、`UploadBucket`パラメータを、 AWS CDK コードの他の場所で定義されたバケットの ARN に置き換えます。

TypeScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
```

```
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    parameters=dict(UploadBucket=bucket.bucket_arn)
)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .parameters(java.util.Map.of(    // Map.of requires Java 9+
   "UploadBucket", bucket.getBucketArn()))
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfnInc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    Parameters = new Dictionary<string, string>
    {
        { "UploadBucket", bucket.BucketArn }
    }
});
```

その他のテンプレート要素

リソースだけでなく、任意の AWS CloudFormation テンプレート要素をインポートできます。インポートされた AWS CDK 要素はスタックの一部になります。これらの要素をインポートするには、`CfnInclude` オブジェクトの次のメソッドを使用します。

- `getCondition()` – AWS CloudFormation 条件 。
- `getHook()` – ブルー/グリーンデプロイ AWS CloudFormation 用のフック。

- `getMapping()` - AWS CloudFormation マッピング。
- `getOutput()` - AWS CloudFormation を出力します。
- `getParameter()` – AWS CloudFormation パラメータ 。
- `getRule()` – テンプレートの AWS Service Catalog AWS CloudFormation ルール。

これらのメソッドはそれぞれ、特定のタイプの AWS CloudFormation 要素を表すクラスのインスタンスを返します。これらのオブジェクトは変更可能です。それらに加えた変更は、スタックから AWS CDK 生成されたテンプレートに表示されます。以下は、テンプレートからパラメータをインポートし、デフォルト値を変更する例です。

TypeScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

JavaScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

Python

```
param = template.get_parameter("MyParameter")
param.default = "AWS CDK"
```

Java

```
CfnParameter param = template.getParameter("MyParameter");
param.setDefaultValue("AWS CDK")
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
var param = template.GetParameter("MyParameter");
param.Default = "AWS CDK";
```

ネストされたスタック

ネストされたスタックをインポートするには、メインテンプレートをインポートするとき、または後で指定します。ネストされたテンプレートはローカルファイルに保存する必要がありますが、メインテンプレートではNestedStackリソースとして参照されます。また、 AWS CDK コードで使用されるリソース名は、メインテンプレートのネストされたスタックに使用される名前と一致する必要があります。

メインテンプレートでこのリソース定義を指定すると、次のコードは参照されるネストされたスタックを双方向にインポートする方法を示しています。

```
"NestedStack": {
  "Type": "AWS::CloudFormation::Stack",
  "Properties": {
    "TemplateURL": "https://my-s3-template-source.s3.amazonaws.com/nested-stack.json"
  }
```

TypeScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedTemplate', {
  templateFile: 'nested-template.json',
});
```

JavaScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
```

```
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedStack', {
  templateFile: 'my-nested-template.json',
});
```

Python

```
# include nested stack when importing main stack
main_template = cfn_inc.CfnInclude(self, "MainStack",
    template_file="main-template.json",
    load_nested_stacks=dict(NestedStack=
        cfn_inc.CfnIncludeProps(template_file="nested-template.json")))

# or add it some time after importing the main stack
nested_template = main_template.load_nested_stack("NestedStack",
    template_file="nested-template.json")
```

Java

```
CfnInclude mainTemplate = CfnInclude.Builder.create(this, "MainStack")
 .templateFile("main-template.json")
 .loadNestedStacks(java.util.Map.of(    // Map.of requires Java 9+
    "NestedStack", CfnIncludeProps.builder()
    .templateFile("nested-template.json").build()))
 .build();

// or add it some time after importing the main stack
IncludedNestedStack nestedTemplate = mainTemplate.loadNestedStack("NestedTemplate",
 CfnIncludeProps.builder()
  .templateFile("nested-template.json")
  .build());
```

C#

```
// include nested stack when importing main stack
var mainTemplate = new cfnInc.CfnInclude(this, "MainStack", new
 cfnInc.CfnIncludeProps
```

```
{
    TemplateFile = "main-template.json",
    LoadNestedStacks = new Dictionary<string, cfnInc.ICfnIncludeProps>
    {
        { "NestedStack", new cfnInc.CfnIncludeProps { TemplateFile = "nested-
template.json" } }
    }
});

// or add it some time after importing the main stack
var nestedTemplate = mainTemplate.LoadNestedStack("NestedTemplate", new
 cfnInc.CfnIncludeProps {
    TemplateFile = 'nested-template.json'
});
```

どちらの方法でも、ネストされた複数のスタックをインポートできます。メインテンプレートをインポートするときは、ネストされた各スタックのリソース名とそのテンプレートファイルのマッピングを指定します。このマッピングには、任意の数のエントリを含めることができます。最初のインポート後にこれを行うには、ネストされたスタックごとに を loadNestedStack()1 回呼び出します。

ネストされたスタックをインポートしたら、メインテンプレートの getNestedStack()メソッドを使用してスタックにアクセスできます。

TypeScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

JavaScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

Python

```
nested_stack = main_template.get_nested_stack("NestedStack").stack
```

Java

```
NestedStack nestedStack = mainTemplate.getNestedStack("NestedStack").getStack();
```

C#

```
var nestedStack = mainTemplate.GetNestedStack("NestedStack").Stack;
```

`getNestedStack()` メソッドは`IncludedNestedStack`インスタンスを返します。このインスタンスから、例に示すように、 `stack`プロパティを介してインスタンスにアクセスできます AWS CDK `NestedStack`。経由で元の AWS CloudFormation テンプレートオブジェクトにアクセスすることもできます。`includedTemplate`そこからリソースやその他の AWS CloudFormation 要素をロードできます。

Systems Manager パラメータストアから値を取得する

は、 AWS Systems Manager Parameter Store 属性の値を取得 AWS Cloud Development Kit (AWS CDK) できます。合成中、 はデプロイ AWS CloudFormation 中に によって解決されるトークン AWS CDK を生成します。

は、プレーン値とセキュア値の両方の取得 AWS CDK をサポートします。どちらの種類の値でも、特定のバージョンをリクエストできます。プレーンな値の場合は、最新バージョンを取得するリクエストからバージョンを省略できます。安全な値には、Secure 属性の値をリクエストするときにバージョンを指定する必要があります。

> **Note**
>
> このトピックでは、 AWS Systems Manager パラメータストアから属性を読み取る方法を示します。からシークレットを読み取ることもできます AWS Secrets Manager （ 「」を参照から値を取得する AWS Secrets Manager ）。

トピック

- デプロイ時に Systems Manager 値を読み取る
- 合成時に Systems Manager 値を読み取る
- Systems Manager に値を書き込む

デプロイ時に Systems Manager 値を読み取る

Systems Manager パラメータストアから値を読み取るには、 valueForStringパラメータと valueForSecureStringParameterメソッドを使用します。目的の属性がプレーン文字列か安全な文字列値かに基づいて、メソッドを選択します。これらのメソッドは、実際の値ではなくトークン を返します。この値は、デプロイ AWS CloudFormation 中に によって解決されます。以下に例を示します。

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

Python

```
import aws_cdk.aws_ssm as ssm

# Get latest version or specified version of plain string attribute
```

```
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name")
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name", 1)

# Get specified version of secure string attribute
secure_string_token = ssm.StringParameter.value_for_secure_string_parameter(
    self, "my-secure-parameter-name", 1)   # must specify version
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

//Get latest version or specified version of plain string attribute
String latestStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name");       // latest version
String versionOfStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name", 1);    // version 1

//Get specified version of secure string attribute
String secureStringToken = StringParameter.valueForSecureStringParameter(
            this, "my-secure-parameter-name", 1);   // must specify version
```

C#

```
using Amazon.CDK.AWS.SSM;

// Get latest version or specified version of plain string attribute
var latestStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name");      // latest version
var versionOfStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name", 1);   // version 1

// Get specified version of secure string attribute
var secureStringToken = StringParameter.ValueForSecureStringParameter(
    this, "my-secure-parameter-name", 1);   // must specify version
```

現在、この機能をサポートしているAWS サービスは限られています。

合成時に Systems Manager 値を読み取る

合成時にパラメータを指定すると便利です。これにより、 AWS CloudFormation テンプレートはデプロイ中に値を解決するのではなく、常に同じ値を使用します。

合成時に Systems Manager パラメータストアから値を読み取るには、 valueFromLookupメソッド (Python:) を使用しますvalue_from_lookup。このメソッドは、 パラメータの実際の値をthe section called “Context”値として返します。値がコマンドラインにキャッシュcdk.jsonされていないか、渡されていない場合は、現在の AWS アカウントから取得されます。このため、スタックは明示的な AWS 環境情報で合成する必要があります。

以下に例を示します。

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

Python

```
import aws_cdk.aws_ssm as ssm

string_value = ssm.StringParameter.value_from_lookup(self, "my-plain-parameter-
name")
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

String stringValue = StringParameter.valueFromLookup(this, "my-plain-parameter-
name");
```

C#

```
using Amazon.CDK.AWS.SSM;

var stringValue = StringParameter.ValueFromLookup(this, "my-plain-parameter-name");
```

取得できるのは、プレーンな Systems Manager 文字列のみです。安全な文字列は取得できません。常に最新バージョンが返されます。特定のバージョンをリクエストすることはできません。

> **Important**
>
> 取得された値は、合成された AWS CloudFormation テンプレートになります。 AWS CloudFormation テンプレートにアクセスできるユーザーとその価値の種類によっては、セキュリティ上のリスクが生じる可能性があります。通常、プライベートにしておくパスワード、キー、またはその他の値にはこの機能を使用しないでください。

Systems Manager に値を書き込む

AWS CLI、、または AWS SDK を使用して AWS Management Console、Systems Manager のパラメータ値を設定できます。次の例では、ssm put-parameter CLI コマンドを使用します。

```
aws ssm put-parameter --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --name "secure-parameter-name" --type "SecureString" --value
 "secure-parameter-value"
```

既に存在する SSM 値を更新する場合は、 `--overwrite`オプションも含めます。

```
aws ssm put-parameter --overwrite --name "parameter-name" --type "String" --value
 "parameter-value"
aws ssm put-parameter --overwrite --name "secure-parameter-name" --type "SecureString"
 --value "secure-parameter-value"
```

から値を取得する AWS Secrets Manager

AWS CDK アプリ AWS Secrets Manager で の値を使用するには、 fromSecretAttributes() メソッドを使用します。これは、Secrets Manager から取得され、 AWS CloudFormation デプロイ時に使用される値を表します。以下に例を示します。

TypeScript

```
import * as sm from "aws-cdk-lib/aws-secretsmanager";

export class SecretsManagerStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
```

JavaScript

```
const sm = require("aws-cdk-lib/aws-secretsmanager");

class SecretsManagerStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
    });
  }
}

module.exports = { SecretsManagerStack }
```

Python

```
import aws_cdk.aws_secretsmanager as sm

class SecretsManagerStack(cdk.Stack):
```

```
    def __init__(self, scope: cdk.App, id: str, **kwargs):
      super().__init__(scope, name, **kwargs)

      secret = sm.Secret.from_secret_attributes(self, "ImportedSecret",
          secret_complete_arn="arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>",
          # If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
          # encryption_key=....
      )
```

Java

```
import software.amazon.awscdk.services.secretsmanager.Secret;
import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

public class SecretsManagerStack extends Stack {
    public SecretsManagerStack(App scope, String id) {
        this(scope, id, null);
    }

    public SecretsManagerStack(App scope, String id, StackProps props) {
        super(scope, id, props);

        Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret",
 SecretAttributes.builder()
            .secretCompleteArn("arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>")
             // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
             // .encryptionKey(...)
             .build());
    }
}
```

C#

```
using Amazon.CDK.AWS.SecretsManager;

public class SecretsManagerStack : Stack
{
    public SecretsManagerStack(App scope, string id, StackProps props) : base(scope,
 id, props) {
```

```
        var secret = Secret.FromSecretAttributes(this, "ImportedSecret", new
 SecretAttributes {
            SecretCompleteArn = "arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>"
            // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
            // encryptionKey = ...,
        });
    }
```

Tip

AWS CLI create-secret CLI コマンドを使用して、テスト時などにコマンドラインからシークレットを作成します。

```
aws secretsmanager create-secret --name ImportedSecret --secret-string
 mygroovybucket
```

このコマンドは、前の例で使用できる ARN を返します。

Secret インスタンスを作成したら、インスタンスの secretValue 属性からシークレットの値を取得できます。値は、特殊なタイプの である`SecretValue`インスタンスで表されますthe section called “トークン”。トークンであるため、解決後にのみ意味があります。CDK アプリは実際の値にアクセスする必要はありません。代わりに、アプリケーションは`SecretValue`インスタンス (またはその文字列または数値表現) を CDK メソッドが値を必要とする任意の に渡すことができます。

CloudWatch アラームを設定する

aws-cloudwatch パッケージを使用して、 CloudWatch メトリクスに Amazon CloudWatch アラームを設定します。定義済みのメトリクスを使用するか、独自のメトリクスを作成できます。

トピック

- 既存のメトリクスの使用
- 独自のメトリクスの作成
- アラームの作成

既存のメトリクスの使用

多くの AWS Construct Library モジュールでは、メトリクスを含むオブジェクトのインスタンスの便利なメソッドにメトリクスの名前を渡すことで、既存のメトリクスにアラームを設定できます。例えば、Amazon SQS キューの場合、キューの metric() メソッドApproximateNumberOfMessagesVisibleから メトリクスを取得できます。 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_sqs.Queue.html#metricmetricname-props

TypeScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

JavaScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

Python

```
metric = queue.metric("ApproximateNumberOfMessagesVisible")
```

Java

```
Metric metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

C#

```
var metric = queue.Metric("ApproximateNumberOfMessagesVisible");
```

独自のメトリクスの作成

次のように独自のメトリクスを作成します。ここで、名前空間の値は Amazon SQS キューの AWS/Amazon SQSのような値である必要があります。また、メトリクスの名前とディメンションを指定する必要があります。

TypeScript

```
const metric = new cloudwatch.Metric({
```

```
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

JavaScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

Python

```
metric = cloudwatch.Metric(
    namespace="MyNamespace",
    metric_name="MyMetric",
    dimensionsMap=dict(MyDimension="MyDimensionValue")
)
```

Java

```
Metric metric = Metric.Builder.create()
        .namespace("MyNamespace")
        .metricName("MyMetric")
        .dimensionsMap(java.util.Map.of(    // Java 9 or later
            "MyDimension", "MyDimensionValue"))
        .build();
```

C#

```
var metric = new Metric(this, "Metric", new MetricProps
{
    Namespace = "MyNamespace",
    MetricName = "MyMetric",
    Dimensions = new Dictionary<string, object>
    {
        { "MyDimension", "MyDimensionValue" }
    }
});
```

アラームの作成

既存のメトリクスまたは定義したメトリクスを取得したら、アラームを作成できます。この例では、直近の 3 つの評価期間のうち 2 つに 100 を超えるメトリクスがある場合にアラームが発生します。`comparisonOperator` プロパティを使用して、アラームの より小さいなどの比較を使用できます。Greater-than-or-equal-to が AWS CDK デフォルトであるため、指定する必要はありません。

TypeScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2
});
```

Python

```
alarm = cloudwatch.Alarm(self, "Alarm",
    metric=metric,
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
import software.amazon.awscdk.services.cloudwatch.Alarm;
import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
        .metric(metric)
        .threshold(100)
```

```
        .evaluationPeriods(3)
        .datapointsToAlarm(2).build();
```

C#

```
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
    Metric = metric,
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

アラームを作成する別の方法は、メトリクスの createAlarm () メソッドを使用することです。これは、基本的にAlarmコンストラクタと同じプロパティを受け取ります。すでにわかっているため、メトリクスを渡す必要はありません。

TypeScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

Python

```
metric.create_alarm(self, "Alarm",
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
metric.createAlarm(this, "Alarm", new CreateAlarmOptions.Builder()
        .threshold(100)
        .evaluationPeriods(3)
        .datapointsToAlarm(2)
        .build());
```

C#

```
metric.CreateAlarm(this, "Alarm", new CreateAlarmOptions
{
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

コンテキスト変数の値を保存および取得する

cdk.json ファイルの または AWS Cloud Development Kit (AWS CDK) CLIを使用してコンテキスト変数を指定できます。次に、 TryGetContextメソッドを使用して値を取得します。

トピック

- コンテキスト変数を指定する
- コンテキスト変数の値を取得する

コンテキスト変数を指定する

コンテキスト変数は、 AWS CDK CLI コマンドの一部として、または で指定できますcdk.json。

コマンドラインコンテキスト変数を作成するには、次の例に示すように、--context (-c) オプションを使用します。

```
cdk synth -c bucket_name=mygroovybucket
```

cdk.json ファイルに同じコンテキスト変数と値を指定するには、次のコードを使用します。

```
{
  "context": {
```

```
    "bucket_name": "myotherbucket"
  }
}
```

ファイルと cdk.json ファイルの両方を使用してコンテキスト変数を指定すると AWS CDK CLI、値が AWS CDK CLI優先されます。

コンテキスト変数の値を取得する

アプリでコンテキスト変数の値を取得するには、コンストラクトのコンテキストで TryGetContextメソッドを使用します。(つまり、 または Python thisselfでは、 はいくつかのコンストラクトのインスタンスです)。

この例では、 bucket_nameコンテキスト変数の値を取得します。リクエストされた値が定義されていない場合、 は例外を発生させるのではなく undefined (NonePython では 、Java null および C# では 、Go nilでは) TryGetContextを返します。

TypeScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

JavaScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

Python

```
bucket_name = self.node.try_get_context("bucket_name")
```

Java

```
String bucketName = (String)this.getNode().tryGetContext("bucket_name");
```

C#

```
var bucketName = this.Node.TryGetContext("bucket_name");
```

コンストラクトのコンテキスト外では、次のようにアプリオブジェクトからコンテキスト変数にアクセスできます。

TypeScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name')
```

JavaScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

Python

```
app = cdk.App()
bucket_name = app.node.try_get_context("bucket_name")
```

Java

```
App app = App();
String bucketName = (String)app.getNode().tryGetContext("bucket_name");
```

C#

```
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

コンテキスト変数の使用の詳細については、「」を参照してくださいthe section called “Context”。

AWS CloudFormation パブリックレジストリからのリソースの使用

AWS CloudFormation Public Registry では、 で使用できるリソース、モジュール、フックなど、パブリックとプライベートの両方の拡張機能を管理できます AWS アカウント。`CfnResource` コンストラクトを使用して、 AWS Cloud Development Kit (AWS CDK) アプリケーションでパブリックリソース拡張を使用できます。

AWS CloudFormation パブリックレジストリの詳細については、「 ユーザーガイド」の「 AWS CloudFormation レジストリの使用AWS CloudFormation 」を参照してください。

によって公開されるすべてのパブリック拡張 AWS は、すべてのリージョンのすべてのアカウントで利用でき、ユーザー側で操作する必要はありません。ただし、使用する各サードパーティ拡張機能は、使用する各アカウントとリージョンでアクティブ化する必要があります。

> **Note**
>
> サードパーティーのリソースタイプ AWS CloudFormation で を使用すると、料金が発生します。料金は、1 か月あたりに実行するハンドラーオペレーションの数と、ハンドラーオペレーションの期間に基づきます。詳細については、「 のCloudFormation 料金」を参照してください。

パブリック拡張の詳細については、「 ユーザーガイド」の「 でのパブリック拡張の使用 CloudFormation」を参照してください。 AWS CloudFormation

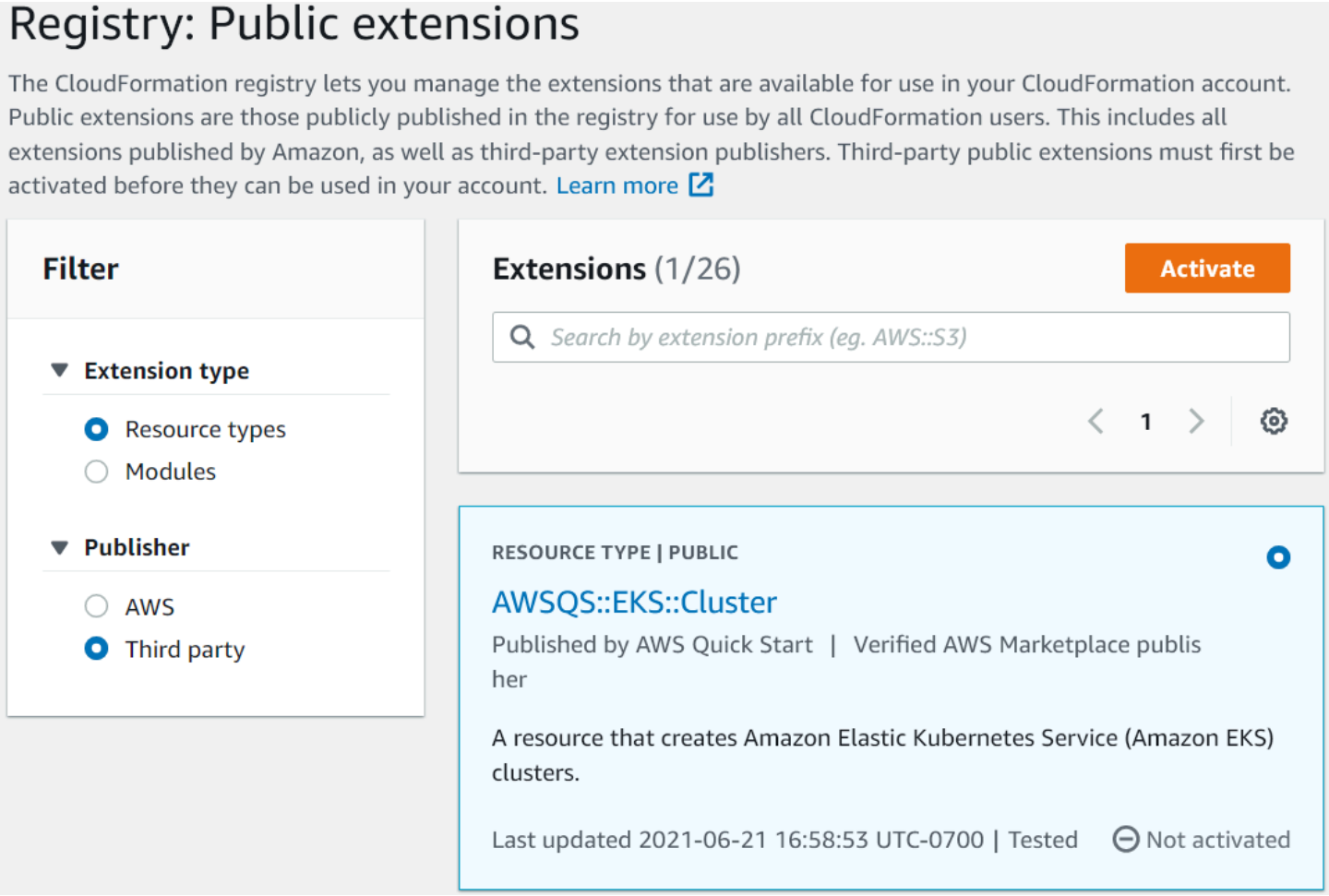
トピック

- アカウントとリージョンでのサードパーティーリソースのアクティブ化
- AWS CloudFormation パブリックレジストリから CDK アプリへのリソースの追加

アカウントとリージョンでのサードパーティーリソースのアクティブ化

によって公開された拡張機能はアクティベーションを必要とし AWS ません。すべての アカウントとリージョンで常に利用できます。サードパーティーの拡張機能は、、 AWS Management Console、 AWS Command Line Interfaceまたは特別な AWS CloudFormation リソースをデプロイすることでアクティブ化できます。

を使用してサードパーティーの拡張機能をアクティブ化 AWS Management Console したり、使用可能なリソースを確認したりするには



1. 拡張機能を使用する AWS アカウントにサインインし、使用するリージョンに切り替えます。
2. サービスメニューを使用して CloudFormation コンソールに移動します。
3. ナビゲーションバーの「パブリック拡張」を選択し、パブリッシャー の下にある「サードパーティ」ラジオボタンを有効にします。利用可能なサードパーティーのパブリック拡張のリストが表示されます。(によって公開されたパブリック拡張のリストAWSを表示することもできますがAWS、アクティブ化する必要はありません)。
4. リストを参照して、アクティブ化する拡張機能を見つけます。または、それを検索し、拡張のカードの右上隅にあるラジオボタンを有効にします。
5. リストの上部にあるアクティブ化ボタンを選択すると、選択した拡張機能がアクティブ化されます。拡張機能のアクティブ化ページが表示されます。
6. アクティブ化 ページで、拡張機能のデフォルト名を上書きし、実行ロールとログ記録設定を指定できます。新しいバージョンがリリースされたときに拡張機能を自動的に更新するかどうか

を選択することもできます。これらのオプションを希望どおりに設定したら、ページの下部にある拡張を有効にするを選択します。

を使用してサードパーティー拡張機能をアクティブ化するには AWS CLI

- `activate-type`コマンドを実行します。示されている場合は、使用するカスタムタイプの ARN を置き換えます。

  以下に例を示します。

  ```
  aws cloudformation activate-type --public-type-arn public_extension_ARN --auto-update-activated
  ```

CloudFormation または CDK を使用してサードパーティー拡張機能をアクティブ化するには

- タイプのリソースをデプロイ`AWS::CloudFormation::TypeActivation`し、次のプロパティを指定します。

  a. `TypeName` - など、タイプの名前`AWSQS::EKS::Cluster`。

  b. `MajorVersion` - 必要な拡張機能のメジャーバージョン番号。最新バージョンが必要な場合は を省略します。

  c. `AutoUpdate` - パブリッシャーによって新しいマイナーバージョンがリリースされたときに、この拡張機能を自動的に更新するかどうか。(メジャーバージョンの更新では、`MajorVersion`プロパティを明示的に変更する必要があります)。

  d. `ExecutionRoleArn` - この拡張機能を実行する IAM ロールの ARN。

  e. `LoggingConfig` - 拡張機能のログ記録構成。

  `TypeActivation` リソースは、 `CfnResource`コンストラクトを使用して CDK によってデプロイできます。これは、次のセクションで実際の拡張機能について示します。

AWS CloudFormation パブリックレジストリから CDK アプリへのリソースの追加

CfnResource コンストラクトを使用して、 AWS CloudFormation パブリックレジストリのリソースをアプリケーションに含めます。このコンストラクトは CDK の aws-cdk-libモジュールにあります。

例えば、 AWS CDK アプリケーションMY::S5::UltimateBucketで使用する という名前のパブリックリソースがあるとします。このリソースは、バケット名という 1 つのプロパティを受け取ります。対応するCfnResourceインスタンス化は次のようになります。

TypeScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

JavaScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

Python

```
ubucket = CfnResource(self, "MyUltimateBucket",
            type="MY::S5::UltimateBucket::MODULE",
            properties=dict(
                BucketName="UltimateBucket"))
```

Java

```
CfnResource.Builder.create(this, "MyUltimateBucket")
 .type("MY::S5::UltimateBucket::MODULE")
```

```
 .properties(java.util.Map.of(    // Map.of requires Java 9+
     "BucketName", "UltimateBucket"))
 .build();
```

C#

```
new CfnResource(this, "MyUltimateBucket", new CfnResourceProps
{
    Type = "MY::S5::UltimateBucket::MODULE",
    Properties = new Dictionary<string, object>
    {
        ["BucketName"] = "UltimateBucket"
    }
});
```

AWS CDK アプリケーションのデプロイ

AWS Cloud Development Kit (AWS CDK) アプリケーションをデプロイします。

トピック

AWS CDK 合成時のポリシー検証

トピック

合成時のポリシー検証

お客様またはお客様の組織が、 AWS CloudFormation Guardや OPA などのポリシー検証ツールを使用して AWS CloudFormation テンプレートの制約を定義する場合は、合成 AWS CDK 時にそれらを と統合できます。適切なポリシー検証プラグインを使用すると、合成後すぐに、生成された AWS CloudFormation テンプレートをポリシーと照合して AWS CDK アプリケーションをチェックできます。違反がある場合、合成は失敗し、レポートがコンソールに出力されます。

合成 AWS CDK 時に によって実行される検証では、デプロイライフサイクルのある時点でコントロールが検証されますが、合成以外で発生するアクションには影響しません。例としては、コンソールで直接実行されたアクションや、サービス APIsを介して実行されたアクションなどがあります。合成後の AWS CloudFormation テンプレートの変更には耐性がありません。同じルールセットをより権限的に検証する他のメカニズムには、AWS CloudFormation フックや など、個別に設定する必要がありますAWS Config。それでも、 が開発中にルールセット AWS CDK を評価する能力は、検出速度と開発者の生産性を向上させるため、引き続き役立ちます。

AWS CDK ポリシー検証の目的は、開発中に必要なセットアップ量を最小限に抑え、できるだけ簡単にすることです。

> **Note**
>
> この機能は実験的と見なされ、プラグイン API と検証レポートの形式の両方が将来変更される可能性があります。

トピック

アプリケーションデベロッパー向け

アプリケーションで 1 つ以上の検証プラグインを使用するには、 の policyValidationBeta1プロパティを使用しますStage。

```
import { CfnGuardValidator } from '@cdklabs/cdk-validator-cfnguard';
const app = new App({
  policyValidationBeta1: [
    new CfnGuardValidator()
  ],
});
// only apply to a particular stage
const prodStage = new Stage(app, 'ProdStage', {
  policyValidationBeta1: [...],
});
```

合成直後に、この方法で登録されたすべてのプラグインが呼び出され、定義したスコープで生成されたすべてのテンプレートが検証されます。特に、Appオブジェクトにテンプレートを登録すると、すべてのテンプレートが検証の対象となります。

> **Warning**
>
> クラウドアセンブリを変更する以外にも、プラグインは AWS CDK アプリケーションができることをすべて実行できます。ファイルシステムからデータを読み取ったり、ネットワークにアクセスしたりできます。プラグインのコンシューマーとして、使用が安全であることを確認するのはお客様の責任です。

AWS CloudFormation Guard プラグイン

CfnGuardValidator プラグインを使用すると、 AWS CloudFormation Guardを使用してポリシーの検証を実行できます。CfnGuardValidator プラグインには、プロAWS Control Tower アクティブコントロールの一部が組み込まれています。現在のルールのセットは、プロジェクトドキュメント「」に記載されています。「」で説明したように合成時のポリシー検証、組織はフック を使用してより強力な検証方法を設定することをお勧めします。 AWS CloudFormation

顧客の場合AWS Control Tower、これらの同じプロアクティブコントロールを組織全体にデプロイできます。 AWS Control Tower 環境で AWS Control Tower プロアクティブコントロールを有効にすると、コントロールは 経由でデプロイされた非準拠リソースのデプロイを停止できます AWS CloudFormation。マネージドプロアクティブコントロールとその仕組みの詳細については、AWS Control Tower 「」のドキュメントを参照してください。

これらの AWS CDK バンドルされたコントロールとマネージド AWS Control Tower プロアクティブコントロールは、一緒に使用するのが最適です。このシナリオでは、 AWS Control Tower クラウド環境でアクティブなプロアクティブコントロールと同じプロアクティブコントロールを使用して、この検証プラグインを設定できます。その後、cdk synthローカルで実行することで、 AWS CDK アプリケーションが AWS Control Tower コントロールに合格することをすばやく確認できます。

検証レポート

AWS CDK アプリを合成すると、検証プラグインが呼び出され、結果が出力されます。レポートの例を以下に示します。

```
Validation Report (CfnGuardValidator)
-------------------------------------
(Summary)
#####################################
# Status    # failure               #
#####################################
# Plugin    # CfnGuardValidator     #
#####################################
(Violations)
Ensure S3 Buckets are encrypted with a KMS CMK (1 occurrences)
Severity: medium
  Occurrences:

    - Construct Path: MyStack/MyCustomL3Construct/Bucket
    - Stack Template Path: ./cdk.out/MyStack.template.json
```

```
    - Creation Stack:
        ###  MyStack (MyStack)
             # Library: aws-cdk-lib.Stack
             # Library Version: 2.50.0
             # Location: Object.<anonymous> (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:25:20)
             ###  MyCustomL3Construct (MyStack/MyCustomL3Construct)
                  # Library: N/A - (Local Construct)
                  # Library Version: N/A
                  # Location: new MyStack (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:15:20)
                  ###  Bucket (MyStack/MyCustomL3Construct/Bucket)
                       # Library: aws-cdk-lib/aws-s3.Bucket
                       # Library Version: 2.50.0
                       # Location: new MyCustomL3Construct (/home/johndoe/tmp/cdk-tmp-
app/src/main.ts:9:20)
    - Resource Name: my-bucket
    - Locations:
      > BucketEncryption/ServerSideEncryptionConfiguration/0/
ServerSideEncryptionByDefault/SSEAlgorithm
  Recommendation: Missing value for key `SSEAlgorithm` - must specify `aws:kms`
  How to fix:
    > Add to construct properties for `cdk-app/MyStack/Bucket`
      `encryption: BucketEncryption.KMS`

Validation failed. See above reports for details
```

デフォルトでは、レポートは人間が読める形式で出力されます。JSON 形式のレポートが必要な場合は、 CLI @aws-cdk/core:validationReportJson経由で を使用して有効にするか、アプリケーションに直接渡します。

```
const app = new App({
  context: { '@aws-cdk/core:validationReportJson': true },
});
```

または、プロジェクトディレクトリ内の cdk.jsonまたは cdk.context.json ファイルを使用して、このコンテキストキーと値のペアを設定することもできます (「」を参照ランタイムのコンテキスト)。

JSON 形式を選択すると、 AWS CDK はクラウドアセンブリディレクトリの というファイルにポリシー検証レポートを出力policy-validation-report.jsonします。人間が読めるデフォルトの形式の場合、レポートは標準出力に出力されます。

プラグイン作成者の場合

プラグイン

AWS CDK コアフレームワークは、プラグインの登録と呼び出し、フォーマットされた検証レポートの表示を担当します。プラグインの責任は、 AWS CDK フレームワークとポリシー検証ツールの間の翻訳レイヤーとして機能することです。プラグインは、 でサポートされている任意の言語で作成できます AWS CDK。複数の言語で消費される可能性のあるプラグインを作成する場合は、JSII を使用して各 AWS CDK 言語でプラグインを公開TypeScriptできるように、 でプラグインを作成することをお勧めします。

プラグインの作成

AWS CDK コアモジュールとポリシーツール間の通信プロトコルは、`IPolicyValidationPluginBeta1`インターフェイスによって定義されます。新しいプラグインを作成するには、このインターフェイスを実装するクラスを作成する必要があります。実装する必要があるのは、プラグイン名 (`name`プロパティを上書きする) と `validate()`メソッドの 2 つです。

フレームワークは を呼び出し`validate()`、 `IValidationContextBeta1` オブジェクトを渡します。検証するテンプレートの場所は、 によって指定されます`templatePaths`。プラグインは のインスタンスを返す必要があります`ValidationPluginReportBeta1`。このオブジェクトは、合成の最後にユーザーが受け取るレポートを表します。

```
validate(context: IPolicyValidationContextBeta1): PolicyValidationReportBeta1 {
  // First read the templates using context.templatePaths...
  // ...then perform the validation, and then compose and return the report.
  // Using hard-coded values here for better clarity:
  return {
    success: false,
    violations: [{
      ruleName: 'CKV_AWS_117',
      description: 'Ensure that AWS Lambda function is configured inside a VPC',
      fix: 'https://docs.bridgecrew.io/docs/ensure-that-aws-lambda-function-is-
configured-inside-a-vpc-1',
      violatingResources: [{
        resourceName: 'MyFunction3BAA72D1',
        templatePath: '/home/johndoe/myapp/cdk.out/MyService.template.json',
        locations: 'Properties/VpcConfig',
      }],
    }],
  };
```

```
}
```

プラグインはクラウドアセンブリ内の内容を変更できないことに注意してください。これを試みると、合成が失敗します。

プラグインが外部ツールに依存している場合は、一部のデベロッパーはワークステーションにそのツールがまだインストールされていない可能性があることに注意してください。摩擦を最小限に抑えるため、プロセス全体を自動化するために、プラグインパッケージと一緒にいくつかのインストールスクリプトを提供することを強くお勧めします。まだの場合は、パッケージのインストールの一部としてそのスクリプトを実行してください。例えばnpm、 では、 `package.json` ファイル内のpostinstallスクリプトに追加できます。

例外の処理

組織に免除を処理するメカニズムがある場合は、検証プラグインの一部として実装できます。

免除メカニズムの例を説明するシナリオ：

- 組織には、特定のシナリオを除き、パブリック Amazon S3 バケットは許可されないというルールがあります。
- デベロッパーが、これらのシナリオのいずれかに該当する Amazon S3 バケットを作成し、免除をリクエストする (チケットを作成するなど) 。
- セキュリティツールは、免除を登録する内部システムからの読み取り方法を知っている

このシナリオでは、開発者は内部システムで例外をリクエストし、その例外を「登録」する方法が必要になります。ガードプラグインの例に を追加すると、内部チケット発行システムで一致する免除がある違反を除外して免除を処理するプラグインを作成できます。

実装例については、既存のプラグインを参照してください。

- @cdklabs/cdk-validator-cfnguard

CDK Pipelines を使用した継続的インテグレーションとデリバリー (CI/CD)

構築ライブラリの CDK Pipelines AWS モジュールを使用して、 AWS CDK アプリケーションの継続的な配信を設定します。CDK アプリケーションのソースコードを AWS CodeCommit、GitHub、ま

たは にコミットすると AWS CodeStar、CDK Pipelines は新しいバージョンを自動的に構築、テスト、デプロイできます。

CDK Pipelines は自動更新されています。アプリケーションステージまたはスタックを追加すると、パイプラインは自動的に再設定され、それらの新しいステージまたはスタックがデプロイされます。

> **Note**
>
> CDK Pipelines は 2 つの APIsをサポートしています。1 つは CDK Pipelines デベロッパープレビューで利用可能になった元の API です。もう 1 つは、プレビューフェーズで受け取った CDK 顧客からのフィードバックを組み込む最新の API です。このトピックの例では、最新の API を使用しています。サポートされている 2 つの APIs、aws-cdk リポジトリの「CDK Pipelines original API」を参照してください。 GitHub

トピック

AWS 環境のブートストラップ

CDK Pipelines を使用する前に、スタックをデプロイする AWS 環境をブートストラップする必要があります。

CDK パイプラインには、少なくとも 2 つの環境が含まれます。最初の環境は、パイプラインがプロビジョニングされる場所です。2 番目の環境は、アプリケーションのスタックまたはステージを にデプロイする場所です (ステージは関連するスタックのグループです)。これらの環境は同じにすることができますが、ベストプラクティスとして、異なる環境でステージを分離することをお勧めします。

> **Note**
>
> ブートストラップによって作成されたリソースの種類と、ブートストラップスタックをカスタマイズする方法the section called “ブートストラッピング”の詳細については、「」を参照してください。

CDK Pipelines を使用した継続的なデプロイでは、CDK Toolkit スタックに以下を含める必要があります。

- 1 つの Amazon Simple Storage Service (Amazon S3) バケット
- Amazon ECR リポジトリ。
- パイプラインのさまざまな部分に、必要なアクセス許可を付与する IAM ロール。

CDK Toolkit は、既存のブートストラップスタックをアップグレードするか、必要に応じて新しいブートストラップスタックを作成します。

AWS CDK パイプラインをプロビジョニングできる環境をブートストラップするには、次の例cdk bootstrapに示すように を呼び出します。npx コマンドを使用して AWS CDK Toolkit を呼び出すと、必要に応じて一時的にインストールされます。また、現在のプロジェクトにインストールされている Toolkit が存在する場合は、そのバージョンも使用します。

`--cloudformation-execution-policies` は、将来の CDK Pipelines デプロイを実行するポリシーの ARN を指定します。デフォルトのAdministratorAccessポリシーにより、パイプラインはすべてのタイプの AWS リソースをデプロイできます。このポリシーを使用する場合は、 AWS CDK アプリを構成するすべてのコードと依存関係を信頼してください。

ほとんどの組織では、自動化によってデプロイできるリソースの種類をより厳密に制御することを義務付けています。組織内の適切な部門に問い合わせて、パイプラインで使用するポリシーを決定します。

デフォルトの AWS プロファイルに必要な認証設定 と が含まれている場合は、 `--profile`オプションを省略できます AWS リージョン。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

パイプラインによって AWS CDK アプリケーションがデプロイされる追加の環境をブートストラップするには、代わりに次のコマンドを使用します。--trust オプションは、この環境に AWS CDK アプリケーションをデプロイするアクセス許可を持つ他のアカウントを示します。このオプションでは、パイプラインの AWS アカウント ID を指定します。

デフォルトの AWS プロファイルに必要な認証設定 と が含まれている場合は、 --profileオプションを省略することもできます AWS リージョン。

macOS/Linux

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 \
    --trust PIPELINE-ACCOUNT-NUMBER
```

Windows

```
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 ^
    --trust PIPELINE-ACCOUNT-NUMBER
```

> **Tip**
>
> 管理認証情報は、ブートストラップと初期パイプラインのプロビジョニングにのみ使用します。その後、ローカルマシンではなくパイプライン自体を使用して変更をデプロイします。

レガシーブートストラップ環境をアップグレードする場合、新しいバケットが作成されると、以前の Amazon S3 バケットは孤立します。Amazon S3 コンソールを使用して手動で削除します。

ブートストラップスタックの削除からの保護

ブートストラップスタックが削除されると、CDK デプロイをサポートするために環境に最初にプロビジョニングされた AWS リソースも削除されます。これにより、パイプラインは動作しなくなります。この場合、復旧のための一般的な解決策はありません。

環境がブートストラップされたら、環境のブートストラップスタックを削除して再作成しないでください。代わりに、 `cdk bootstrap` コマンドを再度実行して、ブートストラップスタックを新しいバージョンに更新してみてください。

ブートストラップスタックが誤って削除されないように、終了保護を有効にするには、 `cdk bootstrap` コマンドで `--termination-protection`オプションを指定することをお勧めします。新規または既存のブートストラップスタックで終了保護を有効にできます。このオプションの詳細については、「」を参照してください`--termination-protection`。

終了保護を有効にした後、 AWS CLI または CloudFormation コンソールを使用して検証できます。

終了保護を有効にするには

1. 次のコマンドを実行して、新規または既存のブートストラップスタックで終了保護を有効にします。

   ```
   $ cdk bootstrap --termination-protection
   ```

2. AWS CLI または CloudFormation コンソールを使用して確認します。以下に示しているのは、AWS CLIを使用した例です。ブートストラップスタック名を変更した場合は、 をスタック名`CDKToolkit`に置き換えます。

   ```
   $ aws cloudformation describe-stacks --stack-name CDKToolkit --query
    "Stacks[0].EnableTerminationProtection"
   true
   ```

プロジェクトの初期化

新しい空のGitHubプロジェクトを作成し、 `my-pipeline` ディレクトリのワークステーションにクローンを作成します。（ このトピックのコード例では、 を使用しています GitHub。 または を使用 AWS CodeStar することもできます ） AWS CodeCommit。

```
git clone GITHUB-CLONE-URL my-pipeline
```

```
cd my-pipeline
```

> **Note**
>
> アプリのメインディレクトリmy-pipelineには、 以外の名前を使用できます。ただし、その場合は、このトピックの後半でファイルとクラス名を微調整する必要があります。これは、 AWS CDK Toolkit が一部のファイル名とクラス名をメインディレクトリの名前に基づいているためです。

クローンを作成したら、通常どおりプロジェクトを初期化します。

TypeScript

```
$ cdk init app --language typescript
```

JavaScript

```
$ cdk init app --language javascript
```

Python

```
$ cdk init app --language python
```

アプリを作成したら、次の 2 つのコマンドも入力します。これにより、アプリケーションの Python 仮想環境がアクティブ化され、 AWS CDK コア依存関係がインストールされます。

```
$ source .venv/bin/activate # On Windows, run `.\venv\Scripts\activate` instead
$ python -m pip install -r requirements.txt
```

Java

```
$ cdk init app --language java
```

IDE を使用している場合は、プロジェクトを開くかインポートできるようになりました。例えば、Eclipse で、ファイル > インポート > Maven > 既存の Maven プロジェクト を選択します。プロジェクト設定が Java 8 (1.8) を使用するように設定されていることを確認します。

C#

```
$ cdk init app --language csharp
```

Visual Studio を使用している場合は、 src ディレクトリでソリューションファイルを開きます。

Go

```
$ cdk init app --language go
```

アプリを作成したら、次のコマンドを入力して、アプリに必要な AWS コンストラクトライブラリモジュールをインストールします。

```
$ go get
```

> **Important**
>
> cdk.json および cdk.context.jsonファイルは必ず出典管理にコミットしてください。コンテキスト情報 (AWS アカウントから取得した機能フラグやキャッシュされた値など) は、プロジェクトの状態の一部です。別の環境では値が異なる場合があります。これにより、結果に予期しない変更が生じる可能性があります。詳細については、「the section called “Context”」を参照してください。

パイプラインを定義する

CDK Pipelines アプリケーションには、少なくとも 2 つのスタックが含まれます。1 つはパイプライン自体を表し、もう 1 つはパイプラインを通じてデプロイされたアプリケーションを表すスタックです。スタックはステージ にグループ化することもできます。ステージ を使用して、インフラストラクチャスタックのコピーを異なる環境にデプロイできます。今のところ、パイプラインを検討し、後でデプロイするアプリケーションを詳しく説明します。

コンストラクトCodePipelineは、 がデプロイエンジン AWS CodePipeline として使用する CDK パイプラインを表すコンストラクトです。スタックCodePipelineでインスタンス化するときは、パイプラインのソースの場所 (GitHub リポジトリなど) を定義します。また、アプリケーションを構築するためのコマンドも定義します。

例えば、次の例では、ソースが GitHub リポジトリに保存されているパイプラインを定義します。TypeScript CDK アプリケーションのビルドステップも含まれています。指定された場所に GitHub リポジトリに関する情報を入力します。

> **Note**
>
> デフォルトでは、パイプラインは Secrets Manager に保存されている個人用アクセストークンを という名前で GitHub 使用して を認証しますgithub-token。

また、パイプラインスタックのインスタンス化を更新して、 AWS アカウントとリージョンを指定する必要があります。

TypeScript

で lib/my-pipeline-stack.ts (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}
```

で bin/my-pipeline.ts (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
#!/usr/bin/env node
import * as cdk from 'aws-cdk-lib';
```

```
import { MyPipelineStack } from '../lib/my-pipeline-stack';

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

JavaScript

で lib/my-pipeline-stack.js (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}

module.exports = { MyPipelineStack }
```

で bin/my-pipeline.js (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
```

```
const { MyPipelineStack } = require('../lib/my-pipeline-stack');

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

Python

で my-pipeline/my-pipeline-stack.py (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]
                        )
                    )
```

Eclipse app.py:

```
#!/usr/bin/env python3
import aws_cdk as cdk
from my_pipeline.my_pipeline_stack import MyPipelineStack

app = cdk.App()
```

```
MyPipelineStack(app, "MyPipelineStack",
    env=cdk.Environment(account="111111111111", region="eu-west-1")
)

app.synth()
```

Java

で src/main/java/com/myorg/MyPipelineStack.java (プロジェクトフォルダの名前がでない場合、異なる場合がありますmy-pipeline) 。

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
             .pipelineName("MyPipeline")
             .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
             .build();
    }
}
```

で src/main/java/com/myorg/MyPipelineApp.java (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MyPipelineApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyPipelineStack(app, "PipelineStack", StackProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build());

        app.synth();
    }
}
```

C#

で src/MyPipeline/MyPipelineStack.cs (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
using Amazon.CDK;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
```

```
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });
        }
    }
}
```

で src/MyPipeline/Program.cs (プロジェクトフォルダの名前が でない場合、異なる場合がありますmy-pipeline) 。

```
using Amazon.CDK;

namespace MyPipeline
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyPipelineStack(app, "MyPipelineStack", new StackProps
            {
                Env = new Amazon.CDK.Environment {
                    Account = "111111111111", Region = "eu-west-1" }
            });

            app.Synth();
        }
    }
}
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 codebuild "github.com/aws/aws-cdk-go/awscdk/v2/awscodebuild"
 ssm "github.com/aws/aws-cdk-go/awscdk/v2/awsssm"
 pipeline "github.com/aws/aws-cdk-go/awscdk/v2/pipelines"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
 "os"
```

```
)

// my CDK Stack with resources
func NewCdkStack(scope constructs.Construct, id *string, props *awscdk.StackProps)
 awscdk.Stack {
 stack := awscdk.NewStack(scope, id, props)

 // create an example ssm parameter
 _ = ssm.NewStringParameter(stack, jsii.String("ssm-test-param"),
 &ssm.StringParameterProps{
  ParameterName: jsii.String("/testparam"),
  Description:   jsii.String("ssm parameter for demo"),
  StringValue:   jsii.String("my test param"),
 })

 return stack
}

// my CDK Application
func NewCdkApplication(scope constructs.Construct, id *string, props
 *awscdk.StageProps) awscdk.Stage {
 stage := awscdk.NewStage(scope, id, props)

 _ = NewCdkStack(stage, jsii.String("cdk-stack"), &awscdk.StackProps{Env:
 props.Env})

 return stage
}

// my CDK Pipeline
func NewCdkPipeline(scope constructs.Construct, id *string, props
 *awscdk.StackProps) awscdk.Stack {
 stack := awscdk.NewStack(scope, id, props)

 // GitHub repo with owner and repository name
 githubRepo := pipeline.CodePipelineSource_GitHub(jsii.String("owner/repo"),
 jsii.String("main"), &pipeline.GitHubSourceOptions{
  Authentication: awscdk.SecretValue_SecretsManager(jsii.String("my-github-token"),
 nil),
 })

 // self mutating pipeline
 myPipeline := pipeline.NewCodePipeline(stack, jsii.String("cdkPipeline"),
 &pipeline.CodePipelineProps{
```

```
  PipelineName: jsii.String("CdkPipeline"),
  // self mutation true - pipeline changes itself before application deployment
  SelfMutation: jsii.Bool(true),
  CodeBuildDefaults: &pipeline.CodeBuildOptions{
   BuildEnvironment: &codebuild.BuildEnvironment{
    // image version 6.0 recommended for newer go version
    BuildImage: codebuild.LinuxBuildImage_FromCodeBuildImageId(jsii.String("aws/
codebuild/standard:6.0")),
   },
  },
  Synth: pipeline.NewCodeBuildStep(jsii.String("Synth"),
 &pipeline.CodeBuildStepProps{
   Input: githubRepo,
   Commands: &[]*string{
    jsii.String("npm install -g aws-cdk"),
    jsii.String("cdk synth"),
   },
  }),
 })

 // deployment of actual CDK application
 myPipeline.AddStage(NewCdkApplication(stack, jsii.String("MyApplication"),
 &awscdk.StageProps{
  Env: targetAccountEnv(),
 }), &pipeline.AddStageOpts{
  Post: &[]pipeline.Step{
   pipeline.NewCodeBuildStep(jsii.String("Manual Steps"),
 &pipeline.CodeBuildStepProps{
    Commands: &[]*string{
     jsii.String("echo \"My CDK App deployed, manual steps go here ... \""),
    },
   }),
  },
 })

 return stack
}

// main app
func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)
```

```
 // call CDK Pipeline
 NewCdkPipeline(app, jsii.String("CdkPipelineStack"), &awscdk.StackProps{
  Env: pipelineEnv(),
 })

 app.Synth(nil)
}

// env determines the AWS environment (account+region) in which our stack is to
// be deployed. For more information see: https://docs.aws.amazon.com/cdk/latest/
guide/environments.html
func pipelineEnv() *awscdk.Environment {
 return &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
 }
}

func targetAccountEnv() *awscdk.Environment {
 return &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
 }
}
```

パイプラインは 1 回手動でデプロイする必要があります。その後、パイプラインはソースコードリポジトリから最新の状態を維持します。したがって、リポジトリ内のコードがデプロイするコードであることを確認してください。変更を確認して にプッシュし GitHub、以下をデプロイします。

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

> **Tip**
>
> 最初のデプロイが完了したので、ローカル AWS アカウントは管理アクセスを必要としなくなります。これは、アプリケーションへのすべての変更がパイプラインを介してデプロイされるためです。必要なのは、 にプッシュすることだけです GitHub。

アプリケーションステージ

パイプラインに一度に追加できるマルチスタック AWS アプリケーションを定義するには、 のサブクラスを定義しますStage。（ これは CDK Pipelines モジュールの とは異なりCdkStageます。 ）

ステージには、アプリケーションを構成するスタックが含まれます。スタック間に依存関係がある場合、スタックは適切な順序でパイプラインに自動的に追加されます。相互に依存しないスタックは、並行してデプロイされます。を呼び出すことで、スタック間に依存関係を追加できますstack1.addDependency(stack2)。

ステージはデフォルトのenv引数を受け入れ、その中のスタックのデフォルト環境になります。（ スタックには、引き続き独自の環境を指定できます ）。

アプリケーションをパイプラインに追加するには、 のインスタンスaddStage()で を呼び出しますStage。ステージをインスタンス化してパイプラインに複数回追加して、DTAP またはマルチリージョンアプリケーションパイプラインのさまざまなステージを定義できます。

シンプルな Lambda 関数を含むスタックを作成し、そのスタックをステージに配置します。次に、ステージをパイプラインに追加してデプロイできるようにします。

TypeScript

Lambda 関数を含むアプリケーションスタックlib/my-pipeline-lambda-stack.tsを保持する新しい ファイルを作成します。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { Function, InlineCode, Runtime } from 'aws-cdk-lib/aws-lambda';

export class MyLambdaStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}
```

ステージlib/my-pipeline-app-stage.tsを保持する新しい ファイルを作成します。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from "constructs";
import { MyLambdaStack } from './my-pipeline-lambda-stack';

export class MyPipelineAppStage extends cdk.Stage {

    constructor(scope: Construct, id: string, props?: cdk.StageProps) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}
```

を編集lib/my-pipeline-stack.tsしてステージをパイプラインに追加します。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';
import { MyPipelineAppStage } from './my-pipeline-app-stage';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));
  }
}
```

JavaScript

Lambda 関数を含むアプリケーションスタックlib/my-pipeline-lambda-stack.jsを保持する新しい ファイルを作成します。

```
const cdk = require('aws-cdk-lib');
const { Function, InlineCode, Runtime } = require('aws-cdk-lib/aws-lambda');

class MyLambdaStack extends cdk.Stack {
    constructor(scope, id, props) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}

module.exports = { MyLambdaStack }
```

ステージlib/my-pipeline-app-stage.jsを保持する新しい ファイルを作成します。

```
const cdk = require('aws-cdk-lib');
const { MyLambdaStack } = require('./my-pipeline-lambda-stack');

class MyPipelineAppStage extends cdk.Stage {

    constructor(scope, id, props) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}

module.exports = { MyPipelineAppStage };
```

を編集lib/my-pipeline-stack.tsしてステージをパイプラインに追加します。

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');
const { MyPipelineAppStage } = require('./my-pipeline-app-stage');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
```

```
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));

  }
}

module.exports = { MyPipelineStack }
```

Python

Lambda 関数を含むアプリケーションスタックmy_pipeline/my_pipeline_lambda_stack.pyを保持する新しい ファイルを作成します。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.aws_lambda import Function, InlineCode, Runtime

class MyLambdaStack(cdk.Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        Function(self, "LambdaFunction",
            runtime=Runtime.NODEJS_18_X,
            handler="index.handler",
            code=InlineCode("exports.handler = _ => 'Hello, CDK';")
        )
```

ステージmy_pipeline/my_pipeline_app_stage.pyを保持する新しい ファイルを作成します。

```
import aws_cdk as cdk
from constructs import Construct
```

```
from my_pipeline.my_pipeline_lambda_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        lambdaStack = MyLambdaStack(self, "LambdaStack")
```

を編集my_pipeline/my-pipeline-stack.pyしてステージをパイプラインに追加します。

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep
from my_pipeline.my_pipeline_app_stage import MyPipelineAppStage

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]))

        pipeline.add_stage(MyPipelineAppStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))
```

Java

Lambda 関数を含むアプリケーションスタックsrc/main/java/com.myorg/MyPipelineLambdaStack.javaを保持する新しい ファイルを作成します。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

import software.amazon.awscdk.services.lambda.Function;
```

```
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.InlineCode;

public class MyPipelineLambdaStack extends Stack {
    public MyPipelineLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineLambdaStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        Function.Builder.create(this, "LambdaFunction")
          .runtime(Runtime.NODEJS_18_X)
          .handler("index.handler")
          .code(new InlineCode("exports.handler = _ => 'Hello, CDK';"))
          .build();

    }

}
```

ステージsrc/main/java/com.myorg/MyPipelineAppStage.javaを保持する新しい ファイルを作成します。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.Stage;
import software.amazon.awscdk.StageProps;

public class MyPipelineAppStage extends Stage {
    public MyPipelineAppStage(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineAppStage(final Construct scope, final String id, final
 StageProps props) {
        super(scope, id, props);

        Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
    }
```

```
}
```

を編集src/main/java/com.myorg/MyPipelineStack.javaしてステージをパイプラインに追加します。

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.StageProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
            .pipelineName("MyPipeline")
            .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
            .build();

        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build()));
    }
}
```

C#

Lambda 関数を含むアプリケーションスタックsrc/MyPipeline/MyPipelineLambdaStack.csを保持する新しい ファイルを作成します。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.Lambda;

namespace MyPipeline
{
    class MyPipelineLambdaStack : Stack
    {
        public MyPipelineLambdaStack(Construct scope, string id, StackProps
 props=null) : base(scope, id, props)
        {
            new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "index.handler",
                Code = new InlineCode("exports.handler = _ => 'Hello, CDK';")
            });
        }
    }
}
```

ステージsrc/MyPipeline/MyPipelineAppStage.csを保持する新しい ファイルを作成します。

```
using Amazon.CDK;
using Constructs;

namespace MyPipeline
{
    class MyPipelineAppStage : Stage
    {
        public MyPipelineAppStage(Construct scope, string id, StageProps
 props=null) : base(scope, id, props)
        {
            Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
        }
    }
```

```
}
```

を編集src/MyPipeline/MyPipelineStack.csしてステージをパイプラインに追加します。

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });

            pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
            {
                Env = new Environment
                {
                    Account = "111111111111", Region = "eu-west-1"
                }
            }));
        }
    }
}
```

によって追加されたすべてのアプリケーションステージは、対応するパイプラインステージを追加addStage()し、addStage()呼び出しによって返されるStageDeploymentインスタンスで表されます。デプロイ前またはデプロイ後のアクションをステージに追加するには、 addPre() または addPost()メソッドを呼び出します。

TypeScript

```
// import { ManualApprovalStep } from 'aws-cdk-lib/pipelines';

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

    testingStage.addPost(new ManualApprovalStep('approval'));
```

JavaScript

```
// const { ManualApprovalStep } = require('aws-cdk-lib/pipelines');

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

testingStage.addPost(new ManualApprovalStep('approval'));
```

Python

```
# from aws_cdk.pipelines import ManualApprovalStep

testing_stage = pipeline.add_stage(MyPipelineAppStage(self, "testing",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))

testing_stage.add_post(ManualApprovalStep('approval'))
```

Java

```
// import software.amazon.awscdk.pipelines.StageDeployment;
// import software.amazon.awscdk.pipelines.ManualApprovalStep;

StageDeployment testingStage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));
```

```
testingStage.addPost(new ManualApprovalStep("approval"));
```

C#

```
var testingStage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new
 StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

testingStage.AddPost(new ManualApprovalStep("approval"));
```

Wave にステージを追加して、ステージを複数のアカウントやリージョンにhttps://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.Wave.htmlデプロイする場合などに、ステージを並行してデプロイできます。

TypeScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

JavaScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

Python

```
wave = pipeline.add_wave("wave")
wave.add_stage(MyApplicationStage(self, "MyAppEU",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))
wave.add_stage(MyApplicationStage(self, "MyAppUS",
    env=cdk.Environment(account="111111111111", region="us-west-1")))
```

Java

```
// import software.amazon.awscdk.pipelines.Wave;
final Wave wave = pipeline.addWave("wave");
wave.addStage(new MyPipelineAppStage(this, "MyAppEU", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
        .build()));
wave.addStage(new MyPipelineAppStage(this, "MyAppUS", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("us-west-1")
                .build())
        .build()));
```

C#

```
var wave = pipeline.AddWave("wave");
wave.AddStage(new MyPipelineAppStage(this, "MyAppEU", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));
wave.AddStage(new MyPipelineAppStage(this, "MyAppUS", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "us-west-1"
    }
}));
```

デプロイのテスト

CDK パイプラインにステップを追加して、実行しているデプロイを検証できます。例えば、CDK Pipeline ライブラリの を使用して`ShellStep`、次のようなタスクを実行できます。

- Lambda 関数によってバックアップされた、新しくデプロイされた Amazon API Gateway へのアクセスの試行
- AWS CLI コマンドを発行してデプロイされたリソースの設定を確認する

最も単純な形式では、検証アクションの追加は次のようになります。

TypeScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

JavaScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

Python

```
# stage was returned by pipeline.add_stage

stage.add_post(ShellStep("validate",
  commands=[''../tests/validate.sh'']
))
```

Java

```
// stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create("validate")
        .commands(Arrays.asList("'../tests/validate.sh'"))
```

```
        .build());
```

C#

```
// stage was returned by pipeline.addStage

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Commands = new string[] { "'../tests/validate.sh'" }
}));
```

多くの AWS CloudFormation デプロイでは、予測不可能な名前のリソースが生成されます。このため、CDK Pipelines はデプロイ後に AWS CloudFormation 出力を読み取る方法を提供します。これにより、ロードバランサーの生成された URL をテストアクションに渡す (例えば) ことができます。

出力を使用するには、関心のあるCfnOutputオブジェクトを公開します。次に、ステップのenvFromCfnOutputsプロパティに渡して、そのステップ内の環境変数として利用できるようにします。

TypeScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

JavaScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
```

```
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

Python

```
# given a stack lb_stack that exposes a load balancer construct as load_balancer
self.load_balancer_address = cdk.CfnOutput(lb_stack, "LbAddress",
    value=f"https://{lb_stack.load_balancer.load_balancer_dns_name}/")

# pass the load balancer address to a shell step
stage.add_post(ShellStep("lbaddr",
    env_from_cfn_outputs={"lb_addr": lb_stack.load_balancer_address}
    commands=["echo $lb_addr"]))
```

Java

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = CfnOutput.Builder.create(lbStack, "LbAddress")
                            .value(String.format("https://%s/",
                                    lbStack.loadBalancer.loadBalancerDnsName))
                            .build();

stage.addPost(ShellStep.Builder.create("lbaddr")
    .envFromCfnOutputs(     // Map.of requires Java 9 or later
        java.util.Map.of("lbAddr", loadBalancerAddress))
    .commands(Arrays.asList("echo $lbAddr"))
    .build());
```

C#

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = new CfnOutput(lbStack, "LbAddress", new CfnOutputProps
{
    Value = string.Format("https://{0}/", lbStack.loadBalancer.LoadBalancerDnsName)
});

stage.AddPost(new ShellStep("lbaddr", new ShellStepProps
{
    EnvFromCfnOutputs = new Dictionary<string, CfnOutput>
    {
        {  "lbAddr", loadBalancerAddress }
    },
```

```
    Commands = new string[] { "echo $lbAddr" }
}));
```

簡単な検証テストは で記述できますがShellStep、テストが数行を超えると、このアプローチは扱いにくくなります。より複雑なテストでは、 inputsプロパティShellStepを介して追加のファイル (完全なシェルスクリプトや他の言語のプログラムなど) を に取り込むことができます。入力は、ソース (GitHub リポジトリなど) や別の など、出力を持つ任意のステップにすることができますShellStep。

ファイルがテストで直接使用できる場合 (たとえば、ファイル自体が実行可能である場合) 、ソースリポジトリからファイルを取り込むのが適切です。この例では、 GitHub リポジトリを source (の一部としてインラインでインスタンス化するのではなく) として宣言しますCodePipeline。次に、このファイルセットをパイプラインと検証テストの両方に渡します。

TypeScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

JavaScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
```

```
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

Python

```
source   = CodePipelineSource.git_hub("OWNER/REPO", "main")

pipeline =  CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=ShellStep("Synth",
                    input=source,
                    commands=["npm install -g aws-cdk",
                        "python -m pip install -r requirements.txt",
                        "cdk synth"]))

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

stage.add_post(ShellStep("validate", input=source,
    commands=["sh ../tests/validate.sh"],
))
```

Java

```
final CodePipelineSource source = CodePipelineSource.gitHub("OWNER/REPO", "main");

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(ShellStep.Builder.create("Synth")
                .input(source)
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
```

```
                .build())
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(source)
        .commands(Arrays.asList("sh ../tests/validate.sh"))
        .build());
```

C#

```
var source = CodePipelineSource.GitHub("OWNER/REPO", "main");

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = new ShellStep("Synth", new ShellStepProps
    {
        Input = source,
        Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
    })
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = source,
    Commands = new string[] { "sh ../tests/validate.sh" }
}));
```

合成ステップから追加のファイルを取得することは、合成の一部として行われるテストをコンパイルする必要がある場合に適しています。

TypeScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

JavaScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, "test", {
  env: { account: "111111111111", region: "eu-west-1" }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
```

```
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

Python

```
synth_step = ShellStep("Synth",
                input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                commands=["npm install -g aws-cdk",
                  "python -m pip install -r requirements.txt",
                  "cdk synth"])

pipeline   = CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=synth_step)

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

# run a script that was compiled during synthesis
stage.add_post(ShellStep("validate",
    input=synth_step,
    commands=["node test/validate.js"],
))
```

Java

```
final ShellStep synth = ShellStep.Builder.create("Synth")
                            .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                            .commands(Arrays.asList("npm install -g aws-cdk", "cdk
 synth"))
                            .build();

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(synth)
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
```

```
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(synth)
        .commands(Arrays.asList("node ./tests/validate.js"))
        .build());
```

C#

```
var synth = new ShellStep("Synth", new ShellStepProps
{
    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
    Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
});

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = synth
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = synth,
    Commands = new string[] { "node ./tests/validate.js" }
}));
```

セキュリティ上の考慮事項

どのような形式の継続的デリバリーでも、固有のセキュリティリスクがあります。 AWS 責任共有モデル では、 AWS クラウド内の情報のセキュリティはお客様の責任となります。CDK Pipelines ライ

ブラリは、安全なデフォルトを組み込み、ベストプラクティスをモデリングすることで、すぐに開始できます。

ただし、本来の目的を達成するために高レベルのアクセスを必要とするライブラリは、完全なセキュリティを保証することはできません。 AWS および組織の外部には、多くの攻撃ベクトルがあります。

特に、次の点に注意してください。

- 依存するソフトウェアに注意してください。パイプラインで実行するすべてのサードパーティーソフトウェアを検証します。これは、デプロイされるインフラストラクチャが変更される可能性があるためです。
- 依存関係ロックを使用して、誤ってアップグレードされないようにします。CDK Pipelines は `package-lock.json`と を尊重し`yarn.lock`、依存関係が期待どおりであることを確認します。
- CDK Pipelines は、自分のアカウントで作成されたリソースで実行され、それらのリソースの設定は、パイプラインを介してコードを送信するデベロッパーによって制御されます。したがって、CDK Pipelines 自体は、コンプライアンスチェックをバイパスしようとする悪意のあるデベロッパーから保護できません。脅威モデルに CDK コードを記述するデベロッパーが含まれている場合は、 AWS CloudFormation 実行ロールに無効化するアクセス許可がないAWS CloudFormation フック (予防) や AWS Config (事後対応) などの外部コンプライアンスメカニズムを設定する必要があります。
- 実稼働環境の認証情報は有効期間が短くなければなりません。ブートストラップと初期プロビジョニングの後、デベロッパーがアカウント認証情報を持つ必要はありません。変更はパイプラインを通じてデプロイできます。認証情報を最初に必要としないことで、認証情報が漏洩する可能性を減らします。

トラブルシューティング

CDK Pipelines の使用を開始するときによく発生する問題は次のとおりです。

パイプライン: 内部障害

```
CREATE_FAILED  | AWS::CodePipeline::Pipeline | Pipeline/Pipeline
Internal Failure
```

GitHub アクセストークンを確認します。リポジトリが見つからないか、リポジトリへのアクセス許可がない可能性があります。

キー: ポリシーには、1 つ以上の無効なプリンシパルを含むステートメントが含まれています

```
CREATE_FAILED | AWS::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.
```

ターゲット環境の 1 つが新しいブートストラップスタックでブートストラップされていません。すべてのターゲット環境がブートストラップされていることを確認します。

スタックは ROLLBACK_COMPLETE 状態であり、更新できません。

```
Stack STACK_NAME is in ROLLBACK_COMPLETE state and can not be updated. (Service:
AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request
ID: ...)
```

スタックは以前のデプロイに失敗し、再試行不可能な状態です。コンソールから AWS CloudFormation スタックを削除し、デプロイを再試行してください。

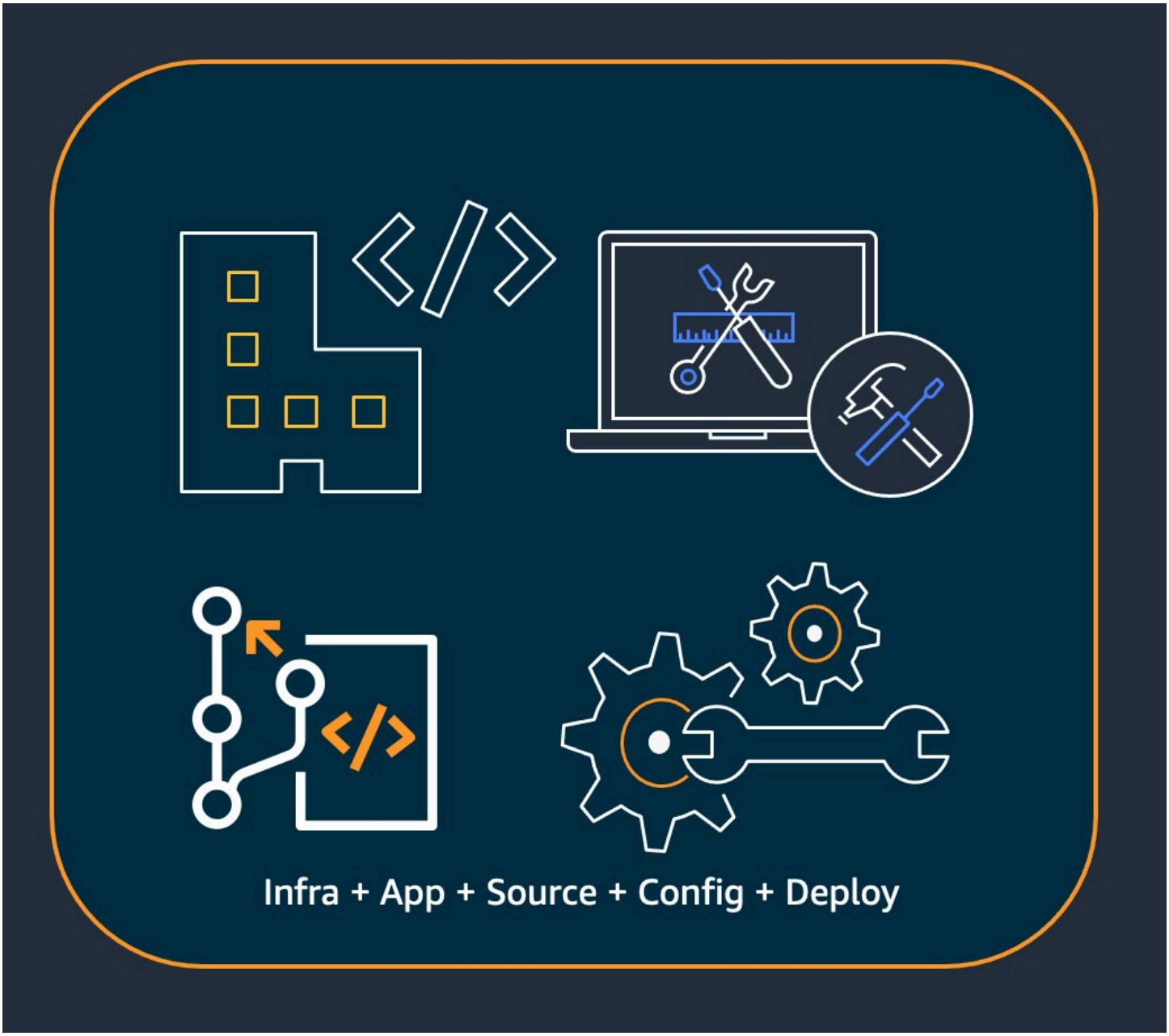
を使用したクラウドインフラストラクチャの開発とデプロイのベストプラクティス AWS CDK

を使用すると AWS CDK、デベロッパーまたは管理者は、サポートされているプログラミング言語を使用してクラウドインフラストラクチャを定義できます。CDK アプリケーションは、API、データベース、モニタリングリソースなどの論理ユニットに整理し、オプションで自動デプロイ用のパイプラインを用意する必要があります。論理ユニットは、以下を含むコンストラクトとして実装する必要があります。

- インフラストラクチャ (Amazon S3 バケット、Amazon RDS データベース、Amazon VPC ネットワークなど)
- ランタイムコード (AWS Lambda 関数など)
- 設定コード

スタックは、これらの論理ユニットのデプロイモデルを定義します。CDK の概念の詳細な概要については、「」を参照してください開始。

AWS CDK は、顧客や社内チームのニーズ、および複雑なクラウドアプリケーションのデプロイや継続的なメンテナンス中に発生することが多い障害パターンを慎重に考慮したものです。多くの場合、障害は、設定の変更など、完全にテストされていないアプリケーションへのout-of-band「」変更に関連していることがわかりました。したがって、ビジネスロジックだけでなく、インフラストラクチャと設定でもアプリケーション全体がコードで定義されているモデル AWS CDK を中心に開発しました。そうすれば、提案された変更を慎重に見直し、本番環境に似た環境で包括的にさまざまな度合いでテストし、問題が発生した場合は完全にロールバックできます。



デプロイ時に、 は以下を含むクラウドアセンブリを AWS CDK 合成します。

- AWS CloudFormation すべてのターゲット環境でインフラストラクチャを記述する テンプレート
- ランタイムコードとそのサポートファイルを含むファイルアセット

CDK を使用すると、アプリケーションのメインバージョン管理ブランチのすべてのコミットは、アプリケーションの完全で一貫性のあるデプロイ可能なバージョンを表すことができます。その後、アプリケーションは変更が行われるたびに自動的にデプロイできます。

の背後にある哲学は、推奨されるベストプラクティス AWS CDK につながります。ベストプラクティスは 4 つのカテゴリに分かれています。

- the section called “組織のベストプラクティス”
- the section called “コーディングのベストプラクティス”
- the section called “構築のベストプラクティス”
- the section called “アプリケーションのベストプラクティス”

> **Tip**
>
> CDK で定義されたインフラストラクチャに適用される場合は、 および使用する個々の AWS サービスのベストプラクティス AWS CloudFormationも考慮してください。

組織のベストプラクティス

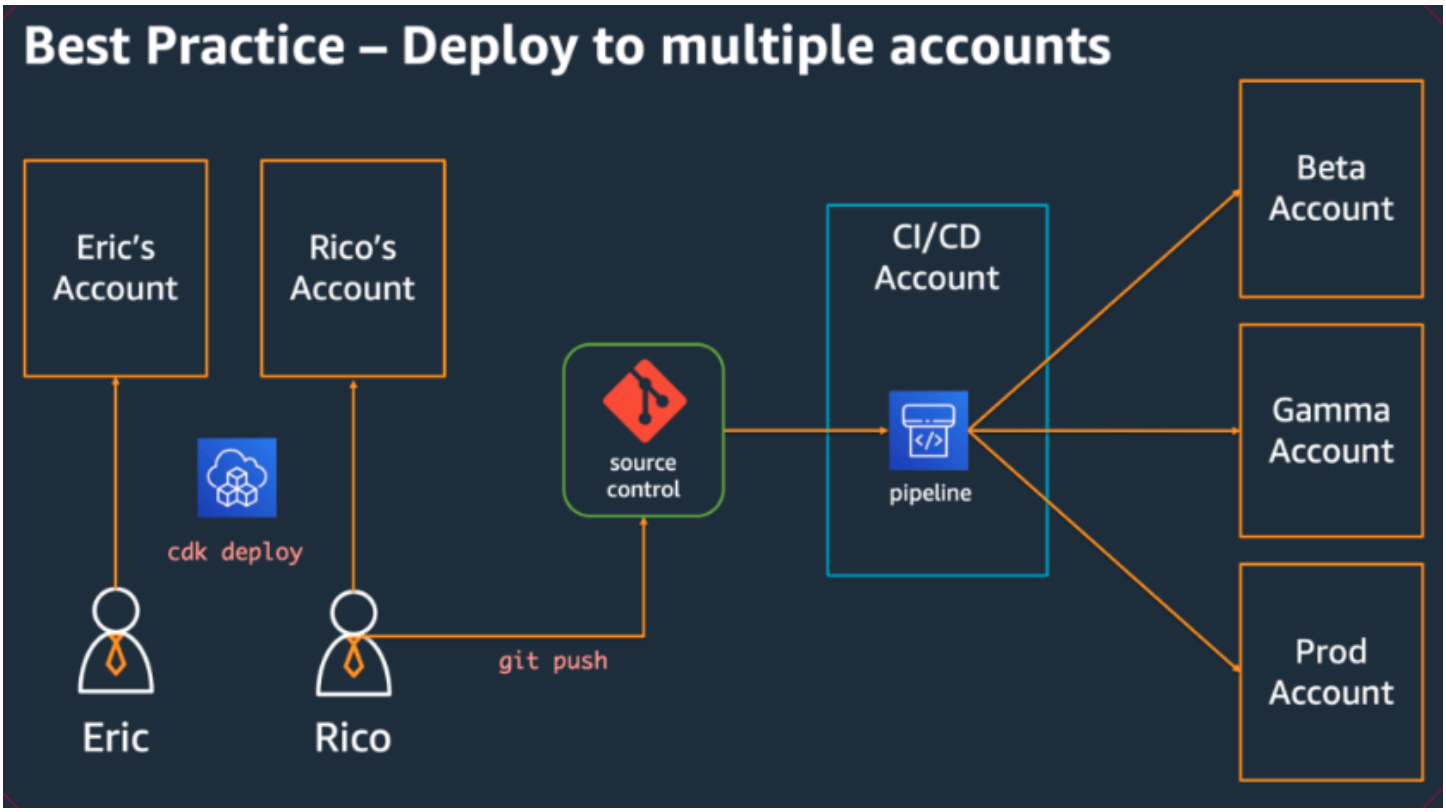
AWS CDK 導入の初期段階では、組織を成功させるためのセットアップ方法を検討することが重要です。CDK を採用する際に、エキスパートのチームがトレーニングを行い、残りの会社を指示することがベストプラクティスです。このチームのサイズは、小規模な会社の 1 人または 2 人から、大規模な企業でフルエッジの Cloud Center of Excellence (CCoE) までさまざまです。このチームは、社内のクラウドインフラストラクチャの標準とポリシーを設定するだけでなく、デベロッパーのトレーニングと参加も担当します。

CCoE は、クラウドインフラストラクチャに使用するプログラミング言語に関するガイダンスを提供する場合があります。詳細は組織によって異なりますが、適切なポリシーを採用することで、デベロッパーは会社のクラウドインフラストラクチャを理解して維持できます。

CCoE は、 内の組織単位を定義する「ランディングゾーン」も作成します AWS。ランディングゾーンは、ベストプラクティスの設計図に基づいて、事前設定され、安全でスケーラブルなマルチアカウント AWS 環境です。ランディングゾーンを構成するサービスを結び付けるには、 を使用できます。これはAWS Control Tower、単一のユーザーインターフェイスからマルチアカウントシステム全体を設定および管理します。

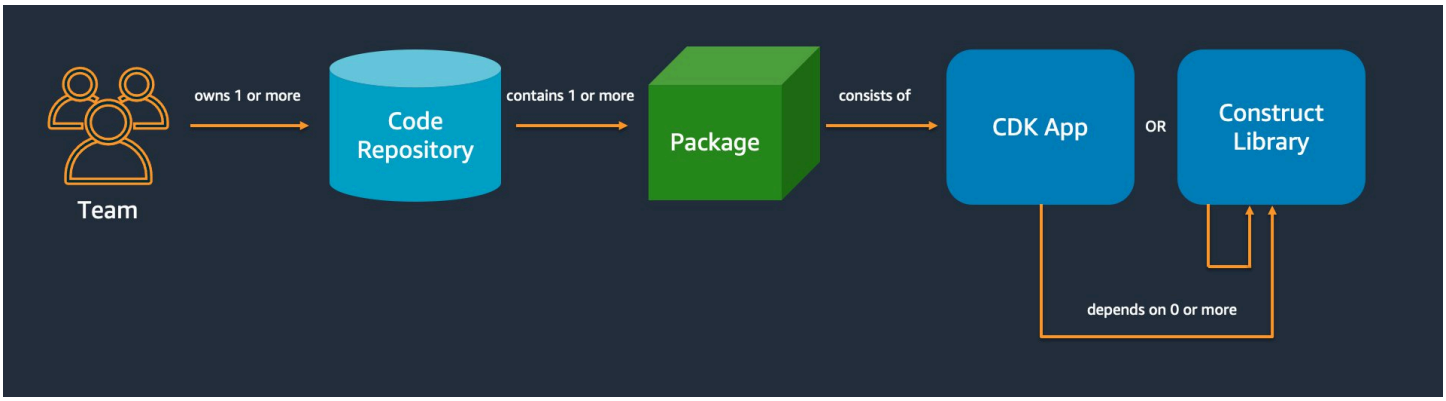
開発チームは、必要に応じて独自のアカウントを使用してテストを行い、これらのアカウントに新しいリソースをデプロイできる必要があります。個々のデベロッパーは、これらのリソースを独自の開発ワークステーションの拡張機能として扱うことができます。CDK Pipelines を使用すると、AWS CDK アプリケーションを CI/CD アカウントを介してテスト、統合、実稼働環境 (それぞれ独自

の AWS リージョンまたはアカウントに分離) にデプロイできます。これは、デベロッパーのコードを組織の正規リポジトリにマージすることによって行われます。



コーディングのベストプラクティス

このセクションでは、 AWS CDK コードを整理するためのベストプラクティスを紹介します。次の図は、チームとそのチームのコードリポジトリ、パッケージ、アプリケーション、コンストラクトライブラリの関係を示しています。



最初はシンプルに開始し、必要な場合にのみ複雑さを追加します。

ベストプラクティスのほとんどは、できるだけシンプルに保つことですが、よりシンプルではありません。要件がより複雑なソリューションを必要とする場合にのみ、複雑さを追加します。では AWS CDK、新しい要件をサポートするために、必要に応じてコードをリファクタリングできます。考えられるすべてのシナリオを事前に設計する必要はありません。

AWS Well-Architected フレームワークへの準拠

AWS Well-Architected フレームワークは、コンポーネントを、要件に対してまとめて提供するコード、設定、および AWS リソースとして定義します。コンポーネントは、多くの場合、技術所有権の単位であり、他のコンポーネントから切り離されます。ワークロードという用語は、一緒にビジネス価値をもたらす一連のコンポーネントを識別するために使用されます。ワークロードは通常、ビジネスリーダーとテクノロジーリーダーがコミュニケーションする詳細のレベルです。

Well-Architected Framework. AWS CDK apps AWS で定義されているように、 AWS CDK アプリケーションはコンポーネントにマッピングされます。 Well-Architected クラウドアプリケーションのベストプラクティスを調整および提供するためのメカニズムです。また、 などのアーティファクトリポジトリを使用して、コンポーネントを再利用可能なコードライブラリとして作成して共有することもできます AWS CodeArtifact。

すべてのアプリケーションは、1 つのリポジトリ内の 1 つのパッケージから開始します。

1 つのパッケージは AWS CDK アプリケーションのエントリポイントです。ここでは、アプリケーションのさまざまな論理ユニットをデプロイする方法と場所を定義します。また、CI/CD パイプラインを定義してアプリケーションをデプロイします。アプリケーションのコンストラクトは、ソリューションの論理単位を定義します。

複数のアプリケーションで使用するコンストラクトには、追加のパッケージを使用します。(共有コンストラクトには独自のライフサイクルとテスト戦略も必要です)。同じリポジトリ内のパッケージ間の依存関係は、リポジトリのビルドツールによって管理されます。

これは可能ですが、特に自動デプロイパイプラインを使用する場合は、複数のアプリケーションを同じリポジトリに配置することはお勧めしません。これにより、デプロイ中の変更の「ブラスト半径」が増加します。リポジトリに複数のアプリケーションがある場合、1 つのアプリケーションを変更すると、他のアプリケーションのデプロイがトリガーされます (他のアプリケーションが変更されていない場合でも)。さらに、1 つのアプリケーションに障害が発生すると、他のアプリケーションがデプロイされなくなります。

コードライフサイクルまたはチームの所有権に基づいてコードをリポジトリに移動する

パッケージが複数のアプリケーションで使用され始めたら、パッケージを独自のリポジトリに移動します。これにより、パッケージを使用するアプリケーション構築システムでパッケージを参照できます。また、アプリケーションライフサイクルとは無関係に定期的に更新することもできます。ただし、最初は、すべての共有コンストラクトを 1 つのリポジトリに配置するのが理にかなっている場合があります。

また、さまざまなチームが作業している場合は、パッケージを独自のリポジトリに移動します。これにより、アクセスコントロールを実施できます。

リポジトリの境界を越えてパッケージを使用するには、プライベートパッケージリポジトリが必要です。これは NPM、、 PyPiまたは Maven Central に似ていますが、組織内部にあります。また、パッケージをビルド、テスト、プライベートパッケージリポジトリに公開するリリースプロセスも必要です。CodeArtifact は、最も一般的なプログラミング言語のパッケージをホストできます。

パッケージリポジトリ内のパッケージへの依存関係は、 TypeScript や JavaScript アプリケーションの NPM など、言語のパッケージマネージャーによって管理されます。パッケージマネージャーは、ビルドが繰り返し可能であることを確認するのに役立ちます。これは、アプリケーションが依存するすべてのパッケージの特定のバージョンを記録することによって行われます。また、これらの依存関係を制御された方法でアップグレードすることもできます。

共有パッケージには別のテスト戦略が必要です。1 つのアプリケーションでは、テスト環境にアプリケーションをデプロイし、それでも動作することを確認するのに十分かもしれません。ただし、共有パッケージは、一般公開されているかのように、消費するアプリケーションとは独立してテストする必要があります。(組織によっては、実際に一部の共有パッケージを一般公開することを選択する場合があります)。

コンストラクトは任意に単純でも複雑でもかまいません。`Bucket` はコンストラクトですが、`CameraShopWebsite`はコンストラクトでもあります。

同じパッケージ内のインフラストラクチャとランタイムコードのライブ

は、インフラストラクチャをデプロイするための AWS CloudFormation テンプレートを生成するだけでなく、Lambda 関数や Docker イメージなどのランタイムアセット AWS CDK もバンドルし、インフラストラクチャと一緒にデプロイします。これにより、インフラストラクチャを定義するコードと、ランタイムロジックを実装するコードを 1 つのコンストラクトにまとめることができます。こ

れはベストプラクティスです。これら 2 種類のコードは、別々のリポジトリに置く必要も、別々のパッケージに置く必要もありません。

2 種類のコードを進化させるには、インフラストラクチャやロジックなど、機能を完全に記述した自己完結型のコンストラクトを使用できます。自己完結型のコンストラクトを使用すると、2 種類のコードを個別にテストし、プロジェクト間でコードを共有して再利用し、すべてのコードを同期してバージョン管理できます。

構築のベストプラクティス

このセクションでは、コンストラクトを開発するためのベストプラクティスについて説明します。コンストラクトは、リソースをカプセル化する再利用可能な組み合わせ可能なモジュールです。これらは AWS CDK アプリケーションの構成要素です。

コンストラクト付きのモデル、スタック付きのデプロイ

スタックはデプロイの単位であり、スタック内のすべてが一緒にデプロイされます。したがって、複数の AWS リソースからアプリケーションの上位レベルの論理ユニットを構築する場合は、各論理ユニットを としてではなく`Construct`、 として表します`Stack`。スタックは、さまざまなデプロイシナリオでコンストラクトをどのように構成し、接続するかを説明する場合にのみ使用します。

例えば、論理単位の 1 つがウェブサイトである場合、それを構成するコンストラクト (Amazon S3 バケット、API Gateway、Lambda 関数、Amazon RDS テーブルなど) は 1 つの高レベルのコンストラクトに構成する必要があります。次に、そのコンストラクトをデプロイ用の 1 つ以上のスタックでインスタンス化する必要があります。

構築用の コンストラクトとデプロイ用の スタックを使用することで、インフラストラクチャの再利用の可能性を高め、デプロイ方法をより柔軟にすることができます。

環境変数ではなく、プロパティとメソッドを使用して を設定する

コンストラクトとスタック内の環境変数ルックアップは、一般的なアンチパターンです。コンストラクトとスタックの両方がプロパティオブジェクトを受け入れて、完全な設定をコード内で完全に行えるようにする必要があります。そうしないと、コードが実行されるマシンへの依存関係が導入され、追跡と管理が必要な設定情報がさらに多く作成されます。

一般に、環境変数の検索は AWS CDK アプリの最上位レベルに制限する必要があります。また、開発環境で実行するために必要な情報を渡すためにも使用する必要があります。詳細については、「the section called “環境”」を参照してください。

インフラストラクチャのユニットテスト

すべての環境で構築時にユニットテストの完全なスイートを常に実行するには、合成中のネットワークルックアップを避け、コード内のすべての本番ステージをモデル化します。(これらのベストプラクティスについては、後で説明します)。1 つのコミットで常に同じ生成されたテンプレートが生成された場合は、作成したユニットテストを信頼して、生成されたテンプレートが期待どおりに表示されることを確認できます。詳細については、「コンストラクトのテスト」を参照してください。

ステートフルリソースの論理 ID を変更しない

リソースの論理 ID を変更すると、リソースは次のデプロイ時に新しいものに置き換えられます。データベースや S3 バケットなどのステートフルリソース、または Amazon VPC などの永続的なインフラストラクチャの場合、これはほとんど必要ありません。ID が変更される原因となる AWS CDK コードのリファクタリングには注意してください。ステートフルリソースの論理 IDs が静的のままであることをアサートする書き込みユニットテスト。論理 ID は、コンストラクトをインスタンス化するときにid指定した と、コンストラクトツリー内のコンストラクトの位置から導出されます。詳細については、「the section called “論理 IDs”」を参照してください。

コンストラクトがコンプライアンスに十分ではない

多くのエンタープライズのお客様は、L2 コンストラクト (組み込みのデフォルトとベストプラクティスで個々の AWS リソースを表す「キュレートされた」コンストラクト) の独自のラッパーを記述しています。これらのラッパーは、静的暗号化や特定の IAM ポリシーなどのセキュリティのベストプラクティスを適用します。例えば、通常の Amazon S3 Bucketコンストラクトの代わりにアプリケーションMyCompanyBucketで使用する を作成できます。 Amazon S3 このパターンは、ソフトウェア開発ライフサイクルの早い段階でセキュリティガイダンスを表示するのに便利ですが、唯一の強制手段としてそれに依存しないでください。

代わりに、サービスコントロールポリシーやアクセス許可の境界などの AWS 機能を使用して、組織レベルでセキュリティガードレールを適用します。the section called “側面” または CloudFormation Guard などのツールを使用して、デプロイ前にインフラストラクチャ要素のセキュリティプロパティに関するアサーションを行います。最適な動作 AWS CDK のために を使用します。

最後に、独自の「L2+」コンストラクトを作成すると、デベロッパーが Construct Hub のAWS ソリューションコンストラクトやサードパーティーコンストラクトなどの AWS CDK パッケージを利用できなくなる可能性があることに注意してください。これらのパッケージは通常、標準 AWS CDK コンストラクトに基づいて構築され、ラッパーコンストラクトを使用することはできません。

アプリケーションのベストプラクティス

このセクションでは、 コンストラクトを組み合わせてリソースの接続方法 AWS を定義し、 AWS CDK アプリケーションを作成する方法について説明します。

合成時に決定を行う

AWS CloudFormation ではデプロイ時に決定を行うことができますが (`Conditions`、`{ Fn::If }`、および を使用`Parameters`)、 AWS CDK ではこれらのメカニズムにいくつかアクセスできますが、使用しないことをお勧めします。使用できる値のタイプと、それらに対して実行できるオペレーションのタイプは、汎用プログラミング言語で利用できるものと比較して制限されます。

代わりに、プログラミング言語の`if`ステートメントやその他の機能を使用して、 AWS CDK アプリケーションでインスタンス化するコンストラクトなど、すべての決定を試みます。例えば、リストを反復処理し、リスト内の各項目の値を含むコンストラクトをインスタンス化する一般的な CDK イディオムは、 AWS CloudFormation 式を使用するだけでは不可能です。

言語ターゲット AWS CloudFormation としてではなく、 が堅牢なクラウドデプロイ AWS CDK に使用する実装の詳細として扱います。 TypeScript または Python で AWS CloudFormation テンプレートを記述するのではなく、デプロイ CloudFormation に が使用する CDK コードを記述します。

物理名ではなく、生成されたリソース名を使用する

名前は欠陥リソースです。各名前は 1 回だけ使用できます。したがって、テーブル名またはバケット名をインフラストラクチャとアプリケーションにハードコーディングする場合、そのインフラストラクチャを同じアカウントに 2 回デプロイすることはできません。(ここで説明する名前は、Amazon S3 バケットコンストラクトの `bucketName`プロパティなどで指定された名前です。)

何が悪いことなのか、リソースを置き換える必要があるリソースを変更することはできません。Amazon DynamoDB テーブル`KeySchema`の など、リソースの作成時にのみプロパティを設定できる場合、そのプロパティは変更できません。このプロパティを変更するには、新しいリソースが必要です。ただし、新しいリソースを実際に置き換えるには、同じ名前にする必要があります。ただし、既存のリソースがその名前を使用している間は、同じ名前にすることはできません。

名前をできるだけ少なく指定することをお勧めします。リソース名を省略すると、 AWS CDK は問題を引き起こさない方法でリソース名を生成します。リソースとしてテーブルがあるとします。その後、生成されたテーブル名を環境変数として AWS Lambda 関数に渡すことができます。 AWS

CDK アプリケーションで、テーブル名を として参照できますtable.tableName。または、起動時に Amazon EC2 インスタンスで設定ファイルを生成するか、実際のテーブル名を AWS Systems Manager Parameter Store に書き込んで、アプリケーションがそこから読み取ることができるようにすることもできます。

必要な場所が別の AWS CDK スタックの場合は、さらに簡単です。1 つのスタックがリソースを定義し、別のスタックがそのリソースを使用する必要があると仮定すると、以下が適用されます。

- 2 つのスタックが同じ AWS CDK アプリにある場合は、2 つのスタック間の参照を渡します。例えば、リソースのコンストラクトへの参照を、定義スタック () の属性として保存しますthis.stack.uploadBucket = myBucket。次に、その属性を、リソースを必要とするスタックのコンストラクタに渡します。
- 2 つのスタックが異なる AWS CDK アプリにある場合は、静的fromメソッドを使用して、ARN、名前、またはその他の属性に基づいて外部で定義されたリソースを使用します。 (例えば、DynamoDB テーブルTable.fromArn()には を使用します) 。CfnOutput コンストラクトを使用して、 の出力で ARN またはその他の必要な値を出力するかcdk deploy、 を調べます AWS Management Console。または、2 番目のアプリケーションは CloudFormation、最初のアプリケーションによって生成されたテンプレートを読み取り、 Outputsセクションからその値を取得できます。

削除ポリシーとログの保持を定義する

は、作成したすべてを保持するポリシーをデフォルトで保持することで、データが失われないようにAWS CDK 試みます。例えば、データを含むリソース (Amazon S3 バケットやデータベーステーブルなど) のデフォルトの削除ポリシーでは、リソースがスタックから削除されたときにリソースを削除しません。代わりに、リソースはスタックから孤立します。同様に、CDK のデフォルトはすべてのログを永久に保持することです。実稼働環境では、これらのデフォルトにより、実際に必要のない大量のデータと対応する AWS 請求書がすぐに保存される可能性があります。

これらのポリシーを本番稼働用リソースごとに何にするかを慎重に検討し、それに応じて指定します。the section called “側面” を使用して、スタックの削除ポリシーとログ記録ポリシーを検証します。

デプロイ要件に従ってアプリケーションを複数のスタックに分離する

アプリケーションが必要とするスタックの数には、ハードかつ高速なルールはありません。通常、デプロイパターンに基づいて決定します。次のガイドラインに注意してください。

- 通常、できるだけ多くのリソースを同じスタックに保持する方が簡単です。したがって、分離したいことがわかっていない限り、リソースをまとめておきます。
- ステートフルリソース (データベースなど) は、ステートレスリソースとは別のスタックに保存することを検討してください。その後、ステートフルスタックで終了保護を有効にできます。これにより、データ損失のリスクなしに、ステートレススタックの複数のコピーを自由に破棄または作成できます。
- ステートフルリソースは、コンストラクトの名前変更の影響を受けやすくなります。名前を変更すると、リソースが置き換えられます。したがって、移動または名前が変更される可能性が高いコンストラクト内にステートフルリソースをネストしないでください (キャッシュのように状態が失われた場合に状態を再構築できない場合を除く) 。これは、ステートフルリソースを独自のスタックに配置する別の良い理由です。

非決定的な動作を避けるため`cdk.context.json`にコミットする

AWS CDK デプロイを成功させるには、決定性が重要です。 AWS CDK アプリケーションは、特定の環境にデプロイされるたびに、基本的に同じ結果になります。

AWS CDK アプリケーションは汎用プログラミング言語で記述されるため、任意のコードを実行し、任意のライブラリを使用して、任意のネットワーク呼び出しを行うことができます。例えば、 AWS SDK を使用して、アプリケーションの合成中に AWS アカウントから情報を取得できます。そうすると、認証情報のセットアップ要件が増大し、レイテンシーが増加し、 を実行するたびに障害が発生する可能性は低くなります`cdk synth`。

合成中に AWS アカウントやリソースを変更しないでください。アプリを合成しても、副作用はありません。インフラストラクチャへの変更は、 AWS CloudFormation テンプレートの生成後のデプロイフェーズでのみ行う必要があります。これにより、問題が発生した場合、変更を自動的にロールバック AWS CloudFormation できます。 AWS CDK フレームワーク内で簡単に実行できない変更を行うには、カスタムリソースを使用してデプロイ時に任意のコードを実行します。

厳密に読み取り専用呼び出しでも、必ずしも安全ではありません。ネットワーク呼び出しによって返される値が変更された場合の動作を考慮します。これはインフラストラクチャのどの部分に影響しますか？ 既にデプロイされているリソースはどうなりますか？ 以下は、値の急激な変化が問題を引き起こす可能性がある 2 つの状況の例です。

- 指定したリージョンで使用可能なすべてのアベイラビリティーゾーンに Amazon VPC をプロビジョニングし、AZs の数がデプロイ日に 2 の場合、IP スペースは半分に分割されます。が翌日に新しいアベイラビリティーゾーン AWS を起動した場合、その次のデプロイでは IP スペースを 3 つに分割しようとします。そのため、すべてのサブネットを再作成する必要があります。Amazon

EC2 インスタンスがまだ実行されているため、この作業は可能ではない可能性があります。手動でクリーンアップする必要があります。

- 最新の Amazon Linux マシンイメージをクエリして Amazon EC2 インスタンスをデプロイし、次の日に新しいイメージがリリースされると、後続のデプロイで新しい AMI が取得され、すべてのインスタンスが置き換えられます。これは、想定されることではない可能性があります。

これらの状況は、 AWS側の変更が数か月または数年間正常にデプロイされた後に発生する可能性があるため、無悪になる可能性があります。突然、デプロイが「理由なし」に失敗し、何を行ったか、なぜ忘れたのか。

さいわい、 には、非決定的な値のスナップショットを記録するコンテキストプロバイダーと呼ばれるメカニズム AWS CDK が含まれています。これにより、将来の合成オペレーションで、最初にデプロイしたときとまったく同じテンプレートを生成できます。新しいテンプレートでの変更は、コードで行った変更だけです。コンストラクトの `.fromLookup()`メソッドを使用すると、呼び出しの結果は にキャッシュされます`cdk.context.json`。CDK アプリの将来の実行で同じ値が使用されるように、コードを他の部分とともにバージョン管理にコミットする必要があります。CDK Toolkit にはコンテキストキャッシュを管理するコマンドが含まれているため、必要に応じて特定のエントリを更新できます。詳細については、「the section called “Context”」を参照してください。

ネイティブ CDK コンテキストプロバイダーがない値 (AWS または他の場所から) が必要な場合は、別のスクリプトを作成することをお勧めします。スクリプトは値を取得してファイルに書き込み、CDK アプリでそのファイルを読み取る必要があります。スクリプトは、通常のビルドプロセスの一部としてではなく、保存された値を更新する場合にのみ実行します。

にロールとセキュリティグループの AWS CDK 管理を許可する

AWS CDK コンストラクトライブラリの`grant()`便利な方法では、最小限のスコープのアクセス許可を使用して、あるリソースへのアクセスを別のリソースに付与する AWS Identity and Access Management ロールを作成できます。例えば、次のような行があるとします。

```
myBucket.grantRead(myLambda)
```

この 1 行は、Lambda 関数のロール (ユーザー用にも作成されます) にポリシーを追加します。そのロールとそのポリシーは、 CloudFormation ユーザーが記述する必要のない の 12 行以上です。は、関数がバケットから読み取るために必要な最小限のアクセス許可のみ AWS CDK を付与します。

開発者がセキュリティチームによって作成された事前定義されたロールを常に使用する必要がある場合、 AWS CDK コーディングははるかに複雑になります。チームは、アプリケーションを設計する

方法に多くの柔軟性を失う可能性があります。代わりに、サービスコントロールポリシーとアクセス許可の境界を使用して、デベロッパーがガードレール内にとどまるようにすることをお勧めします。

コード内のすべての本番稼働ステージをモデル化する

従来の AWS CloudFormation シナリオでは、特定の環境に固有の設定値を適用した後にさまざまなターゲット環境にデプロイできるように、パラメータ化された単一のアーティファクトを生成することが目的です。CDK では、その設定をソースコードに組み込むことができます。本番環境用のスタックを作成し、他のステージごとに個別のスタックを作成します。次に、各スタックの設定値をコードに入れます。これらのリソースの名前または ARNs を使用して、出典管理にチェックインしたくない機密値には、Secrets Manager や Systems Manager Parameter Store などのサービスを使用します。

アプリケーションを合成すると、 `cdk.out`フォルダに作成されたクラウドアセンブリには、環境ごとに個別のテンプレートが含まれます。ビルド全体が決定的です。アプリケーションに変更はなく out-of-band、特定のコミットでは常にまったく同じ AWS CloudFormation テンプレートとそれに付随するアセットが生成されます。これにより、ユニットテストの信頼性が向上します。

すべてを測定する

人間の介入なしで完全な継続的デプロイという目標を達成するためには、高レベルの自動化が必要です。この自動化は、大量のモニタリングでのみ可能です。デプロイされたリソースのあらゆる側面を測定するには、メトリクス、アラーム、ダッシュボードを作成します。CPU 使用率やディスク容量などの測定を停止しないでください。また、ビジネスメトリクスを記録し、それらの測定値を使用して、ロールバックなどのデプロイの決定を自動化します。の L2 コンストラクトのほとんど AWS CDK には、dynamodb.Table クラスの `metricUserErrors()`メソッドなど、メトリクスの作成に役立つ便利なメソッドがあります。

AWS CDK CLI コマンドリファレンス

このセクションには、 コマンドラインインターフェイス () の AWS Cloud Development Kit (AWS CDK) コマンドリファレンス情報が含まれていますCLI。CDK CLIは CDK Toolkit とも呼ばれます。

使用方法

```
$ cdk <command> <arguments> <options>
```

コマンド

`acknowledge, ack`

発行番号で通知を確認し、再度表示しないようにします。

`bootstrap`

という名前の CDK ブートストラップスタックを AWS 環境にデプロイして、CDK `CDKToolkit`デプロイ用の AWS 環境を準備します。

`context`

CDK アプリケーションのキャッシュされたコンテキスト値を管理します。

`deploy`

環境に 1 つ以上の CDK スタックをデプロイします AWS 。

`destroy`

AWS 環境から 1 つ以上の CDK スタックを削除します。

`diff`

CDK スタック間のインフラストラクチャの変更を確認するには、差分を実行します。

`docs, doc`

ブラウザで CDK ドキュメントを開きます。

`doctor`

ローカル CDK プロジェクトと開発環境に関する有用な情報を検査して表示します。

`import`

AWS CloudFormation リソースのインポートを使用して、既存の AWS リソースを CDK スタックにインポートします。

`init`

テンプレートから新しい CDK プロジェクトを作成します。

`list, ls`

CDK アプリケーションからすべての CDK スタックとその依存関係を一覧表示します。

`metadata`

CDK スタックに関連付けられたメタデータを表示します。

`migrate`

AWS リソース、 AWS CloudFormation スタック、 AWS CloudFormation テンプレートを新しい CDK プロジェクトに移行します。

`notices`

CDK アプリケーションの通知を表示します。

`synthesize, synth`

CDK アプリを合成して、各スタックの AWS CloudFormation テンプレートを含むクラウドアセンブリを作成します。

`watch`

ローカル CDK プロジェクトで、デプロイとホットスワップを実行するための変更を継続的に監視します。

グローバルオプション

以下のオプションは、すべての CDK CLI コマンドと互換性があります。

`--app, -a` *`STRING`*

アプリケーションまたはクラウドアセンブリディレクトリを実行するための コマンドを指定します。

必須: はい

`--asset-metadata` *BOOLEAN*

アセットを使用するリソースの`aws:asset:*` AWS CloudFormation メタデータを含めます。

必須: いいえ

デフォルト値: `true`

`--build` *STRING*

事前合成ビルドを実行するためのコマンド。

必須: いいえ

`--ca-bundle-path` *STRING*

HTTPS リクエストを検証するときに使用する CA 証明書へのパス。

このオプションを指定しない場合、CDK CLI は`AWS_CA_BUNDLE`環境変数から読み取ります。

必須: はい

`--ci` *BOOLEAN*

CDK CLI コマンドが継続的インテグレーション (CI) 環境で実行されていることを示します。

このオプションは、CI パイプラインで一般的な自動オペレーションCLIにより適するように CDK の動作を変更します。

このオプションを指定すると、ログは `stdout`の代わりに に送信されます`stderr`。

必須: いいえ

デフォルト値: `false`

`--context, -c` *ARRAY*

コンテキスト文字列パラメータをキーと値のペアとして追加します。

`--debug` *BOOLEAN*

詳細なデバッグ情報を有効にします。このオプションでは、CDK がバックグラウンドで何をCLI しているのかについて、さらに詳しく説明する詳細な出力が生成されます。

必須: いいえ

デフォルト値: `false`

`--ec2creds, -i` *BOOLEAN*

CDK に Amazon EC2 インスタンスの認証情報の取得CLIを強制します。

デフォルトでは、CDK は Amazon EC2 インスタンスのステータスをCLI推測します。

必須: いいえ

デフォルト値: `false`

`--help, -h` *BOOLEAN*

CDK のコマンドリファレンス情報を表示しますCLI。

必須: いいえ

デフォルト値: `false`

`--ignore-errors` *BOOLEAN*

合成エラーを無視します。これは、無効な出力を生成する可能性があります。

必須: いいえ

デフォルト値: `false`

`--json, -j` *BOOLEAN*

標準出力 () に出力される AWS CloudFormation テンプレートには、YAML の代わりに JSON を使用しますstdout。

必須: いいえ

デフォルト値: `false`

`--lookups` *BOOLEAN*

コンテキストルックアップを実行します。

この値が でfalse、コンテキスト検索を実行する必要がある場合、合成は失敗します。

必須: いいえ

デフォルト値: `true`

`--no-color` *BOOLEAN*

コンソール出力から色やその他のスタイルを削除します。

必須: いいえ

デフォルト値: `false`

`--notices` *BOOLEAN*

関連する通知を表示します。

必須: いいえ

デフォルト値: `false`

`--output, -o` *STRING*

合成されたクラウドアセンブリを出力するディレクトリを指定します。

必須: はい

デフォルト値: `cdk.out`

`--path-metadata` *BOOLEAN*

各リソースの`aws:::cdk::path` AWS CloudFormation メタデータを含めます。

必須: いいえ

デフォルト値: `true`

`--plugin, -p` *ARRAY*

CDK 機能を拡張するnodeパッケージの名前またはパス。このオプションは、1 つのコマンドで複数回指定できます。

このオプションは、プロジェクトの `cdk.json` ファイルまたはローカル開発マシンの `~/.cdk.json` で設定できます。

```
{
   // ...
   "plugin": [
      "module_1",
```

```
      "module_2"
   ],
   // ...
}
```

必須: いいえ

`--profile` *STRING*

CDK で使用する AWS 環境情報を含む AWS プロファイルの名前を指定しますCLI。

必須: はい

`--proxy` *STRING*

指定されたプロキシを使用します。

このオプションを指定しない場合、CDK CLI はHTTPS_PROXY環境変数から読み取ります。

必須: はい

デフォルト値 : HTTPS_PROXY環境変数から読み取ります。

`--role-arn, -r` *STRING*

とのやり取り時に CDK がCLI引き受ける IAM ロールの ARN AWS CloudFormation。

必須: いいえ

`--staging` *BOOLEAN*

アセットを出力ディレクトリにコピーします。

を指定`false`して、出力ディレクトリへのアセットのコピーを防止します。これにより、 CLIは AWS SAM ローカルデバッグを実行するときに元のソースファイルを参照できます。

必須: いいえ

デフォルト値: `true`

`--strict` *BOOLEAN*

警告を含むスタックは作成しないでください。

必須: いいえ

デフォルト値: `false`

`--trace` *BOOLEAN*

スタック警告のトレースを出力します。

必須: いいえ

デフォルト値: `false`

`--verbose, -v` *COUNT*

デバッグログを表示します。このオプションは複数回指定して冗長性を高めることができます。

必須: いいえ

`--version` *BOOLEAN*

CDK CLIバージョン番号を表示します。

必須: いいえ

デフォルト値: `false`

`--version-reporting` *BOOLEAN*

合成された AWS CloudFormation テンプレートに `AWS::CDK::Metadata`リソースを含めます。

必須: いいえ

デフォルト値: `true`

オプションの提供と設定

コマンドライン引数を使用してオプションを渡すことができます。ほとんどのオプションは、設定ファイルで`cdk.json`設定できます。複数の設定ソースを使用する場合、CDK は次の優先順位CLIに従います。

1. コマンドライン値 – コマンドラインで提供されるオプションは、`cdk.json`ファイルで設定されたオプションを上書きします。
2. プロジェクト設定ファイル — CDK プロジェクトの ディレクトリにある`cdk.json`ファイル。
3. ユーザー設定ファイル – ローカルマシン`~/.cdk.json`の にある`cdk.json`ファイル。

コマンドラインでオプションを渡す

ブール値を渡す

ブール値を受け入れるオプションでは、次の方法で指定できます。

- `true` および `false`値を使用 – コマンドでブール値を指定します。以下に例を示します。

```
$ cdk deploy --watch=true
$ cdk deploy --watch=false
```

- オプションの対応するものを指定する – を追加して`false`値`no`を指定することでオプション名を変更します。以下に例を示します。

```
$ cdk deploy --watch
$ cdk deploy --no-watch
```

- デフォルトが `true`または のオプションの場合`false`、デフォルトから変更しない限り、 オプションを指定する必要はありません。

cdk acknowledge

発行番号で通知を確認し、再度表示しないようにします。

これは、対処された通知や適用されない通知を非表示にする場合に便利です。

確認は CDK プロジェクトレベルで保存されます。1 つの CDK プロジェクトで通知を承認しても、その通知は、承認されるまで他のプロジェクトに表示されます。

使用方法

```
$ cdk acknowledge <arguments> <options>
```

引数

通知 ID

通知の ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--help, -h` *BOOLEAN*

コマンドのcdk acknowledgeコマンドリファレンス情報を表示します。

例

別の CDKCLI コマンドの実行時に表示される通知を確認および非表示にする

```
$ cdk deploy
... # Normal output of the command

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: If a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.

        Affected versions: <1.126.0.

        More information at: https://github.com/aws/aws-cdk/issues/16603


17061   Error when building EKS cluster with monocdk import

        Overview: When using monocdk/aws-eks to build a stack containing
                  an EKS cluster, error is thrown about missing
                  lambda-layer-node-proxy-agent/layer/package.json.

        Affected versions: >=1.126.0 <=1.130.0.
```

```
        More information at: https://github.com/aws/aws-cdk/issues/17061

$ cdk acknowledge 16603
```

cdk bootstrap

という名前の CDK ブートストラップスタックを AWS 環境にデプロイして、CDK デプロイ用の AWS 環境CDKToolkitを準備します。

ブートストラップスタックは、 環境で Amazon S3 バケットと Amazon ECR リポジトリを AWS プロビジョニングする CloudFormation スタックです。 AWS CDK CLI は、これらのリソースを使用して、デプロイ中に合成されたテンプレートと関連アセットを保存します。

使用方法

```
$ cdk bootstrap <arguments> <options>
```

引数

AWS 環境

ブートストラップスタックをデプロイするターゲット AWS 環境の形式は ですaws://<account-id>/<region>。

例: aws://123456789012/us-east-1

この引数は、ブートストラップスタックを複数の環境にデプロイするために、1 つのコマンドで複数回指定できます。

デフォルトでは、CDK CLIは CDK アプリで参照されるすべての環境をブートストラップするか、デフォルトのソースから環境を決定します。これは、 --profileオプション、環境変数、またはデフォルトの AWS CLI ソースを使用して指定された環境です。

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--bootstrap-bucket-name, --toolkit-bucket-name, -b` *STRING*

CDK で使用される Amazon S3 バケットの名前CLI。このバケットは作成され、現在存在してはいけません。

Amazon S3 バケットのデフォルト名を上書きするには、このオプションを指定します。

このオプションを使用する場合は、合成をカスタマイズする必要がある場合があります。詳細については、「CDK スタック合成をカスタマイズする」を参照してください。

デフォルト値： 未定義

`--bootstrap-customer-key` *BOOLEAN*

ブートストラップバケットのカスタマーマスターキー (CMK) を作成します (課金されますが、最新のブートストラップのみアクセス許可をカスタマイズできます) 。

このオプションは `--bootstrap-kms-key-id` と互換性がありません。

デフォルト値： 未定義

`--bootstrap-kms-key-id` *STRING*

SSE-KMS 暗号化に使用する AWS KMS マスターキー ID。

Amazon S3 バケットの暗号化に使用されるデフォルト AWS KMS キーを上書きするには、このオプションを指定します。

このオプションは `--bootstrap-customer-key` と互換性がありません。

デフォルト値： 未定義

`--cloudformation-execution-policies` *ARRAY*

スタックのデプロイ AWS CloudFormation 中に が引き受けるデプロイロールにアタッチする必要があるマネージド ARNs。

デフォルトでは、スタックは `AdministratorAccess`ポリシーを使用して完全な管理者アクセス許可でデプロイされます。

このオプションは、1 つのコマンドで複数回指定できます。複数の ARNs 1 つの文字列として指定し、個々の ARNsをカンマで区切ることもできます。以下に例を示します。

```
$ cdk bootstrap --cloudformation-execution-policies "arn:aws:iam::aws:policy/
AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess"
```

デプロイの失敗を回避するには、指定したポリシーが、ブートストラップされる環境に実行するデプロイに十分であることを確認してください。

このオプションは、最新のブートストラップにのみ適用されます。

> **Important**
>
> 最新のブートストラップテンプレートは、 が暗示するアクセス許可`--cloudformation-execution-policies`を`--trust`リスト内の任意の AWS アカウントに効果的に付与します。デフォルトでは、これにより、ブートストラップされたアカウントの任意のリソースに対する読み取りおよび書き込みのアクセス許可が拡張されます。ブートストラップスタックには、使い慣れたポリシーと信頼できるアカウントを設定してください。

デフォルト値: []

`--custom-permissions-boundary, -cpb` *`STRING`*

使用するアクセス許可の境界の名前を指定します。

このオプションは `--example-permissions-boundary` と互換性がありません。

デフォルト値： 未定義

`--example-permissions-boundary, -epb` *`BOOLEAN`*

が提供するアクセス許可の境界の例を使用します AWS CDK。

このオプションは `--custom-permissions-boundary` と互換性がありません。

CDK が提供するアクセス許可の境界ポリシーを例として見なす必要があります。機能をテストする場合は、コンテンツを編集し、サンプルポリシーを参照してください。実際のデプロイ用の新しいポリシーが存在しない場合は、そのポリシーに変換します。問題はドリフトを回避することです。ほとんどの場合、アクセス許可の境界は維持され、専用の規則があり、命名が含まれています。

アクセス許可の境界の使用など、アクセス許可の設定の詳細については、「セキュリティと安全の開発ガイド」を参照してください。

デフォルト値： 未定義

`--execute` *BOOLEAN*

変更セットを実行するかどうかを設定します。

デフォルト値: `true`

`--force, -f` *BOOLEAN*

ブートストラップテンプレートのバージョンをダウングレードする場合でも、必ずブートストラップしてください。

デフォルト値: `false`

`--help, -h` *BOOLEAN*

コマンドの`cdk bootstrap`コマンドリファレンス情報を表示します。

`--previous-parameters` *BOOLEAN*

既存のパラメータには以前の値を使用します。

一連のパラメータを使用してブートストラップテンプレートをデプロイしたら、このオプションを に設定`false`して、今後のデプロイでパラメータを変更する必要があります。の場合`false`、以前に指定したすべてのパラメータを再供給する必要があります。

デフォルト値: `true`

`--public-access-block-configuration` *BOOLEAN*

CDK によって作成および使用される Amazon S3 バケットのパブリックアクセス設定をブロックしますCLI。

デフォルト値: `true`

`--qualifier` *STRING*

ブートストラップスタックごとに一意の文字列値。この値は、ブートストラップスタック内のリソースの物理 ID に追加されます。

修飾子を指定することで、同じ環境で複数のブートストラップスタックをプロビジョニングする際のリソース名の競合を回避できます。

修飾子を変更する場合、CDK アプリは変更された値をスタックシンセサイザーに渡す必要があります。詳細については、「CDK スタック合成をカスタマイズする」を参照してください。

デフォルト値: `hnb659fds`。この値には意味がありません。

`--show-template` *BOOLEAN*

ブートストラップの代わりに、現在のブートストラップテンプレートを標準出力 () に出力しますstdout。その後、必要に応じてテンプレートをコピーしてカスタマイズできます。

デフォルト値: false

`--tags, -t` *ARRAY*

の形式でブートストラップスタックに追加するタグKEY=VALUE。

デフォルト値: []

`--template` *STRING*

組み込みのテンプレートではなく、特定のファイルのテンプレートを使用します。

`--termination-protection` *BOOLEAN*

ブートストラップスタック AWS CloudFormation の終了保護を切り替えます。

の場合true、終了保護が有効になります。これにより、ブートストラップスタックが誤って削除されるのを防ぐことができます。

終了保護の詳細については、「 AWS CloudFormation ユーザーガイド」の「スタックが削除されないように保護する」を参照してください。

デフォルト値： 未定義

`--toolkit-stack-name` *STRING*

作成するブートストラップスタックの名前。

デフォルトでは、 は指定された AWS 環境に という名前のスタックCDKToolkitをcdk bootstrapデプロイします。ブートストラップスタックに別の名前を指定するには、このオプションを使用します。

デフォルト値: CDKToolkit

必須: はい

`--trust` *ARRAY*

この環境へのデプロイを実行するために信頼する必要がある AWS アカウント IDs。

ブートストラップを実行しているアカウントは常に信頼されます。

このオプションでは、 も指定する必要があります--cloudformation-execution-policies。

このオプションは、1 つのコマンドで複数回指定できます。

このオプションは、最新のブートストラップにのみ適用されます。

既存のブートストラップスタックに信頼されたアカウントを追加するには、以前に提供したアカウントを含め、信頼するすべてのアカウントを指定する必要があります。信頼する新しいアカウントのみを指定すると、以前に信頼されたアカウントは削除されます。

以下は、2 つのアカウントを信頼する例です。

```
$ cdk bootstrap aws://123456789012/us-west-2 --trust 234567890123 --
trust 987654321098 --cloudformation-execution-policies arn:aws:iam::aws:policy/
AdministratorAccess
 #  Bootstrapping environment aws://123456789012/us-west-2...
Trusted accounts for deployment: 234567890123, 987654321098
Trusted accounts for lookup: (none)
Execution policies: arn:aws:iam::aws:policy/AdministratorAccess
CDKToolkit: creating CloudFormation changeset...
 #  Environment aws://123456789012/us-west-2 bootstrapped.
```

> **Important**
>
> 最新のブートストラップテンプレートは、 が暗示するアクセス許可--cloudformation-execution-policiesを--trustリスト内の任意の AWS アカウントに効果的に付与します。デフォルトでは、これにより、ブートストラップされたアカウントの任意のリソースに対する読み取りおよび書き込みのアクセス許可が拡張されます。ブートストラップスタックには、使い慣れたポリシーと信頼できるアカウントを設定してください。

デフォルト値: []

`--trust-for-lookup` *ARRAY*

この環境で値を検索するために信頼する必要がある AWS アカウント IDs。

このオプションを使用して、環境にデプロイされるスタックを合成するアクセス許可をアカウントに付与します。実際には、これらのスタックを直接デプロイするアクセス許可は付与しません。

このオプションは、1 つのコマンドで複数回指定できます。

このオプションは、最新のブートストラップにのみ適用されます。

デフォルト値: []

例

prod プロファイルで指定された AWS 環境をブートストラップする

```
$ cdk bootstrap --profile prod
```

ブートストラップスタックを環境 foo と bar にデプロイする

```
$ cdk bootstrap --app='node bin/main.js' foo bar
```

ブートストラップテンプレートをエクスポートしてカスタマイズする

ブートストラップテンプレートで満たされていない特定の要件がある場合は、ニーズに合わせてカスタマイズできます。

ブートストラップテンプレートをエクスポートし、変更して、 を使用してデプロイできます AWS CloudFormation。既存のテンプレートをエクスポートする例を次に示します。

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

カスタムテンプレートを使用するCLIように CDK に指示することもできます。以下に例を示します。

```
$ cdk bootstrap --template my-bootstrap-template.yaml
```

アクセス許可の境界を持つブートストラップ。次に、そのアクセス許可の境界を削除します。

カスタムアクセス許可の境界を使用してブートストラップするには、以下を実行します。

```
$ cdk bootstrap --custom-permissions-boundary my-permissions-boundary
```

アクセス許可の境界を削除するには、以下を実行します。

```
$ cdk bootstrap --no-previous-parameters
```

修飾子を使用して、開発環境用に作成されたリソースを区別する

```
$ cdk bootstrap --qualifier dev2024
```

cdk context

AWS CDK アプリケーションのキャッシュされたコンテキスト値を管理します。

コンテキストは、スタックの合成とデプロイ方法に影響を与える可能性のある設定と環境情報を表します。`cdk context` を使用して以下を行います。

- 設定したコンテキスト値を表示します。
- コンテキスト値を設定および管理します。
- コンテキスト値を削除します。

使用方法

```
$ cdk context <options>
```

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--clear` *BOOLEAN*

すべてのコンテキストをクリアします。

`--force, -f` *BOOLEAN*

欠落しているキーエラーを無視します。

デフォルト値: `false`

`--help, -h` *BOOLEAN*

コマンドの`cdk context`コマンドリファレンス情報を表示します。

`--reset, -e` *`STRING`*

リセットするコンテキストキーまたはそのインデックス。

cdk deploy

環境に 1 つ以上の AWS CDK スタックをデプロイします AWS 。

デプロイ中、CDK CLIはコンソールから確認できるものと似た進行状況インジケータを出力します AWS CloudFormation 。

AWS 環境がブートストラップされていない場合、アセットがなく、合成されたテンプレートが 51,200 バイト未満のスタックのみが正常にデプロイされます。

使用方法

```
$ cdk deploy <arguments> <options>
```

引数

CDK スタック論理 ID

デプロイするアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--all` *`BOOLEAN`*

CDK アプリにすべてのスタックをデプロイします。

デフォルト値: `false`

`--asset-parallelism` *`BOOLEAN`*

アセットを並行して構築および公開するかどうかを指定します。

`--asset-prebuild` *BOOLEAN*

最初のスタックをデプロイする前に、すべてのアセットを構築するかどうかを指定します。このオプションは、Dockerビルドの失敗に役立ちます。

デフォルト値: `true`

`--build-exclude, -E` *ARRAY*

指定された ID でアセットを再構築しないでください。

このオプションは、1 つのコマンドで複数回指定できます。

デフォルト値: `[]`

`--change-set-name` *STRING*

作成する AWS CloudFormation 変更セットの名前。

このオプションは `--method='direct'` と互換性がありません。

`--concurrency` *NUMBER*

スタック間の依存関係を考慮しながら、複数のスタックを並行してデプロイします。デプロイを高速化するには、このオプションを使用します。その他の AWS アカウント レート制限 AWS CloudFormation も考慮する必要があります。

実行する同時デプロイの最大数 (依存関係の許可) を指定する数値を指定します。

デフォルト値: `1`

`--exclusively, -e` *BOOLEAN*

リクエストされたスタックのみをデプロイし、依存関係は含まれません。

`--force, -f` *BOOLEAN*

デプロイして既存のスタックを更新すると、CDK CLIはデプロイされたスタックのテンプレートとタグをデプロイしようとしているスタックと比較します。変更が検出されない場合、CDK CLIはデプロイをスキップします。

この動作を上書きし、変更が検出されなくてもスタックを常にデプロイするには、このオプションを使用します。

デフォルト値: `false`

`--help, -h` *BOOLEAN*

コマンドのcdk deployコマンドリファレンス情報を表示します。

`--hotswap` *BOOLEAN*

開発を高速化するためのホットスワップデプロイ。このオプションは、可能であれば、より高速でホットスワップデプロイを実行しようとします。例えば、CDK アプリで Lambda 関数のコードを変更すると、CDK CLIは CloudFormation デプロイを実行する代わりにサービス APIsを通じてリソースを直接更新します。

CDK がホットスワップをサポートしていない変更CLIを検出すると、それらの変更は無視され、メッセージが表示されます。フル CloudFormation デプロイをフォールバックとして実行する場合は、`--hotswap-fallback`代わりに を使用します。

CDK CLIは、現在の AWS 認証情報を使用して API コールを実行します。`@aws-cdk/core:newStyleStackSynthesis` 機能フラグが に設定されている場合でも、ブートストラップスタックのロールは引き受けられませんtrue。これらのロールには、 を使用せずに AWS リソースを直接更新するために必要なアクセス許可はありません CloudFormation。そのため、認証情報がホットスワップデプロイを実行しているスタック AWS アカウント と同じスタック用であり、リソースを更新するために必要な IAM アクセス許可があることを確認してください。

ホットスワップは現在、以下の変更でサポートされています。

- Lambda 関数のコードアセット (Dockerイメージとインラインコードを含む)、タグの変更、および設定の変更 (説明と環境変数のみがサポートされます)。
- Lambda のバージョンとエイリアスの変更。
- AWS Step Functions ステートマシンの定義変更。
- Amazon ECS サービスのコンテナアセットの変更。
- Amazon S3 バケットデプロイのウェブサイトアセットの変更。
- AWS CodeBuild プロジェクトのソースと環境の変更。
- AWS AppSync リゾルバーと関数の VTL マッピングテンプレートの変更。
- APIsの AWS AppSync GraphQLスキーマの変更。

特定の CloudFormation 組み込み関数の使用は、ホットスワップデプロイの一部としてサポートされています。具体的には次のとおりです。

- `Ref`
- `Fn::GetAtt` – 部分的にのみサポートされています。サポートされているリソースと属性については、この実装を参照してください。

- `Fn::ImportValue`
- `Fn::Join`
- `Fn::Select`
- `Fn::Split`
- `Fn::Sub`

このオプションは、ネストされたスタックとも互換性があります。

> **Note**
>
> - このオプションは、デプロイを高速化するために CloudFormation スタックに意図的にドリフトを導入します。このため、開発目的でのみ使用してください。このオプションは、本番稼働用デプロイには使用しないでください。
> - このオプションは実験的と見なされ、将来的に重大な変更が生じる可能性があります。
> - 特定のパラメータのデフォルトは、ホットスワップパラメータと異なる場合があります。例えば、Amazon ECS サービスの最小正常率は、現在 に設定されます`0`。これが発生した場合は、それに応じてソースを確認します。

デフォルト値: `false`

`--hotswap-fallback` *`BOOLEAN`*

このオプションは に似ています`--hotswap`。違い`--hotswap-fallback`は、それを必要とする変更が検出された場合、 がフォールバックして完全な CloudFormation デプロイを実行することです。

このオプションの詳細については、「`--hotswap`」を参照してください。

デフォルト値: `false`

`--ignore-no-stacks` *`BOOLEAN`*

CDK アプリケーションにスタックが含まれていない場合でも、デプロイを実行します。

このオプションは、次のシナリオで役立ちます。 `dev`や などの複数の環境を持つアプリがある場合がありますprod。開発を開始するときに、prod アプリにリソースがないか、リソースがコメントアウトされる可能性があります。これにより、デプロイエラーが発生し、アプリケーションにスタックがないことを示すメッセージが表示されます。を使用して、このエラー`--ignore-no-stacks`をバイパスします。

デフォルト値: `false`

`--logs` *`BOOLEAN`*

選択したスタック内のすべてのリソースからのすべてのイベントについて、標準出力 (`stdout`) に Amazon CloudWatch ログを表示します。

このオプションは とのみ互換性があります`--watch`。

デフォルト値: `true`

`--method, -m` *`STRING`*

デプロイを実行するように メソッドを設定します。

- `change-set` – デフォルトのメソッド。CDK CLIは、デプロイされる CloudFormation 変更を含む変更セットを作成し、デプロイを実行します。
- `direct` – 変更セットを作成しないでください。代わりに、変更をすぐに適用します。これは通常、変更セットを作成するよりも高速ですが、進行状況情報が失われます。
- `prepare-change-set` – 変更セットを作成しますが、デプロイは実行しません。これは、変更セットを検査する外部ツールがある場合や、変更セットの承認プロセスがある場合に便利です。

有効な値: `change-set`、`direct`、`prepare-change-set`

デフォルト値: `change-set`

`--notification-arns` *`ARRAY`*

スタック関連イベント CloudFormation を通知する Amazon SNS トピックARNs。

`--outputs-file, -O` *`STRING`*

デプロイからのスタック出力が書き込まれる場所へのパス。

デプロイ後、スタック出力は JSON 形式で指定された出力ファイルに書き込まれます。

このオプションは、プロジェクトの `cdk.json` ファイルまたはローカル開発マシンの `~/.cdk.json` で設定できます。

```
{
   "app": "npx ts-node bin/myproject.ts",
   // ...
   "outputsFile": "outputs.json"
}
```

複数のスタックがデプロイされている場合、出力は同じ出力ファイルに書き込まれ、スタック名を表すキーで編成されます。

`--parameters` *ARRAY*

デプロイ CloudFormation 中に追加のパラメータを に渡します。

このオプションは、 形式の配列を受け入れますSTACK:KEY=VALUE。

- STACK – パラメータを関連付けるスタックの名前。
- KEY – スタックの パラメータの名前。
- VALUE – デプロイ時に渡す値。

スタック名が指定されていない場合、または *がスタック名として指定されている場合、パラメータはデプロイされるすべてのスタックに適用されます。スタックが パラメータを使用しない場合、デプロイは失敗します。

パラメータはネストされたスタックには伝達されません。ネストされたスタックにパラメータを渡すには、 NestedStackコンストラクトを使用します。

デフォルト値: {}

`--previous-parameters` *BOOLEAN*

既存のパラメータには以前の値を使用します。

このオプションを に設定する場合はfalse、すべてのデプロイですべてのパラメータを指定する必要があります。

デフォルト値: true

`--progress` *STRING*

CDK がデプロイの進行状況CLIを表示する方法を設定します。

- bar — スタックデプロイイベントを進行状況バーとして表示し、リソースのイベントは現在デプロイされています。
- events – すべての CloudFormation イベントを含む完全な履歴を提供します。

このオプションは、プロジェクトの cdk.json ファイルまたはローカル開発マシンの ~/.cdk.json で設定することもできます。

```
{
   "progress": "events"
```

```
}
```

有効な値:bar、events|

デフォルト値: bar

`--require-approval` *STRING*

手動承認が必要なセキュリティ重視の変更を指定します。

- any-change – スタックへの変更には手動承認が必要です。
- broadening – 変更に許可またはセキュリティグループルールの拡張が含まれる場合は、手動承認が必要です。
- never – 承認は必要ありません。

有効な値: any-change、broadening、never

デフォルト値: broadening

`--rollback` *BOOLEAN*

デプロイ中にリソースの作成または更新に失敗した場合、デプロイは CDK がCLI戻る前に最新の安定状態にロールバックされます。その時点までに行われたすべての変更は元に戻されます。作成されたリソースは削除され、更新はロールバックされます。

この動作を無効にするfalseには、 を指定します。リソースの作成または更新に失敗した場合、CDK CLI はその時点までに行われた変更をそのままにして、 を返します。これは、すぐに反復する開発環境で役立つ場合があります。

には--rollback=false、 --no-rollbackまたは を使用できます-R。

> **Note**
>
> の場合false、リソースの置換を引き起こすデプロイは常に失敗します。このオプションは、新しいリソースを更新または作成するデプロイにのみ使用できます。

デフォルト値: true

`--toolkit-stack-name` *STRING*

既存の CDK Toolkit スタックの名前。

このオプションは、レガシー合成を使用する CDK アプリケーションにのみ使用されます。

`--watch` *BOOLEAN*

CDK プロジェクトファイルを継続的に監視し、変更が検出されたときに指定されたスタックを自動的にデプロイします。

このオプションは`--hotswap`、デフォルトで を意味します。

このオプションには同等の CDK CLI コマンドがあります。詳細については、「cdk watch」を参照してください。

例

という名前のスタックをデプロイする MyStackName

```
$ cdk deploy MyStackName --app='node bin/main.js'
```

アプリケーションに複数のスタックをデプロイする

`cdk list` を使用してスタックを一覧表示します。

```
$ cdk list
CdkHelloWorldStack
CdkStack2
CdkStack3
```

すべてのスタックをデプロイするには、 `--all`オプションを使用します。

```
$ cdk deploy --all
```

デプロイするスタックを選択するには、スタック名を引数として指定します。

```
$ cdk deploy CdkHelloWorldStack CdkStack3
```

パイプラインスタックをデプロイする

を使用してスタック名`cdk list`をパスとして表示し、パイプライン階層内の場所を示します。

```
$ cdk list
PipelineStack
PiplelineStack/Prod
```

```
PipelineStack/Prod/MyService
```

--all オプションまたはワイルドカード*を使用して、すべてのスタックをデプロイします。上記のようにスタックの階層がある場合、 --allと *は最上位レベルのスタックのみに一致します。階層内のすべてのスタックを照合するには、 を使用します**。

これらのパターンを組み合わせることができます。次の は、Prodステージ内のすべてのスタックをデプロイします。

```
$ cdk deploy PipelineStack/Prod/**
```

デプロイ時にパラメータを渡す

CDK スタックでパラメータを定義します。Amazon SNS トピックTopicNameParamに という名前のパラメータを作成する例を次に示します。

```
new sns.Topic(this, 'TopicParameter', {
    topicName: new cdk.CfnParameter(this, 'TopicNameParam').value.toString()
});
```

のパラメータ値を指定するにはparameterized、以下を実行します。

```
$ cdk deploy --parameters "MyStackName:TopicNameParam=parameterized"
```

--force オプションを使用してパラメータ値を上書きできます。以下は、以前のデプロイからトピック名を上書きする例です。

```
$ cdk deploy --parameters "MyStackName:TopicNameParam=parameterName" --force
```

デプロイ後にスタック出力をファイルに書き込む

CDK スタックファイルに出力を定義します。関数 ARN の出力を作成する例を次に示します。

```
const fn = new lambda.Function(this, "fn", {
  handler: "index.handler",
  code: lambda.Code.fromInline(`exports.handler = \${handler.toString()}`),
  runtime: lambda.Runtime.NODEJS_LATEST
});

new cdk.CfnOutput(this, 'FunctionArn', {
```

```
  value: fn.functionArn,
});
```

スタックをデプロイし、出力を に書き込みますoutputs.json。

```
$ cdk deploy --outputs-file outputs.json
```

デプロイoutputs.json後の の例を次に示します。

```
{
  "MyStack": {
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MyStack-fn5FF616E3-
G632ITHSP5HK"
  }
}
```

この例では、 キーはCfnOutputインスタンスの論理 ID FunctionArnに対応します。

複数のスタックがデプロイされている場合のデプロイoutputs.json後の例を次に示します。

```
{
  "MyStack": {
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MyStack-fn5FF616E3-
G632ITHSP5HK"
  },
  "AnotherStack": {
    "VPCId": "vpc-z0mg270fee16693f"
  }
}
```

デプロイ方法を変更する

変更セットを使用せずに迅速にデプロイするには、 を使用します--method='direct'。

```
$ cdk deploy --method='direct'
```

変更セットを作成してデプロイしない場合は、 を使用します--method='prepare-change-set'。デフォルトでは、 という名前の変更セットcdk-deploy-change-setが作成されます。この名前の以前の変更セットが存在する場合、上書きされます。変更が検出されない場合、空の変更セットは引き続き作成されます。

変更セットに名前を付けることもできます。以下に例を示します。

```
$ cdk deploy --method='prepare-change-set' --change-set-name='MyChangeSetName'
```

cdk destroy

AWS 環境から 1 つ以上の AWS CDK スタックを削除します。

スタックを削除すると、 DeletionPolicyの で設定されていない限り、スタック内のリソースは破棄されますRetain。

スタックの削除中、このコマンドはcdk deploy動作と同様の進行状況情報を出力します。

使用方法

```
$ cdk destroy <arguments> <options>
```

引数

CDK スタック論理 ID

削除するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--all` *BOOLEAN*

使用可能なすべてのスタックを破棄します。

デフォルト値: `false`

`--exclusively, -e` *BOOLEAN*

リクエストされたスタックのみを破棄し、依存関係を含めないでください。

`--force, -f` *BOOLEAN*

スタックを破棄する前に確認を求めないでください。

`--help, -h` *BOOLEAN*

コマンドの`cdk destroy`コマンドリファレンス情報を表示します。

例

という名前のスタックを削除する MyStackName

```
$ cdk destroy --app='node bin/main.js' MyStackName
```

cdk diff

差分を実行して、 AWS CDK スタック間のインフラストラクチャの変更を確認します。

このコマンドは通常、ローカル CDK アプリのスタックの現在の状態とデプロイされたスタックの違いを比較するために使用されます。ただし、デプロイされたスタックを任意のローカル AWS CloudFormation テンプレートと比較することもできます。

使用方法

```
$ cdk diff <arguments> <options>
```

引数

CDK スタック論理 ID

差分を実行するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--change-set` *BOOLEAN*

リソースの置換を分析する変更セットを作成するかどうかを指定します。

の場合`true`、CDK CLIは AWS CloudFormation スタックに加えられる正確な変更を表示する変更セットを作成します。この出力には、リソースを更新または置き換えるかどうかが含まれます。CDK CLIは、ルックアップロールの代わりにデプロイロールを使用してこのアクションを実行します。

の場合`false`、 CloudFormation テンプレートを比較することで、より速く、精度の低い差分が実行されます。リソースの置換を必要とするプロパティに対して検出された変更は、リソースリファレンスをハードコードされた ARN に置き換えるなど、純粋に変更があった場合でも、リソースの置換として表示されます。

デフォルト値: `true`

`--context-lines` *NUMBER*

任意の JSON 差分レンダリングに含めるコンテキスト行の数。

デフォルト値: `3`

`--exclusively, -e` *BOOLEAN*

リクエストされたスタックの差分のみ、依存関係は含まれません。

`--fail` *BOOLEAN*

差異が検出され1た場合は、 のコードで失敗して終了します。

`--help, -h` *BOOLEAN*

コマンドの`cdk diff`コマンドリファレンス情報を表示します。

`--processed` *BOOLEAN*

CloudFormation 変換が既に処理されているテンプレートと比較するかどうかを指定します。

デフォルト値: `false`

`--quiet, -q` *BOOLEAN*

変更が検出され`stdout`ない場合は、CDK スタック名とデフォルト`cdk diff`メッセージを に出力しないでください。

デフォルト値: `false`

`--security-only` *BOOLEAN*

セキュリティの変更が拡大された場合にのみ差分があります。

デフォルト値: `false`

`--strict` *BOOLEAN*

cdk diff 動作をより正確または厳密に変更します。true の場合、CDK CLI はAWS::CDK::Metadataリソースや読み取り不可能な非 ASCII 文字を除外しません。

デフォルト値: `false`

`--template` *STRING*

CDK スタックを比較する CloudFormation テンプレートへのパス。

例

という名前の現在デプロイされているスタックと異なる MyStackName

```
$ cdk diff MyStackName --app='node bin/main.js'
```

特定の CloudFormation テンプレートとの差分

```
$ cdk diff MyStackName --app='node bin/main.js' --template-path='./
MyStackNameTemplate.yaml'
```

ローカルスタックとデプロイされたスタックを区別します。変更が検出されない場合、stdout に出力しない

```
$ cdk diff MyStackName --app='node bin/main.js' --quiet
```

cdk docs

ブラウザで AWS CDK ドキュメントを開きます。

使用方法

```
$ cdk docs <options>
```

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--browser, -b` *`STRING`*

ブラウザを開くために使用するコマンド。開くファイルのパスのプレースホルダー%uとして を使用します。

デフォルト値: `open %u`

`--help, -h` *`BOOLEAN`*

コマンドの`cdk docs`コマンドリファレンス情報を表示します。

例

で AWS CDK ドキュメントを開く Google Chrome

```
$ cdk docs --browser='chrome %u'
```

cdk doctor

ローカル AWS CDK プロジェクトと開発環境に関する有用な情報を検査して表示します。

この情報は CDK の問題のトラブルシューティングに役立つため、バグレポートを送信するときに提供する必要があります。

使用方法

```
$ cdk doctor <options>
```

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--help, -h` *BOOLEAN*

コマンドのcdk doctorコマンドリファレンス情報を表示します。

例

cdk doctor コマンドの簡単な例

```
$ cdk doctor
## CDK Version: 1.0.0 (build e64993a)
## AWS environment variables:
  - AWS_EC2_METADATA_DISABLED = 1
  - AWS_SDK_LOAD_CONFIG = 1
```

cdk import

AWS CloudFormation リソースのインポートを使用して、既存の AWS リソースを CDK スタックにインポートします。

このコマンドを使用すると、他のメソッドを使用して作成された既存のリソースを取得し、 を使用して管理を開始できます AWS CDK。

リソースを CDK 管理に移行することを検討する場合、IAM ロール、Lambda 関数、イベントルールなどを使用して新しいリソースを作成することが許容されることがあります。Amazon S3 バケットや DynamoDB テーブルなどのステートフルリソースなど、他のリソースでは、新しいリソースを作成するとサービスに影響を与える可能性があります。を使用してcdk import、サービスの中断を最小限に抑えながら既存のリソースをインポートできます。サポートされている AWS リソースのリストについては、「 ユーザーガイド」の「リソースタイプのサポートAWS CloudFormation 」を参照してください。

既存のリソースを CDK スタックにインポートするには

1. を実行してcdk diff、CDK スタックに保留中の変更がないことを確認します。を実行する場合cdk import、インポートオペレーションで許可される変更は、インポートされる新しいリソースの追加のみです。
2. スタックにインポートするリソースのコンストラクトを追加します。例えば、Amazon S3 バケットに以下を追加します。

```
new s3.Bucket(this, 'ImportedS3Bucket', {});
```

その他の変更は追加しないでください。また、リソースが現在持っている状態を正確にモデル化する必要があります。バケットの例では、 AWS KMS キー、ライフサイクルポリシー、およびバケットに関連するその他のものを必ず含めてください。そうしないと、後続の更新オペレーションが期待どおりに動作しない可能性があります。

3. `cdk import` を実行します。CDK アプリに複数のスタックがある場合は、特定のスタック名を引数として渡します。
4. CDK CLIは、インポートするリソースの実際の名前を渡すように促します。この情報を入力すると、インポートが開始されます。
5. が成功`cdk import`を報告すると、リソースは CDK によって管理されます。コンストラクト設定の後の変更は、リソースに反映されます。

この機能には現在、以下の制限があります。

- ネストされたスタックにリソースをインポートすることはできません。
- 指定したプロパティがインポートされたリソースに対して正しく、完全かどうかはチェックされません。インポート後にドリフト検出オペレーションを開始してみてください。
- 他のリソースに依存するリソースはすべて、正しい順序でまとめて、または個別にインポートする必要があります。そうしないと、未解決のリファレンスで CloudFormation デプロイが失敗します。
- このコマンドは、暗号化されたステージングバケットを読み取るために必要なデプロイロール認証情報を使用します。これには、デプロイロールに必要な IAM アクセス許可を含むブートストラップテンプレートのバージョン 12 が必要です。

使用方法

```
$ cdk import <arguments> <options>
```

引数

CDK スタック論理 ID

リソースをインポートするアプリケーションからの CDK スタックの論理 ID。この引数は、1 つのコマンドで複数回指定できます。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--change-set-name` *STRING*

作成する CloudFormation 変更セットの名前。

`--execute` *BOOLEAN*

変更セットを実行するかどうかを指定します。

デフォルト値: `true`

`--force, -f` *BOOLEAN*

デフォルトでは、テンプレート差分に更新または削除が含まれている場合、CDK はプロセスを CLI終了します。この動作`true`を上書きし、常にインポートを続行するには、 を指定します。

`--help, -h` *BOOLEAN*

コマンドの`cdk import`コマンドリファレンス情報を表示します。

`--record-resource-mapping, -r` *STRING*

このオプションを使用して、インポートされる CDK リソースへの既存の物理リソースのマッピングを生成します。マッピングは、指定したファイルパスに書き込まれます。実際のインポートオペレーションは実行されません。

`--resource-mapping, -m` *STRING*

このオプションを使用して、リソースマッピングを定義するファイルを指定します。CDK CLIはこのファイルを使用して、インタラクティブに質問するのではなく、インポートのために物理リソースをリソースにマッピングします。

このオプションはスクリプトから実行できます。

`--rollback` *BOOLEAN*

障害発生時にスタックを安定状態にロールバックします。

を指定するにはfalse、 --no-rollbackまたは を使用できます-R。

を指定するfalseと、より迅速に反復処理できます。リソース置換を含むデプロイは常に失敗します。

デフォルト値: true

`--toolkit-stack-name` *STRING*

作成する CDK Toolkit スタックの名前

cdk init

テンプレートから新しい AWS CDK プロジェクトを作成します。

使用方法

```
$ cdk init <arguments> <options>
```

引数

テンプレートのタイプ

新しい CDK プロジェクトを初期化する CDK テンプレートタイプ。

- app - CDK アプリケーションのテンプレート。
- lib – AWS コンストラクトライブラリのテンプレート。
- sample-app – いくつかのコンストラクトを含む CDK アプリケーションの例。

有効な値: app、lib、sample-app

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--generate-only` *BOOLEAN*

git リポジトリの設定、依存関係のインストール、プロジェクトのコンパイルなどの追加の操作を開始せずにプロジェクトファイルを生成するには、このオプションを指定します。

デフォルト値: false

`--help, -h` *BOOLEAN*

のコマンドリファレンス情報を表示しますcdk init command。

`--language, -l` *STRING*

新しいプロジェクトに使用する言語。このオプションは、プロジェクトcdk.jsonの設定ファイルまたはローカル開発マシン~/.cdk.jsonの で設定できます。

有効な値: csharp、fsharp、go、java、javascript、python、 typescript

`--list` *BOOLEAN*

使用可能なテンプレートタイプと言語を一覧表示します。

例

使用可能なテンプレートタイプと言語を一覧表示する

```
$ cdk init --list
Available templates:
* app: Template for a CDK Application
   ## cdk init app --language=[csharp|fsharp|go|java|javascript|python|typescript]
* lib: Template for a CDK Construct Library
   ## cdk init lib --language=typescript
* sample-app: Example CDK Application with some constructs
   ## cdk init sample-app --language=[csharp|fsharp|go|java|javascript|python|
typescript]
```

ライブラリテンプレート TypeScript から に新しい CDK アプリを作成する

```
$ cdk init lib --language=typescript
```

cdk list

CDK アプリからすべての AWS CDK スタックとその依存関係を一覧表示します。

使用方法

```
$ cdk list <arguments> <options>
```

引数

CDK スタック論理 ID

このコマンドを実行するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--help, -h` *BOOLEAN*

コマンドの`cdk list`コマンドリファレンス情報を表示します。

`--long, -l` *BOOLEAN*

各スタックの AWS 環境情報を表示します。

デフォルト値: `false`

`--show-dependencies, -d` *BOOLEAN*

各スタックのスタック依存関係情報を表示します。

デフォルト値: `false`

例

CDK アプリの「node bin/main.js」にあるすべてのスタックを一覧表示します。

```
$ cdk list --app='node bin/main.js'
Foo
Bar
Baz
```

各スタックの AWS 環境の詳細を含むすべてのスタックを一覧表示する

```
$ cdk list --app='node bin/main.js' --long
-
    name: Foo
    environment:
        name: 000000000000/bermuda-triangle-1
        account: '000000000000'
        region: bermuda-triangle-1
-
    name: Bar
    environment:
        name: 111111111111/bermuda-triangle-2
        account: '111111111111'
        region: bermuda-triangle-2
-
    name: Baz
    environment:
        name: 333333333333/bermuda-triangle-3
        account: '333333333333'
        region: bermuda-triangle-3
```

cdk metadata

CDK スタックに関連付けられたメタデータを表示します。

使用方法

```
$ cdk metadata <arguments> <options>
```

引数

CDK スタック論理 ID

メタデータを表示するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--help, -h` *BOOLEAN*

コマンドの`cdk metadata`コマンドリファレンス情報を表示します。

cdk migrate

デプロイされた AWS リソース、 AWS CloudFormation スタック、 CloudFormation テンプレートを新しい AWS CDK プロジェクトに移行します。

このコマンドは、 を使用して指定した値で という名前の単一のスタックを含む新しい CDK アプリケーションを作成します`--stack-name`。移行ソースは、`--from-scan`、、`--from-stack`または を使用して設定できます`--from-path`。

の使用の詳細については、`cdk migrate`「」を参照してください既存のリソースと AWS CloudFormation テンプレートを に移行する AWS CDK。

> **Note**
>
> `cdk migrate` コマンドは実験的であり、将来的に重大な変更が生じる可能性があります。

使用方法

```
$ cdk migrate <options>
```

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

必要なオプション

`--stack-name` *STRING*

移行後に CDK アプリ内で作成される AWS CloudFormation スタックの名前。

必須: はい

条件オプション

`--from-path` *PATH*

移行する AWS CloudFormation テンプレートへのパス。ローカルテンプレートを指定するには、このオプションを指定します。

必須: 条件的。ローカル AWS CloudFormation テンプレートから移行する場合は必須です。

`--from-scan` *STRING*

デプロイされたリソースを AWS 環境から移行する場合、このオプションを使用して、新しいスキャンを開始するか、最後に成功したスキャンを AWS CDK CLIで使用するかを指定します。

必須: 条件的。デプロイされた AWS リソースから移行する場合に必要です。

使用できる値: `most-recent`、 `new`

`--from-stack` *BOOLEAN*

デプロイされた AWS CloudFormation スタックから移行するには、このオプションを指定します。`--stack-name` を使用して、デプロイされた AWS CloudFormation スタックの名前を指定します。

必須: 条件的。デプロイされた AWS CloudFormation スタックから移行する場合に必要です。

オプション

`--account` *STRING*

AWS CloudFormation スタックテンプレートを取得するアカウント。

必須: いいえ

デフォルト : AWS CDK CLIはデフォルトのソースからアカウント情報を取得します。

`--compress` *BOOLEAN*

生成された CDK プロジェクトをZIPファイルに圧縮するには、このオプションを指定します。

必須: いいえ

`--filter` *ARRAY*

AWS アカウントと からデプロイされたリソースを移行するときに を使用します AWS リージョン。このオプションは、移行するデプロイされたリソースを決定するフィルターを指定します。

このオプションは、キーと値のペアの配列を受け入れます。ここで、キーはフィルタータイプを表し、値はフィルターする値を表します。

使用できるキーは次のとおりです。

- `resource-identifier` – リソースの識別子。値は、リソースの論理 ID または物理 ID にすることができます。例えば `resource-identifier="ClusterName"` です。
- `resource-type-prefix` – AWS CloudFormation リソースタイプのプレフィックス。例えば、 を指定`resource-type-prefix="AWS::DynamoDB::"`して、すべての Amazon DynamoDB リソースをフィルタリングします。
- `tag-key` – リソースタグのキー。例えば `tag-key="myTagKey"` です。
- `tag-value` – リソースタグの値。例えば `tag-value="myTagValue"` です。

AND 条件付きロジックに複数のキーと値のペアを指定します。次の例では、タグキーmyTagKeyとして でタグ付けされている DynamoDB リソースをフィルタリングします: `--filter resource-type-prefix="AWS::DynamoDB::", tag-key="myTagKey"`。

OR 条件付きロジックの オプションを 1 つのコマンドで`--filter`複数回指定します。次の例では、DynamoDB リソースであるか、タグキー myTagKey として でタグ付けされているリソースをフィルタリングします`--filter resource-type-prefix="AWS::DynamoDB::" --filter tag-key="myTagKey"`。

必須: いいえ

`--help, -h` *BOOLEAN*

コマンドのcdk migrateコマンドリファレンス情報を表示します。

`--language` *STRING*

移行中に作成された CDK プロジェクトに使用するプログラミング言語。

必須: いいえ

有効な値: typescript、python、java、csharp、go。

デフォルト: typescript

`--output-path` *PATH*

移行された CDK プロジェクトの出力パス。

必須: いいえ

デフォルト： デフォルトでは、 は AWS CDK CLI現在の作業ディレクトリを使用します。

`--region` *STRING*

AWS CloudFormation スタックテンプレート AWS リージョン を取得する 。

必須: いいえ

デフォルト : AWS CDK CLIはデフォルトのソースから AWS リージョン 情報を取得します。

例

CloudFormation スタックからの移行の簡単な例

を使用して、特定の AWS 環境でデプロイされた CloudFormation スタックから移行します--from-stack。新しい CDK スタック--stack-nameに名前を付けるには、 を指定します。以下は、 を使用しているmyCloudFormationStack新しい CDK アプリに移行する例ですTypeScript。

```
$ cdk migrate --language typescript --from-stack --stack-name 'myCloudFormationStack'
```

ローカル CloudFormation テンプレートからの移行の簡単な例

を使用してローカル JSON または YAML CloudFormation テンプレートから移行します`--from-path`。新しい CDK スタック`--stack-name`に名前を付けるには、 を指定します。以下は、ローカル`template.json`ファイルからの`myCloudFormationStack`スタックを含む新しい CDK TypeScript アプリを に作成する例です。

```
$ cdk migrate --stack-name "myCloudFormationStack" --language typescript --from-path
 "./template.json"
```

デプロイされた AWS リソースからの移行の簡単な例

を使用して、 CloudFormation スタックに関連付けられていない特定の AWS 環境からデプロイされた AWS リソースを移行します`--from-scan`。CDK は IaC generatorサービスCLIを利用してリソースをスキャンし、テンプレートを生成します。次に、CDK はテンプレートCLIを参照して新しい CDK アプリを作成します。以下は、移行された AWS リソースを含む新しい`myCloudFormationStack`スタックTypeScriptを使用して に新しい CDK アプリケーションを作成する例です。

```
$ cdk migrate --language typescript --from-scan --stack-name "myCloudFormationStack"
```

cdk notices

CDK アプリケーションの通知を表示します。

通知には、セキュリティの脆弱性、リグレッション、サポートされていないバージョンの使用に関する重要なメッセージが含まれる場合があります。

このコマンドは、承認されているかどうかにかかわらず、関連する通知を表示します。関連する通知は、デフォルトですべてのコマンドの後に表示される場合もあります。

通知は、次の方法で抑制できます。

- コマンドオプション経由。以下に例を示します。

  ```
  $ cdk deploy --no-notices
  ```

- プロジェクトの `cdk.json` ファイル内のコンテキストを使用して、すべての通知を無期限に非表示にします。

```
{
  "notices": false,
  "context": {
    // ...
  }
}
```

- `cdk acknowledge` コマンドを使用して各通知を承認します。

使用方法

```
$ cdk notices <options>
```

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--help, -h` *BOOLEAN*

コマンドの`cdk notices`コマンドリファレンス情報を表示します。

例

cdk deploy コマンドの実行後に表示されるデフォルトの通知の例

```
$ cdk deploy

... # Normal output of the command

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: If a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.
```

```
        Affected versions: <1.126.0.

        More information at: https://github.com/aws/aws-cdk/issues/16603


17061   Error when building EKS cluster with monocdk import

        Overview: When using monocdk/aws-eks to build a stack containing
                  an EKS cluster, error is thrown about missing
                  lambda-layer-node-proxy-agent/layer/package.json.

        Affected versions: >=1.126.0 <=1.130.0.

        More information at: https://github.com/aws/aws-cdk/issues/17061


If you don't want to see an notice anymore, use "cdk acknowledge ID". For example, "cdk
 acknowledge 16603"
```

cdk notices コマンドを実行する簡単な例

```
$ cdk notices

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: if a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.

        Affected versions: framework: <=2.15.0 >=2.10.0

        More information at: https://github.com/aws/aws-cdk/issues/16603


If you don't want to see a notice anymore, use "cdk acknowledge <id>". For example,
 "cdk acknowledge 16603"
```

cdk synthesize

CDK アプリを合成して、各スタックの AWS CloudFormation テンプレートを含むクラウドアセンブリを作成します。

クラウドアセンブリは、アプリケーションを AWS 環境にデプロイするために必要なものをすべて含むファイルです。例えば、アプリ内の各スタックの CloudFormation テンプレートと、アプリで参照するファイルアセットまたはDockerイメージのコピーが含まれます。

アプリケーションに 1 つのスタックが含まれている場合、または 1 つのスタックが引数として指定されている場合、 CloudFormation テンプレートは標準出力 (stdout) にも YAML 形式で表示されます。

アプリケーションに複数のスタックが含まれている場合、 `cdk synth`はクラウドアセンブリを に合成しますcdk.out。

使用方法

```
$ cdk synthesize <arguments> <options>
```

引数

CDK スタック論理 ID

合成するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--exclusively, -e` *BOOLEAN*

リクエストされたスタックのみを合成し、依存関係を含めないでください。

`--help, -h` *BOOLEAN*

コマンドのcdk synthesizeコマンドリファレンス情報を表示します。

`--quiet, -q` *BOOLEAN*

CloudFormation テンプレートを に出力しないでくださいstdout。

このオプションは CDK プロジェクトの cdk.json ファイルで設定できます。以下に例を示します。

```
{
   "quiet": true
}
```

デフォルト値: `false`

`--validation` *BOOLEAN*

追加のチェックを実行して、合成後に生成された CloudFormation テンプレートを検証します。

このオプションは、 validateOnSynth 属性またはCDK_VALIDATION環境変数を使用して設定することもできます。

デフォルト値: `true`

例

ログ ID を使用して CDK スタックのクラウドアセンブリを合成 MyStackName し、CloudFormation テンプレートを stdout に出力します。

```
$ cdk synth MyStackName
```

CDK アプリ内のすべてのスタックのクラウドアセンブリを合成し、cdk.out に保存します。

```
$ cdk synth
```

のクラウドアセンブリを合成しますが MyStackName、依存関係は含まれません

```
$ cdk synth MyStackName --exclusively
```

のクラウドアセンブリを合成しますが MyStackName、 CloudFormation テンプレートを stdout に出力しないでください。

```
$ cdk synth MyStackName --quiet
```

cdk watch

ローカル AWS CDK プロジェクトにデプロイとホットスワップを実行するための変更がないか継続的に監視します。

このコマンドは に似ていますがcdk deploy、1 つのコマンドで継続的なデプロイとホットスワップを実行できます。

このコマンドは のショートカットですcdk deploy --watch。

cdk watch セッションを終了するには、 を押してプロセスを中断しますCtrl+C。

観察されるファイルは、cdk.jsonファイル内の "watch"設定によって決まります。1 つの文字列または文字列の配列"exclude"を受け入れる "include"と の 2 つのサブキーがあります。各エントリは、cdk.jsonファイルの場所を基準にしたパスとして解釈されます。* と の両方**が受け入れられます。

cdk init コマンドを使用してプロジェクトを作成する場合、プロジェクトの cdk.json ファイルcdk watchで に対して次のデフォルトの動作が設定されます。

- "include" は に設定され"**/*"、プロジェクトのルート内のすべてのファイルとディレクトリが含まれます。
- "exclude" はオプションです。ただし、デフォルトで既に無視されているファイルとフォルダを除きます。これは、、CDK 出力ディレクトリ.、および ディレクトリで始まるファイルとnode_modulesディレクトリで構成されます。

設定する最小設定は watchです"watch": {}。

CDK コードまたはアプリケーションコードのいずれかがデプロイ前にビルドステップを必要とする場合、 cdk watch は cdk.json ファイル内の "build" キーを使用します。

> **Note**
>
> このコマンドは実験的と見なされ、将来的に重大な変更が生じる可能性があります。

にも同じ制限cdk deploy --hotswapが適用されますcdk watch。詳細については、「cdk deploy --hotswap」を参照してください。

使用方法

```
$ cdk watch <arguments> <options>
```

引数

CDK スタック論理 ID

監視するアプリケーションからの CDK スタックの論理 ID。

タイプ: 文字列

必須: いいえ

オプション

すべての CDK CLI コマンドで動作するグローバルオプションのリストについては、「」を参照してくださいグローバルオプション。

`--build-exclude, -E` *ARRAY*

指定された ID でアセットを再構築しないでください。

このオプションは、1 つのコマンドで複数回指定できます。

デフォルト値: []

`--change-set-name` *STRING*

作成する CloudFormation 変更セットの名前。

`--concurrency` *NUMBER*

スタック間の依存関係を考慮しながら、複数のスタックを並行してデプロイしてホットスワップします。デプロイを高速化するには、このオプションを使用します。その他の AWS アカウントレート制限 CloudFormation も考慮する必要があります。

実行する同時デプロイの最大数 (依存関係の許可) を指定する数値を指定します。

デフォルト値: `1`

`--exclusively, -e` *BOOLEAN*

リクエストされたスタックのみをデプロイし、依存関係は含まれません。

`--force, -f` *BOOLEAN*

テンプレートが同じであっても、常にスタックをデプロイします。

デフォルト値: `false`

`--help, -h` *BOOLEAN*

コマンドの`cdk watch`コマンドリファレンス情報を表示します。

`--hotswap` *BOOLEAN*

デフォルトでは、 `cdk watch`は可能な場合はホットスワップデプロイを使用してリソースを更新します。CDK CLIはホットスワップデプロイの実行を試み、失敗してもフル CloudFormation デプロイにフォールバックしません。ホットスワップで更新できない変更は無視されます。

デフォルト値: `true`

`--hotswap-fallback` *BOOLEAN*

デフォルトでは、 はホットスワップデプロイの実行`cdk watch`を試み、 CloudFormation デプロイを必要とする変更を無視します。ホットスワップ CloudFormation デプロイが失敗した場合に`--hotswap-fallback`フォールバックして完全なデプロイを実行するには、 を指定します。

`--logs` *BOOLEAN*

デフォルトでは、 はアプリケーション内のすべての CloudWatch ロググループを`cdk watch`モニタリングし、ログイベントを にローカルにストリーミングします`stdout`。

デフォルト値: `true`

`--progress` *STRING*

CDK がデプロイの進行状況CLIを表示する方法を設定します。

- `bar` — スタックデプロイイベントを進行状況バーとして表示し、リソースのイベントは現在デプロイされています。
- `events` – すべての CloudFormation イベントを含む完全な履歴を提供します。

このオプションは、プロジェクトの `cdk.json` ファイルまたはローカル開発マシンの `~/.cdk.json` で設定することもできます。

```
{
   "progress": "events"
}
```

有効な値:`bar`、`events`|

デフォルト値: `bar`

`--rollback` *BOOLEAN*

デプロイ中にリソースの作成または更新に失敗した場合、デプロイは CDK がCLI戻る前に最新の安定状態にロールバックされます。その時点までに行われたすべての変更は元に戻されます。作成されたリソースは削除され、更新はロールバックされます。

`--no-rollback` または `-R`を使用して、この動作を無効にします。リソースの作成または更新に失敗した場合、CDK CLI はその時点までに行われた変更をそのままにして、 を返します。これは、すぐに反復する開発環境で役立つ場合があります。

> **Note**
>
> の場合`false`、リソースの置換を引き起こすデプロイは常に失敗します。この値は、新しいリソースを更新または作成するデプロイにのみ使用できます。

デフォルト値: `true`

`--toolkit-stack-name` *STRING*

既存の CDK Toolkit スタックの名前。

このオプションは、レガシー合成を使用する CDK アプリケーションにのみ使用されます。

例

論理 ID を持つ CDK スタックの変更 DevelopmentStack を監視する

```
$ cdk watch DevelopmentStack
Detected change to 'lambda-code/index.js' (type: change). Triggering 'cdk deploy'
DevelopmentStack: deploying...

 #  DevelopmentStack
```

変更をモニタリングする対象と除外する対象について cdk.json ファイルを設定する

```
{
   "app": "mvn -e -q compile exec:java",
   "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
   }
}
```

cdk.json ファイルを設定して、デプロイJava前に を使用して CDK プロジェクトを構築する

```
{
  "app": "mvn -e -q exec:java",
  "build": "mvn package",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
}
```

AWS CDK リファレンス

このセクションでは、 AWS Cloud Development Kit (AWS CDK)に関するリファレンス情報を紹介します。

トピック

- API リファレンス
- AWS CDK バージョニング

API リファレンス

API リファレンスには、 AWS コンストラクトライブラリと、 が提供するその他の APIs に関する情報が含まれています AWS Cloud Development Kit (AWS CDK)。 AWS コンストラクトライブラリのほとんどは、 というTypeScript名前の 1 つのパッケージに含まれていますaws-cdk-lib。実際のパッケージ名は言語によって異なります。サポートされているプログラミング言語ごとに、API リファレンスの個別のバージョンが提供されます。

CDK API リファレンスはサブモジュールに整理されています。ごとに 1 つ以上のサブモジュールがあります AWS のサービス。

各サブモジュールには、APIs。例えば、S3 の概要は、Amazon Simple Storage Service (Amazon S3) バケットにデフォルトの暗号化を設定する方法を示しています。

AWS CDK バージョニング

このトピックでは、 がバージョニング AWS Cloud Development Kit (AWS CDK) を処理する方法に関するリファレンス情報を提供します。

バージョン番号は、メジャー .minor .patch の 3 つの数値バージョン部分で構成され、セマンティックバージョニングモデルに厳密に準拠しています。つまり、安定した APIsされます。

マイナーリリースとパッチリリースには下位互換性があります。同じメジャーバージョンを持つ以前のバージョンで記述されたコードは、同じメジャーバージョン内の新しいバージョンにアップグレードできます。また、ビルドと実行が継続され、同じ出力が生成されます。

トピック

AWS CDKCLI 互換性

AWS CDK CLI は常に、意味的に下位または等しいバージョン番号のコンストラクトライブラリと互換性があります。したがって、同じメジャーバージョン内で常に安全に を AWS CDK CLIアップグレードできます。

AWS CDK CLI は、意味的に上位のバージョンのコンストラクトライブラリと常に互換性があるわけではありません。互換性は、同じクラウドアセンブリスキーマバージョンが 2 つのコンポーネントで使用されているかどうかによって異なります。 AWS CDK フレームワークは、合成中にクラウドアセンブリを生成し、 はそれをデプロイに AWS CDK CLI消費します。クラウドアセンブリの形式を定義するスキーマは厳密に指定され、バージョン管理されます。

AWS 特定のクラウドアセンブリスキーマバージョンを使用してライブラリを構築することは、そのスキーマバージョン 以降を使用するバージョンと AWS CDK CLI互換性があります。これには、AWS CDK CLI特定のコンストラクトライブラリリリースより前の のリリースが含まれる場合があります。

コンストラクトライブラリに必要なクラウドアセンブリのバージョンが、 でサポートされているバージョンと互換性がない場合 AWS CDK CLI、次のようなエラーメッセージが表示されます。

```
Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but
 found 4.0.0.
    Please upgrade your CLI in order to interact with this app.
```

このエラーを AWS CDK CLI解決するには、 を必要なクラウドアセンブリバージョンと互換性のあるバージョンに更新するか、利用可能な最新バージョンに更新します。代替方法 (アプリケーションが使用するコンストラクトライブラリモジュールをダウングレードする) は、一般的にお勧めしません。

> **Note**
>
> クラウドアセンブリスキーマの詳細については、「Cloud Assembly Versioning」を参照してください。

AWS ライブラリのバージョニングを構築する

AWS Construct Library のモジュールは、概念から成熟した API に進化するにつれて、さまざまな段階を移動します。ステージによって、 の後続のバージョンでさまざまな程度の API 安定性が提供されます AWS CDK。

メイン AWS CDK ライブラリ の APIs `aws-cdk-lib`は安定しており、ライブラリは完全に意味的にバージョン管理されています。このパッケージには、すべての AWS サービスと、安定しているすべての上位レベル AWS CloudFormation (L2 および L3) モジュール用の (L1) コンストラクトが含まれています。(`App`や などのコア CDK クラスも含まれていますStack) 。APIs は、CDK の次のメジャーリリースまで、このパッケージから削除されません (ただし、廃止される可能性があります) 。個々の API に重大な変更が加えられることはありません。重大な変更が必要な場合は、まったく新しい API が追加されます。

既に組み込まれているサービスの開発中の新しい APIs `aws-cdk-lib`は、 `Beta`*`N`* サフィックスを使用して識別されます。 は 1 からN始まり、新しい API への重大な変更があるたびに増加します。`Beta`*`N`*APIs は削除されることはなく、非推奨にすぎないため、既存のアプリは引き続き の新しいバージョンで動作しますaws-cdk-lib。API が安定していると見なされると、 サフィックスのない新しい API `Beta`*`N`* が追加されます。

以前は L1 APIs しか持っていなかった AWS サービスに対して上位レベルの (L2 または L3) APIsの開発が開始されると、これらの APIsは最初は別のパッケージに配布されます。このようなパッケージの名前には「アルファ」サフィックスがあり、そのバージョンは、 `alpha`サブバージョンと互換性のある最初のバージョンと一致しaws-cdk-libます。モジュールが意図したユースケースをサポートすると、その APIsが に追加されますaws-cdk-lib。

言語バインディングの安定性

時間の経過とともに、追加のプログラミング言語 AWS CDK のサポートが に追加される可能性があります。すべての言語で説明されている API は同じですが、API の表現方法は言語によって異なり、言語サポートの進化に伴って変更される可能性があります。このため、言語バインディングは、本番環境での使用準備が整っていると見なされるまで、しばらく実験的と見なされます。

Language	Stability
TypeScript	Stable
JavaScript	Stable

Language	Stability
Python	Stable
Java	Stable
C#/.NET	Stable
Go	Stable

AWS CDK チュートリアル

このセクションには、のチュートリアルが含まれています。 AWS Cloud Development Kit (AWS CDK)

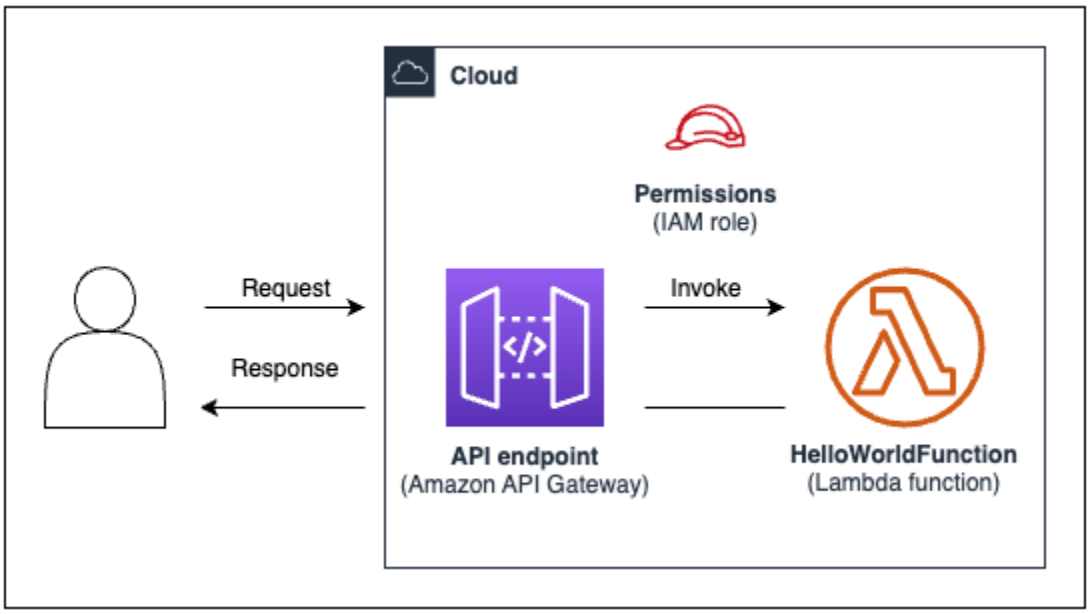
トピック

サーバーレス Hello World アプリケーションを作成する

このチュートリアルでは、 を使用して、以下で構成される基本的な API バックエンドを実装するシンプルなサーバーレスHello Worldアプリケーション AWS Cloud Development Kit (AWS CDK) を作成します。

- Amazon API Gateway REST API – HTTP GETリクエストを通じて関数を呼び出すために使用される HTTP エンドポイントを提供します。
- AWS Lambda 関数 – HTTPエンドポイントで呼び出されたときに`Hello World!`メッセージを返す関数。
- 統合とアクセス許可 – リソースが相互にやり取りし、Amazon への ログの書き込みなどのアクションを実行するための設定の詳細とアクセス許可 CloudWatch。

以下の図は、このアプリケーションのコンポーネントを示しています。



このチュートリアルでは、次のステップでアプリケーションを作成して操作します。

1. AWS CDK プロジェクトを作成します。
2. コンストラクトライブラリの L2 コンストラクトを使用して、Lambda 関数と API Gateway REST API AWS を定義します。
3. アプリケーションを にデプロイします AWS クラウド。
4. でアプリケーションとやり取りします AWS クラウド。
5. AWS クラウドからサンプルアプリケーションを削除します。

トピック

前提条件

このチュートリアルをスタートする前に、以下を完了してください。

- AWS アカウント を作成し、 AWS Command Line Interface （ AWS CLI) をインストールして設定します。
- Node.js と をインストールしますnpm。
- を使用して CDK Toolkit をグローバルにインストールしますnpm install -g aws-cdk。

詳細については、「の開始方法 AWS CDK」を参照してください。

また、以下の基本的な理解もお勧めします。

- は何ですか AWS CDK? の基本的な概要については、「」を参照してください AWS CDK。
- AWS CDK 主要概念を学ぶ では、 のコアコンセプトの概要について説明します AWS CDK。

ステップ 1: CDK プロジェクトを作成する

このステップでは、 cdk init コマンドを使用して新しい CDK AWS CDK CLIプロジェクトを作成します。

CDK プロジェクトを作成するには

1. 選択した開始ディレクトリから、マシンcdk-hello-worldに という名前のプロジェクトディレクトリを作成して移動します。

   ```
   $ mkdir cdk-hello-world && cd cdk-hello-world
   ```

2. cdk init コマンドを使用して、任意のプログラミング言語で新しいプロジェクトを作成します。

   TypeScript

   ```
   $ cdk init --language typescript
   ```

   AWS CDK ライブラリをインストールします。

   ```
   $ npm install aws-cdk-lib constructs
   ```

JavaScript

```
$ cdk init --language javascript
```

AWS CDK ライブラリをインストールします。

```
$ npm install aws-cdk-lib constructs
```

Python

```
$ cdk init --language python
```

仮想環境をアクティブ化します。

```
$ source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
```

AWS CDK ライブラリとプロジェクトの依存関係をインストールします。

```
(.venv)$ python3 -m pip install -r requirements.txt
```

Java

```
$ cdk init --language java
```

AWS CDK ライブラリとプロジェクトの依存関係をインストールします。

```
$ mvn package
```

C#

```
$ cdk init --language csharp
```

AWS CDK ライブラリとプロジェクトの依存関係をインストールします。

```
$ dotnet restore src
```

Go

```
$ cdk init --language go
```

プロジェクトの依存関係をインストールします。

```
$ go get github.com/aws/aws-cdk-go/awscdk/v2
$ go get github.com/aws/aws-cdk-go/awscdk/v2/awslambda
$ go get github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway
$ go mod tidy
```

CDK は、次の構造でプロジェクトCLIを作成します。

TypeScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.ts
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### cdk-hello-world.test.ts
### tsconfig.json
```

JavaScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
```

```
### bin
#   ### cdk-hello-world.js
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### cdk-hello-world.test.js
```

Python

```
cdk-hello-world
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### cdk_hello_world
#   ### __init__.py
#   ### cdk_hello_world_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
```

Java

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
#   ### main
#   #   ### java
#   #       ### com
#   #           ### myorg
#   #               ### CdkHelloWorldApp.java
```

```
#   #               ### CdkHelloWorldStack.java
### target
```

C#

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### src
    ### CdkHelloWorld
    #   ### CdkHelloWorld.csproj
    #   ### CdkHelloWorldStack.cs
    #   ### GlobalSuppressions.cs
    #   ### Program.cs
    ### CdkHelloWorld.sln
```

Go

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk-hello-world.go
### cdk-hello-world_test.go
### cdk.json
### go.mod
### go.sum
```

CDK は、単一のスタックを含む CDK アプリCLIを自動的に作成します。CDK アプリケーションインスタンスは App クラスから作成されます。CDK アプリケーションファイルの一部を次に示します。

TypeScript

にありますbin/cdk-hello-world.ts。

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { CdkHelloWorldStack } from '../lib/cdk-hello-world-stack';
```

```
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

JavaScript

にありますbin/cdk-hello-world.js。

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { CdkHelloWorldStack } = require('../lib/cdk-hello-world-stack');
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

Python

にありますapp.py。

```
#!/usr/bin/env python3
import os
import aws_cdk as cdk
from cdk_hello_world.cdk_hello_world_stack import CdkHelloWorldStack

app = cdk.App()
CdkHelloWorldStack(app, "CdkHelloWorldStack",)
app.synth()
```

Java

にありますsrc/main/java/.../CdkHelloWorldApp.java。

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class JavaApp {
```

```
    public static void main(final String[] args) {
        App app = new App();

        new JavaStack(app, "JavaStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

にありますsrc/CdkHelloWorld/Program.cs。

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CdkHelloWorld
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new CdkHelloWorldStack(app, "CdkHelloWorldStack", new StackProps
            {

            });
            app.Synth();
        }
    }
}
```

Go

にありますcdk-hello-world.go。

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
```

```
    "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
    defer jsii.Close()
    app := awscdk.NewApp(nil)
    NewCdkHelloWorldStack(app, "CdkHelloWorldStack", &CdkHelloWorldStackProps{
        awscdk.StackProps{
            Env: env(),
        },
    })
    app.Synth(nil)
}

func env() *awscdk.Environment {
    return nil
}
```

ステップ 2: Lambda 関数を作成する

CDK プロジェクト内で、新しいhello.jsファイルを含むlambdaディレクトリを作成します。以下に例を示します。

TypeScript

プロジェクトのルートから、以下を実行します。

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### lambda
    ### hello.js
```

JavaScript

プロジェクトのルートから、以下を実行します。

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### lambda
    ### hello.js
```

Python

プロジェクトのルートから、以下を実行します。

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### lambda
    ### hello.js
```

Java

プロジェクトのルートから、以下を実行します。

```
$ mkdir -p src/main/resources/lambda
$ cd src/main/resources/lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### src
    ### main
        ###resources
            ###lambda
                ###hello.js
```

C#

プロジェクトのルートから、以下を実行します。

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### lambda
    ### hello.js
```

Go

プロジェクトのルートから、以下を実行します。

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

これで、CDK プロジェクトに以下が追加されます。

```
cdk-hello-world
### lambda
    ### hello.js
```

> **Note**
>
> このチュートリアルを簡単にするために、すべての CDK プログラミング言語に JavaScript Lambda 関数を使用します。

新しく作成されたファイルに以下を追加して、Lambda 関数を定義します。

```
exports.handler = async (event) => {
    return {
        statusCode: 200,
        headers: { "Content-Type": "text/plain" },
        body: JSON.stringify({ message: "Hello, World!" }),
    };
```

```
};
```

ステップ 3: コンストラクトを定義する

このステップでは、 AWS CDK L2 コンストラクトを使用して Lambda リソースと API Gateway リソースを定義します。

CDK スタックを定義するプロジェクトファイルを開きます。このファイルを変更してコンストラクトを定義します。開始スタックファイルの例を次に示します。

TypeScript

にありますlib/cdk-hello-world-stack.ts。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Your constructs will go here

  }
}
```

JavaScript

にありますlib/cdk-hello-world-stack.js。

```
const { Stack, Duration } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigateway = require('aws-cdk-lib/aws-apigateway');

class CdkHelloWorldStack extends Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    // Your constructs will go here

  }
```

```
}

module.exports = { CdkHelloWorldStack }
```

Python

にありますcdk_hello_world/cdk_hello_world_stack.py。

```
from aws_cdk import Stack
from constructs import Construct

class CdkHelloWorldStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

          // Your constructs will go here
```

Java

にありますsrc/main/java/.../CdkHelloWorldStack.java。

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Your constructs will go here
    }
}
```

C#

にありますsrc/CdkHelloWorld/CdkHelloWorldStack.cs。

```
using Amazon.CDK;
using Constructs;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Your constructs will go here
        }
    }
}
```

Go

の場所cdk-hello-world.go :

```
package main

import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

type CdkHelloWorldStackProps struct {
    awscdk.StackProps
}

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Your constructs will go here

    return stack
}
```

```
func main() {

    // ...

}

func env() *awscdk.Environment {

    return nil

}
```

このファイルでは、 AWS CDK は以下を実行します。

- CDK スタックインスタンスは `Stack` クラスからインスタンス化されます。
- `Constructs` ベースクラスはインポートされ、スタックインスタンスのスコープまたは親として提供されます。

Lambda 関数リソースを定義する

Lambda 関数リソースを定義するには、コンストラクトライブラリから `aws-lambda` L2 AWS コンストラクトをインポートして使用します。

スタックファイルを次のように変更します。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// Import Lambda L2 construct
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
```

```
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
// Import Lambda L2 construct
const lambda = require('aws-cdk-lib/aws-lambda');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}

module.exports = { CdkHelloWorldStack }
```

Python

```
from aws_cdk import (
    Stack,
    # Import Lambda L2 construct
    aws_lambda as _lambda,
)
# ...

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)
```

```
        # Define the Lambda function resource
        hello_world_function = _lambda.Function(
            self,
            "HelloWorldFunction",
            runtime = _lambda.Runtime.NODEJS_20_X, # Choose any supported Node.js
 runtime
            code = _lambda.Code.from_asset("lambda"), # Points to the lambda
 directory
            handler = "hello.handler", # Points to the 'hello' file in the lambda
 directory
        )
```

> **Note**
>
> lambda は の組み込み識別子_lambdaであるため、aws_lambdaモジュールは としてインポートされますPython。

Java

```
// ...
// Import Lambda L2 construct
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Define the Lambda function resource
        Function helloWorldFunction = Function.Builder.create(this,
 "HelloWorldFunction")
                .runtime(Runtime.NODEJS_20_X)  // Choose any supported Node.js
 runtime
                .code(Code.fromAsset("src/main/resources/lambda")) // Points to the
 lambda directory
```

```
                .handler("hello.handler")  // Points to the 'hello' file in the
 lambda directory
                .build();
    }
}
```

C#

```
// ...
// Import Lambda L2 construct
using Amazon.CDK.AWS.Lambda;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define the Lambda function resource
            var helloWorldFunction = new Function(this, "HelloWorldFunction", new
 FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X, // Choose any supported Node.js
 runtime
                Code = Code.FromAsset("lambda"), // Points to the lambda directory
                Handler = "hello.handler" // Points to the 'hello' file in the
 lambda directory
            });
        }
    }
}
```

Go

```
package main

import (
    // ...
    // Import Lambda L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
    // Import S3 assets construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
```

```
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    helloWorldFunction := awslambda.NewFunction(stack,
 jsii.String("HelloWorldFunction"), &awslambda.FunctionProps{
        Runtime: awslambda.Runtime_NODEJS_20_X(), // Choose any supported Node.js
 runtime
        Code:    awslambda.Code_FromAsset(jsii.String("lambda"),
 &awss3assets.AssetOptions{}), // Points to the lambda directory
        Handler: jsii.String("hello.handler"), // Points to the 'hello' file in the
 lambda directory
    })

    return stack
}

// ...
```

ここでは、Lambda 関数リソースを作成し、次のプロパティを定義します。

- `runtime` – 関数が実行される環境。ここでは、Node.jsバージョン を使用します20.x。
- `code` – ローカルマシン上の関数コードへのパス。
- `handler` – 関数コードを含む特定のファイルの名前。

API Gateway REST APIリソースを定義する

API Gateway REST APIリソースを定義するには、コンストラクトライブラリから [aws-apigateway](#) L2 AWS コンストラクトをインポートして使用します。

スタックファイルを次のように変更します。

TypeScript

```
// ...
//Import API Gateway L2 construct
import * as apigateway from 'aws-cdk-lib/aws-apigateway';


export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });

    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  }
}
```

JavaScript

```
// ...
// Import API Gateway L2 construct
const apigateway = require('aws-cdk-lib/aws-apigateway');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });
```

```
    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  };
};

// ...
```

Python

```
from aws_cdk import (
    # ...
    # Import API Gateway L2 construct
    aws_apigateway as apigateway,
)
from constructs import Construct

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # ...

        # Define the API Gateway resource
        api = apigateway.LambdaRestApi(
            self,
            "HelloWorldApi",
            handler = hello_world_function,
            proxy = False,
        )

        # Define the '/hello' resource with a GET method
        hello_resource = api.root.add_resource("hello")
        hello_resource.add_method("GET")
```

Java

```
// ...
// Import API Gateway L2 construct
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.apigateway.Resource;
```

```
public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // ...

        // Define the API Gateway resource
        LambdaRestApi api = LambdaRestApi.Builder.create(this, "HelloWorldApi")
                .handler(helloWorldFunction)
                .proxy(false) // Turn off default proxy integration
                .build();

        // Define the '/hello' resource and its GET method
        Resource helloResource = api.getRoot().addResource("hello");
        helloResource.addMethod("GET");
    }
}
```

C#

```
// ...
// Import API Gateway L2 construct
using Amazon.CDK.AWS.APIGateway;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
           // ...

            // Define the API Gateway resource
            var api = new LambdaRestApi(this, "HelloWorldApi", new
 LambdaRestApiProps
            {
```

```
                Handler = helloWorldFunction,
                Proxy = false
            });

            // Add a '/hello' resource with a GET method
            var helloResource = api.Root.AddResource("hello");
            helloResource.AddMethod("GET");
        }
    }
}
```

Go

```
// ...

import (
    // ...
    // Import Api Gateway L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    // ...

    // Define the API Gateway resource
    api := awsapigateway.NewLambdaRestApi(stack, jsii.String("HelloWorldApi"),
 &awsapigateway.LambdaRestApiProps{
        Handler: helloWorldFunction,
        Proxy: jsii.Bool(false),
    })

    // Add a '/hello' resource with a GET method
```

```
    helloResource := api.Root().AddResource(jsii.String("hello"),
 &awsapigateway.ResourceOptions{})
    helloResource.AddMethod(jsii.String("GET"),
 awsapigateway.NewLambdaIntegration(helloWorldFunction,
 &awsapigateway.LambdaIntegrationOptions{}), &awsapigateway.MethodOptions{})

    return stack
}

// ...
```

ここでは、API Gateway REST APIリソースと以下を作成します。

- REST API と Lambda 関数の統合。API が関数を呼び出せるようにします。これには、Lambda アクセス許可リソースの作成が含まれます。
- API エンドポイントのルートに追加helloされる という名前の新しいリソースまたはパス。これにより、 をベース /helloに追加する新しいエンドポイントが作成されますURL。
- hello リソースの GET メソッド。GET リクエストが/helloエンドポイントに送信されると、Lambda 関数が呼び出され、そのレスポンスが返されます。

ステップ 4: アプリケーションをデプロイ用に準備する

このステップでは、必要に応じて、 cdk synth コマンドを使用して基本的な検証を構築して、アプリケーションをデプロイ用に準備します AWS CDK CLI。

必要に応じて、アプリケーションを構築します。

TypeScript

プロジェクトのルートから、以下を実行します。

```
$ npm run build
```

JavaScript

構築は必要ありません。

Python

構築は必要ありません。

Java

プロジェクトのルートから、以下を実行します。

```
$ mvn package
```

C#

プロジェクトのルートから、以下を実行します。

```
$ dotnet build src
```

Go

構築は必要ありません。

`cdk synth` を実行して CDK コードから AWS CloudFormation テンプレートを合成します。L2 コンストラクトを使用すると、Lambda 関数 と 間のやり取りを容易に AWS CloudFormation するためにが必要とする設定の詳細の多くはREST API、 によってプロビジョニングされます AWS CDK。

プロジェクトのルートから、以下を実行します。

```
$ cdk synth
```

> **Note**
>
> 次のようなエラーが表示された場合は、 `cdk-hello-world` ディレクトリにあることを確認し、もう一度試してください。
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

成功すると AWS CDK CLI、 はコマンドプロンプトで AWS CloudFormation テンプレートを YAML 形式で出力します。フォーマットJSONされたテンプレートも `cdk.out` ディレクトリに保存されます。

テンプレートの出力例を次に示します AWS CloudFormation 。

AWS CloudFormation テンプレート

```
Resources:
  HelloWorldFunctionServiceRoleunique-identifier:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/ServiceRole/Resource
  HelloWorldFunctionunique-identifier:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-unique-identifier-assets-${AWS::AccountId}-${AWS::Region}
        S3Key: unique-identifier.zip
      Handler: hello.handler
      Role:
        Fn::GetAtt:
          - HelloWorldFunctionServiceRoleunique-identifier
          - Arn
      Runtime: nodejs20.x
    DependsOn:
      - HelloWorldFunctionServiceRoleunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/Resource
      aws:asset:path: asset.unique-identifier
      aws:asset:is-bundled: false
      aws:asset:property: Code
  HelloWorldApiunique-identifier:
    Type: AWS::ApiGateway::RestApi
    Properties:
```

```
      Name: HelloWorldApi
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Resource
  HelloWorldApiDeployment<unique-identifier>:
    Type: AWS::ApiGateway::Deployment
    Properties:
      Description: Automatically created by the RestApi construct
      RestApiId:
        Ref: HelloWorldApi<unique-identifier>
    DependsOn:
      - HelloWorldApihelloGET<unique-identifier>
      - HelloWorldApihello<unique-identifier>
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Deployment/Resource
  HelloWorldApiDeploymentStageprod<012345ABC>:
    Type: AWS::ApiGateway::Stage
    Properties:
      DeploymentId:
        Ref: HelloWorldApiDeployment<unique-identifier>
      RestApiId:
        Ref: HelloWorldApi<unique-identifier>
      StageName: prod
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/DeploymentStage.prod/Resource
  HelloWorldApihello<unique-identifier>:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId:
        Fn::GetAtt:
          - HelloWorldApi<unique-identifier>
          - RootResourceId
      PathPart: hello
      RestApiId:
        Ref: HelloWorldApi<unique-identifier>
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/Resource
  HelloWorldApihelloGETApiPermissionCdkHelloWorldStackHelloWorldApi<unique-identifier>:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunction<unique-identifier>
          - Arn
```

```
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApi9E278160
            - /
            - Ref: HelloWorldApiDeploymentStageprodunique-identifier
            - /GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETApiPermissionTestCdkHelloWorldStackHelloWorldApiunique-
identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApiunique-identifier
            - /test-invoke-stage/GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.Test.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETunique-identifier:
```

```
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS_PROXY
        Uri:
          Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - ":apigateway:"
              - Ref: AWS::Region
              - :lambda:path/2015-03-31/functions/
              - Fn::GetAtt:
                  - HelloWorldFunctionunique-identifier
                  - Arn
              - /invocations
      ResourceId:
        Ref: HelloWorldApihellounique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/Resource
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:unique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Outputs:
  HelloWorldApiEndpointunique-identifier:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: HelloWorldApiunique-identifier
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
```

```
          - Ref: HelloWorldApiDeploymentStageprodunique-identifier
          - /
Conditions:
  CDKMetadataAvailable:
    Fn::Or:
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - af-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ca-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-northwest-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-north-1
          - Fn::Equals:
```

```
              - Ref: AWS::Region
              - eu-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-2
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-3
          - Fn::Equals:
              - Ref: AWS::Region
              - il-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - sa-east-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-2
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-2
Parameters:
  BootstrapVersion:
    Type: AWS::SSM::Parameter::Value<String>
    Default: /cdk-bootstrap/hnb659fds/version
    Description: Version of the CDK Bootstrap resources in this environment,
 automatically retrieved from SSM Parameter Store. [cdk:skip]
Rules:
  CheckBootstrapVersion:
```

```
    Assertions:
      - Assert:
          Fn::Not:
            - Fn::Contains:
                - - "1"
                  - "2"
                  - "3"
                  - "4"
                  - "5"
                - Ref: BootstrapVersion
        AssertDescription: CDK bootstrap stack version 6 required. Please run 'cdk
 bootstrap' with a recent version of the CDK CLI.
```

L2 コンストラクトを使用して、リソースを設定するためのいくつかのプロパティを定義し、ヘルパーメソッドを使用してそれらを統合します。は、アプリケーションのプロビジョニングに必要な AWS CloudFormation リソースとプロパティの大部分 AWS CDK を設定します。

ステップ 5: アプリケーションをデプロイする

このステップでは、 `cdk deploy` コマンドを使用して AWS CDK CLIアプリケーションをデプロイします。 AWS CDK は AWS CloudFormation サービスと連携してリソースをプロビジョニングします。

> **Important**
>
> デプロイする前に、 AWS 環境の 1 回限りのブートストラップを実行する必要があります。手順については、「環境のブートストラップ」を参照してください。

プロジェクトのルートから、以下を実行します。プロンプトが表示されたら、変更を確認します。

```
$ cdk deploy

#  Synthesis time: 2.44s

...

Do you wish to deploy these changes (y/n)? y
```

デプロイが完了すると、 はエンドポイント URL を AWS CDK CLI出力します。この URL を次のステップにコピーします。以下に例を示します。

```
...
#  HelloWorldStack

#  Deployment time: 45.37s

Outputs:
HelloWorldStack.HelloWorldApiEndpointunique-identifier = https://<api-id>.execute-
api.<region>.amazonaws.com/prod/
Stack ARN:
arn:aws:cloudformation:region:account-id:stack/HelloWorldStack/unique-identifier
...
```

ステップ 6: アプリケーションを操作する

このステップでは、API エンドポイントへの GET リクエストを開始し、Lambda 関数のレスポンスを受け取ります。

前のステップのエンドポイント URL を見つけて、/helloパスを追加します。次に、ブラウザまたはコマンドプロンプトを使用して、エンドポイントに GET リクエストを送信します。以下に例を示します。

```
$ curl https://<api-id>.execute-api.<region>.amazonaws.com/prod/hello
{"message":"Hello World!"}%
```

これで、 を使用してアプリケーションの作成、デプロイ、操作が正常に完了しました AWS CDK。

ステップ 7: アプリケーションを削除する

このステップでは、 を使用して AWS CDK CLIからアプリケーションを削除します AWS クラウド。

アプリケーションを削除するには、 を実行しますcdk destroy。プロンプトが表示されたら、アプリケーションを削除するリクエストを確認します。

```
$ cdk destroy
Are you sure you want to delete: CdkHelloWorldStack (y/n)? y
CdkHelloWorldStack: destroying... [1/1]
...
 #  CdkHelloWorldStack: destroyed
```

トラブルシューティング

エラー: {「メッセージ」 : 「内部サーバーエラー」}%

デプロイされた Lambda 関数を呼び出すと、このエラーが表示されます。このエラーは、複数の理由で発生する可能性があります。

さらにトラブルシューティングを行うには

を使用して AWS CLI Lambda 関数を呼び出します。

1. スタックファイルを変更して、デプロイされた Lambda 関数名の出力値をキャプチャします。以下に例を示します。

```
...

class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    new CfnOutput(this, 'HelloWorldFunctionName', {
      value: helloWorldFunction.functionName,
      description: 'JavaScript Lambda function'
    });

    // Define the API Gateway resource
    // ...
```

2. アプリケーションをもう一度デプロイします。 AWS CDK CLI は、デプロイされた Lambda 関数名の値を出力します。

```
$ cdk deploy

#  Synthesis time: 0.29s
...
 #  CdkHelloWorldStack

#  Deployment time: 20.36s
```

```
Outputs:
...
CdkHelloWorldStack.HelloWorldFunctionName = CdkHelloWorldStack-
HelloWorldFunctionunique-identifier
...
```

3. を使用して AWS CLI で Lambda 関数を呼び出し AWS クラウド 、レスポンスをテキストファイルに出力します。

```
$ aws lambda invoke --function-name CdkHelloWorldStack-HelloWorldFunctionunique-
identifier output.txt
```

4. 結果output.txtを確認します。

考えられる原因: API Gateway リソースがスタックファイルに誤って定義されています。

output.txt が Lambda 関数のレスポンスに成功した場合、API Gateway REST API の定義方法に問題がある可能性があります。は、エンドポイントではなく Lambda を直接 AWS CLI 呼び出します。コードをチェックして、このチュートリアルと一致することを確認してください。次に、再度デプロイします。

考えられる原因: Lambda リソースがスタックファイルに誤って定義されています。

がエラーをoutput.txt返す場合、問題は Lambda 関数の定義方法にある可能性があります。コードをチェックして、このチュートリアルと一致することを確認してください。次に、再度デプロイします。

複数のスタックを持つアプリケーションを作成する

複数のスタック を含む AWS Cloud Development Kit (AWS CDK) アプリケーションを作成できます。 AWS CDK アプリケーションをデプロイすると、各スタックが独自の AWS CloudFormation テンプレートになります。cdk deploy コマンドを使用して、各スタックを AWS CDK CLI個別に合成してデプロイすることもできます。

このチュートリアルでは、以下について説明します。

- クラスを拡張して新しいプロパティまたは引数Stackを受け入れる方法。
- プロパティを使用して、スタックに含まれるリソースとその設定を決定する方法。
- このクラスから複数のスタックをインスタンス化する方法。

このトピックの例では、 `encryptBucket` (Python:) という名前のブールプロパティを使用しますencrypt_bucket。Amazon S3 バケットを暗号化するかどうかを示します。その場合、スタックは AWS Key Management Service () によって管理されるキーを使用して暗号化を有効にしますAWS KMS。アプリケーションは、このスタックの 2 つのインスタンスを作成します。1 つは暗号化あり、もう 1 つは暗号化なしです。

トピック

開始する前に

まず、Node.js と AWS CDK コマンドラインツールをまだインストールしていない場合は、インストールします。詳細については、「の開始方法 AWS CDK」を参照してください。

次に、コマンドラインで次のコマンドを入力して AWS CDK プロジェクトを作成します。

TypeScript

```
mkdir multistack
cd multistack
cdk init --language=typescript
```

JavaScript

```
mkdir multistack
cd multistack
cdk init --language=javascript
```

Python

```
mkdir multistack
cd multistack
```

```
cdk init --language=python
source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
pip install -r requirements.txt
```

Java

```
mkdir multistack
cd multistack
cdk init --language=java
```

結果の Maven プロジェクトを Java IDE にインポートできます。

C#

```
mkdir multistack
cd multistack
cdk init --language=csharp
```

Visual Studio `src/Pipeline.sln`で ファイルを開くことができます。

オプションのパラメータを追加する

`Stack` コンストラクターの `props`引数は、インターフェイス を満たします`StackProps`。この例では、Amazon S3 バケットを暗号化するかどうかをスタックに指示するために、追加の プロパティを受け入れてもらいます。プロパティを含むインターフェイスまたはクラスを作成する必要があります。これにより、コンパイラは プロパティにブール値があることを確認し、IDE でプロパティの自動補完を有効にします。

したがって、IDE または工ディタで指定されたソースファイルを開き、新しいインターフェイス、クラス、または引数を追加します。コードは変更後、次のようになります。追加した行は太字で表示されます。

TypeScript

ファイル: `lib/multistack-stack.ts`

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

interface MultiStackProps extends cdk.StackProps {
```

```
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultiStackProps) {
    super(scope, id, props);

    // The code that defines your stack goes here
  }
}
```

JavaScript

ファイル: lib/multistack-stack.js

JavaScript にはインターフェイス機能がないため、コードを追加する必要はありません。

```
const cdk = require('aws-cdk-stack');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your stack goes here
  }
}

module.exports = { MultistackStack }
```

Python

ファイル: multistack/multistack_stack.py

Python にはインターフェイス機能がないため、キーワード引数を追加して新しいプロパティを受け入れるようにスタックを拡張します。

```
import aws_cdk as cdk
from constructs import Construct

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
```

```
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines your stack goes here
```

Java

ファイル: src/main/java/com/myorg/MultistackStack.java

Java でprops タイプを拡張するには、実際に にアクセスするよりも複雑です。代わりに、スタックのコンストラクタを記述して、オプションのブールパラメータを受け入れます。props はオプションの引数であるため、スキップできる追加のコンストラクタを記述します。デフォルトではになりますfalse。

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;

import software.amazon.awscdk.services.s3.Bucket;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id, boolean
 encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    public MultistackStack(final Construct scope, final String id, final StackProps
 props,
            final boolean encryptBucket) {
        super(scope, id, props);

        // The code that defines your stack goes here
    }
}
```

C#

ファイル: src/Multistack/MultistackStack.cs

```
using Amazon.CDK;
using constructs;

namespace Multistack
{


    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }


    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, MultiStackProps props) :
 base(scope, id, props)
        {
            // The code that defines your stack goes here
        }
    }
}
```

新しいプロパティはオプションです。encryptBucket (Python: encrypt_bucket) が存在しない場合、その値は undefined、または同等の ローカル です。バケットはデフォルトで暗号化されません。

スタッククラスを定義する

次に、新しい プロパティを使用してスタッククラスを定義しましょう。コードは次のようになります。追加または変更する必要があるコードは太字で表示されます。

TypeScript

ファイル: lib/multistack-stack.ts

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from constructs;
```

```
import * as s3 from 'aws-cdk-lib/aws-s3';

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultistackProps) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if (props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}
```

JavaScript

ファイル: lib/multistack-stack.js

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if ( props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
```

```
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}

module.exports = { MultistackStack }
```

Python

ファイル: multistack/multistack_stack.py

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add a Boolean property "encryptBucket" to the stack constructor.
        # If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
        # Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if encrypt_bucket:
            s3.Bucket(self, "MyGroovyBucket",
                      encryption=s3.BucketEncryption.KMS_MANAGED,
                      removal_policy=cdk.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                     removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

ファイル: src/main/java/com/myorg/MultistackStack.java

```
package com.myorg;
```

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.RemovalPolicy;

import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id,
            boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    // main constructor
    public MultistackStack(final Construct scope, final String id,
            final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is
        // unencrypted. Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if (encryptBucket) {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .encryption(BucketEncryption.KMS_MANAGED)
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        } else {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        }
    }
}
```

C#

ファイル: src/Multistack/MultistackStack.cs

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;
```

```
namespace Multistack
{

    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }

    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, IMultiStackProps props =
 null) : base(scope, id, props)
        {
            // Add a Boolean property "EncryptBucket" to the stack constructor.
            // If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
            // Encrypted bucket uses KMS-managed keys (SSE-KMS).
            if (props?.EncryptBucket ?? false)
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    Encryption = BucketEncryption.KMS_MANAGED,
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
            else
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
        }
    }
}
```

2 つのスタックインスタンスを作成する

次に、2 つの個別のスタックをインスタンス化するためのコードを追加します。以前と同様に、太字で示されているコード行は追加する必要があるコードです。既存のMultistackStack定義を削除します。

TypeScript

ファイル: bin/multistack.ts

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MultistackStack } from '../lib/multistack-stack';

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

JavaScript

ファイル: bin/multistack.js

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { MultistackStack } = require('../lib/multistack-stack');

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});
```

```
app.synth();
```

Python

ファイル: ./app.py

```
#!/usr/bin/env python3

import aws_cdk as cdk

from multistack.multistack_stack import MultistackStack

app = cdk.App()
MultistackStack(app, "MyWestCdkStack",
                env=cdk.Environment(region="us-west-1"),
                encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
                env=cdk.Environment(region="us-east-1"),
                encrypt_bucket=True)

app.synth()
```

Java

ファイル: src/main/java/com/myorg/MultistackApp.java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MultistackApp {
    public static void main(final String argv[]) {
        App app = new App();

        new MultistackStack(app, "MyWestCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-west-1")
                        .build())
                .build(), false);
```

```
        new MultistackStack(app, "MyEastCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-east-1")
                        .build())
                .build(), true);

        app.synth();
    }
}
```

C#

ファイル: src/Multistack/Program.cs

```
using Amazon.CDK;

namespace Multistack
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-west-1" },
                EncryptBucket = false
            });

            new MultistackStack(app, "MyEastCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-east-1" },
                EncryptBucket = true
            });

            app.Synth();
        }
    }
}
```

このコードは、 `MultistackStack` クラスの新しい `encryptBucket` (Python: `encrypt_bucket`) プロパティを使用して、以下をインスタンス化します。

- `us-east-1` AWS リージョンに暗号化された Amazon S3 バケットを持つ 1 つのスタック。
- `us-west-1` AWS リージョンに暗号化されていない Amazon S3 バケットを持つ 1 つのスタック。

スタックの合成とデプロイ

これで、アプリケーションからスタックをデプロイできます。まず、 の スタックの AWS CloudFormation テンプレート`MyEastCdkStack`を合成します`us-east-1`。これは暗号化された S3 バケットを持つスタックです。

```
$ cdk synth MyEastCdkStack
```

このスタックを AWS アカウントにデプロイするには、次のいずれかのコマンドを発行します。最初のコマンドでは、デフォルトの AWS プロファイルを使用して認証情報を取得し、スタックをデプロイします。2 つ目は、指定したプロファイルを使用します。*`PROFILE_NAME`* の場合は、`us-east-1` AWS リージョンにデプロイするための適切な認証情報を含む AWS CLI プロファイルの名前を置き換えます。

```
cdk deploy MyEastCdkStack
```

```
cdk deploy MyEastCdkStack --profile=PROFILE_NAME
```

クリーンアップ

デプロイしたリソースの料金を回避するには、次のコマンドを使用してスタックを破棄します。

```
cdk destroy MyEastCdkStack
```

スタックのバケットに何かが保存されている場合、破棄オペレーションは失敗します。このトピックの手順のみに従った場合は、 があってはなりません。ただし、バケットに何かを入れた場合は、スタックを破棄する前にバケットの内容を削除する必要があります。(バケット自体は削除しないでください)。バケットの内容を削除するには AWS CLI 、 AWS Management Console または を使用します。

例

このトピックには、次の例が含まれています。

- サーバーレス Hello World アプリケーションを作成する Lambda、API Gateway、Amazon S3 を使用してサーバーレスアプリケーションを作成します。
- を使用した AWS Fargate サービスの作成 AWS CDK 上のイメージから Amazon ECS Fargate サービスを作成します DockerHub。

を使用した AWS Fargate サービスの作成 AWS CDK

この例では、Amazon ECR 上のイメージからインターネット向け Application Load Balancer が前面にある Amazon Elastic Container Service (Amazon ECS) クラスターで実行されている AWS Fargate サービスを作成する方法について説明します。

Amazon ECS は、クラスターで Docker コンテナを簡単に実行、停止、管理できる非常にスケーラブルで高速なコンテナ管理サービスです。Fargate 起動タイプを使用してサービスまたはタスクを起動することで、Amazon ECS によって管理されるサーバーレスインフラストラクチャでクラスターをホストできます。より詳細な制御のために、Amazon EC2 起動タイプを使用して管理する Amazon Elastic Compute Cloud (Amazon EC2) インスタンスのクラスターでタスクをホストできます。

このチュートリアルでは、Fargate 起動タイプを使用して一部のサービスを起動する方法を示します。を使用して Fargate サービス AWS Management Console を作成した場合は、そのタスクを実行するための多くのステップがあることがわかります。 AWS には、Fargate サービスの作成を順を追って説明するチュートリアルとドキュメントのトピックがいくつかあります。

- Docker コンテナをデプロイする方法 - AWS
- Amazon ECS でのセットアップ
- Fargate を使用した Amazon ECS の開始方法

この例では、同様の Fargate サービスを AWS CDK コードで作成します。

このチュートリアルで使用される Amazon ECS コンストラクトは、以下の利点を提供することで AWS のサービスを使用するのに役立ちます。

- ロードバランサーを自動的に設定します。

- ロードバランサーのセキュリティグループを自動的に開きます。これにより、セキュリティグループを明示的に作成しなくても、ロードバランサーはインスタンスと通信できます。
- ターゲットグループにアタッチされているサービスとロードバランサー間の依存関係を自動的に順序付けします。この場合、 は、インスタンスが作成される前にリスナーを作成する正しい順序 AWS CDK を適用します。
- グループを自動的にスケーリングするユーザーデータを自動的に設定します。これにより、クラスターを AMIs に関連付けるための正しい設定が作成されます。
- パラメータの組み合わせを早期に検証します。これにより、以前に AWS CloudFormation 問題が明らかになり、デプロイ時間を節約できます。例えば、タスクによっては、メモリ設定を誤って設定するのは簡単です。以前は、アプリケーションをデプロイするまでエラーは発生しません。ただし、 AWS CDK はアプリを合成するときに設定ミスを検出し、エラーを出力できるようになりました。
- Amazon ECR のイメージを使用する場合、Amazon Elastic Container Registry (Amazon ECR) のアクセス許可を自動的に追加します。
- 自動的にスケーリングします。 AWS CDK には、Amazon EC2 クラスターを使用するときにインスタンスを自動スケーリングするための メソッドが用意されています。これは、Fargate クラスターでインスタンスを使用する場合に自動的に発生します。

  さらに、 は、自動スケーリングがインスタンスを強制終了しようとしても、タスクが実行されているか、そのインスタンスでスケジュールされている場合に、インスタンスが削除され AWS CDK ないようにします。

  以前は、この機能を使用するには Lambda 関数を作成する必要がありました。
- アセットサポートを提供するため、1 つのステップでマシンから Amazon ECS にソースをデプロイできます。以前は、アプリケーションソースを使用するには、Amazon ECR へのアップロードや Docker イメージの作成など、いくつかの手動ステップを実行する必要がありました。

詳細については、「ECS」を参照してください。

> **Important**
>
> 使用するApplicationLoadBalancedFargateServiceコンストラクトには多数の AWS コンポーネントが含まれており、その中には、使用していない場合でも、 AWS アカウントにプロビジョニングされたままにしておくと煩雑なコストがかからないコンポーネントもあります。この例を完了したら、必ず (cdk destroy) をクリーンアップしてください。

ディレクトリの作成と の初期化 AWS CDK

まず、 AWS CDK コードを保持するディレクトリを作成し、次にそのディレクトリに AWS CDK アプリケーションを作成します。

TypeScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language typescript
```

JavaScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language javascript
```

Python

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language python
source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
pip install -r requirements.txt
```

Java

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language java
```

これで、Maven プロジェクトを IDE にインポートできます。

C#

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language csharp
```

これで Visual Studio `src/MyEcsConstruct.sln`で を開くことができます。

アプリケーションを実行し、空のスタックが作成されていることを確認します。

```
cdk synth
```

Fargate サービスを作成する

Amazon ECS でコンテナタスクを実行するには、次の 2 つの異なる方法があります。

- Amazon ECS がコンテナが実行されている物理マシンを管理するFargate起動タイプを使用します。
- 自動スケーリングの指定など、管理を行うEC2起動タイプを使用します。

この例では、インターネット向け Application Load Balancer をフロントとする ECS クラスターで実行されている Fargate サービスを作成します。

次の AWS コンストラクトライブラリモジュールのインポートを、指定されたファイルに追加します。

TypeScript

ファイル: lib/my_ecs_construct-stack.ts

```
import * as ec2 from "aws-cdk-lib/aws-ec2";
import * as ecs from "aws-cdk-lib/aws-ecs";
import * as ecs_patterns from "aws-cdk-lib/aws-ecs-patterns";
```

JavaScript

ファイル: lib/my_ecs_construct-stack.js

```
const ec2 = require("aws-cdk-lib/aws-ec2");
const ecs = require("aws-cdk-lib/aws-ecs");
const ecs_patterns = require("aws-cdk-lib/aws-ecs-patterns");
```

Python

ファイル: my_ecs_construct/my_ecs_construct_stack.py

```
from aws_cdk import (aws_ec2 as ec2, aws_ecs as ecs,
```

```
                    aws_ecs_patterns as ecs_patterns)
```

Java

ファイル: src/main/java/com/myorg/MyEcsConstructStack.java

```
import software.amazon.awscdk.services.ec2.*;
import software.amazon.awscdk.services.ecs.*;
import software.amazon.awscdk.services.ecs.patterns.*;
```

C#

ファイル: src/MyEcsConstruct/MyEcsConstructStack.cs

```
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;
```

コンストラクタの末尾にあるコメントを次のコードに置き換えます。

TypeScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

JavaScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

Python

```
        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is True
```

Java

```
        Vpc vpc = Vpc.Builder.create(this, "MyVpc")
                             .maxAzs(3)  // Default is all AZs in region
                             .build();
```

```
        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                              .vpc(vpc).build();

        // Create a load-balanced Fargate service and make it public
        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                      .cluster(cluster)           // Required
                      .cpu(512)                   // Default is 256
                       .desiredCount(6)            // Default is 1
                       .taskImageOptions(
                               ApplicationLoadBalancedTaskImageOptions.builder()
                                       .image(ContainerImage.fromRegistry("amazon/
amazon-ecs-sample"))
                                       .build())
                       .memoryLimitMiB(2048)       // Default is 512
                       .publicLoadBalancer(true)   // Default is true
                       .build();
```

C#

```
            var vpc = new Vpc(this, "MyVpc", new VpcProps
            {
                MaxAzs = 3 // Default is all AZs in region
            });

            var cluster = new Cluster(this, "MyCluster", new ClusterProps
            {
                Vpc = vpc
            });

            // Create a load-balanced Fargate service and make it public
            new ApplicationLoadBalancedFargateService(this, "MyFargateService",
                new ApplicationLoadBalancedFargateServiceProps
                {
                    Cluster = cluster,          // Required
                    DesiredCount = 6,           // Default is 1
                    TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
                    {
                        Image = ContainerImage.FromRegistry("amazon/amazon-ecs-
sample")
                    },
                    MemoryLimitMiB = 2048,      // Default is 256
                    PublicLoadBalancer = true   // Default is true
```

```
            }
        );
```

保存し、実行されスタックが作成されていることを確認します。

```
cdk synth
```

スタックは数百行なので、ここでは表示しません。スタックには、1 つのデフォルトインスタンス、3 つのアベイラビリティーゾーンのプライベートサブネットとパブリックサブネット、およびセキュリティグループが含まれている必要があります。

スタックをデプロイします。

```
cdk deploy
```

AWS CloudFormation は、アプリをデプロイする際に実行する数十のステップに関する情報を表示します。

これは、Fargate 搭載の Amazon ECS サービスを作成して Docker イメージを実行するのが非常に簡単です。

クリーンアップ

予期しない AWS 料金が発生しないようにするには、この演習を完了した後に AWS CDK スタックを破棄してください。

```
cdk destroy
```

AWS CDK 例

サポートされているプログラミング言語の AWS CDK スタックとアプリケーションのその他の例については、「」のAWS CDK 「サンプルリポジトリ」を参照してください GitHub。

AWS CDK ツール

このセクションでは、以下に示す AWS CDK ツールについて説明します。

トピック

- AWS CDK ツールキット (cdk コマンド)
- AWS Toolkit for Visual Studio Code
- AWS SAM 統合

AWS CDK ツールキット (**cdk** コマンド)

Toolkit AWS CDK 、CLI コマンド はcdk、 AWS CDK アプリケーションを操作するための主要なツールです。アプリケーションを実行し、定義したアプリケーションモデルを調べ、 によって生成された AWS CloudFormation テンプレートを生成してデプロイします AWS CDK。また、 AWS CDK プロジェクトの作成や操作に役立つ他の機能も提供します。このトピックでは、CDK Toolkit の一般的なユースケースについて説明します。

AWS CDK Toolkit は Node Package Manager と共にインストールされます。ほとんどの場合、グローバルにインストールすることをお勧めします。

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

> **Tip**
>
> の複数のバージョンを定期的に使用する場合は AWS CDK、個々の CDK プロジェクトに一致するバージョンの AWS CDK Toolkit をインストールすることを検討してください。これを行うには、 `npm install` コマンド`-g`から を省略します。次に`npx aws-cdk`、 を使用して呼び出します。これにより、ローカルバージョンが存在する場合は実行され、存在しない場合はグローバルバージョンにフォールバックします。

ツールキットのコマンド

すべての CDK Toolkit コマンドは で始まりcdk、その後にサブコマンド (list、synthesize、deployなど) が続きます。一部のサブコマンドにはls、同等の短いバージョン (synth、 など) があります。オプションと引数は、任意の順序でサブコマンドに従います。

すべてのサブコマンド、オプション、引数の説明については、「」を参照してくださいAWS CDK CLI コマンドリファレンス。

オプションとその値の指定

コマンドラインオプションは 2 つのハイフン () で始まります--。頻繁に使用されるオプションの中には、1 文字のシノニムが 1 つのハイフンで始まるものもあります (例えば、 --appにはシノニムがあります-a) 。 AWS CDK Toolkit コマンドのオプション順序は重要ではありません。

すべてのオプションは値を受け入れ、オプション名に従う必要があります。値は、名前から空白または等号 で区切ることができます=。次の 2 つのオプションは同等です。

```
--toolkit-stack-name MyBootstrapStack
--toolkit-stack-name=MyBootstrapStack
```

一部のオプションはフラグ (ブール値) です。値falseとして trueまたは を指定できます。値を指定しない場合、値は と見なされますtrue。を暗示no-するために、オプション名の前に を付けることもできますfalse。

```
# sets staging flag to true
--staging
--staging=true
--staging true

# sets staging flag to false
--no-staging
--staging=false
--staging false
```

複数の値を指定するには、--context、、--parameters、--plugin--tags、 などのいくつかのオプションを複数回指定--trustできます。これらは CDK Toolkit のヘルプで [array] タイプと表記されます。例:

```
cdk bootstrap --tags costCenter=0123 --tags responsibleParty=jdoe
```

組み込みヘルプ

AWS CDK ツールキットには統合されたヘルプがあります。ユーティリティに関する一般的なヘルプと、提供されたサブコマンドのリストは、次の発行で確認できます。

```
cdk --help
```

例えば など、特定のサブコマンドのヘルプを表示するにはdeploy、 --helpフラグの前に指定します。

```
cdk deploy --help
```

AWS CDK Toolkit のバージョンcdk versionを表示する問題。サポートをリクエストするときに、この情報を入力します。

バージョンレポート

AWS CDK の使用方法を把握するために、 AWS CDK アプリケーションで使用されるコンストラクトは、 として識別されるリソースを使用して収集および報告されますAWS::CDK::Metadata。このリソースは AWS CloudFormation テンプレートに追加され、簡単に確認できます。この情報は、既知のセキュリティまたは信頼性の問題があるコンストラクトを使用してスタックを識別 AWS するために でも使用できます。また、重要な情報をユーザーに連絡するためにも使用できます。

> **Note**
>
> バージョン 1.93.0 以前は、 は、スタックで使用されるコンストラクトではなく、合成中にロードされたモジュールの名前とバージョン AWS CDK を報告していました。

デフォルトでは、 はスタックで使用される次の NPM モジュールでの コンストラクトの使用を AWS CDK レポートします。

- AWS CDK コアモジュール
- AWS ライブラリモジュールの構築
- AWS ソリューション構築モジュール
- AWS Farm Deployment Kit モジュールをレンダリングする

AWS::CDK::Metadata リソースは次のようになります。

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Analytics:
 "v2:deflate64:H4sIAND9SGAAAzXKSw5AMBAA0L1b2PdzBYnEAdio3RglglY60zQi7u6TWL/
XKmNUlxeQSOKwaPTBqrNhwEWU3hGHiCzK0dWWfAxoL/Fd8mvk+QkS/0X6BdjnCdgmOOQKWz
+AqqLDt2Y3YMnLYWwAAAA="
```

Analytics プロパティは、スタック内のコンストラクトの、gzipped、base64 エンコード、プレフィックスエンコードされたリストです。

バージョンレポートをオプトアウトするには、次のいずれかの方法を使用します。

- 引--no-version-reporting数を指定して cdk コマンドを使用して、1 つのコマンドをオプトアウトします。

  ```
  cdk --no-version-reporting synth
  ```

  Toolkit AWS CDK はデプロイ前に新しいテンプレートを合成するため、 --no-version-reporting cdk deploy コマンドにも を追加する必要があります。
- ./cdk.json または で false versionReportingに設定します~/.cdk.json。これは、個々のコマンド--version-reportingで を指定してオプトインしない限りオプトアウトします。

  ```
  {
    "app": "...",
    "versionReporting": false
  }
  ```

による認証 AWS

AWS リソースへのプログラムによるアクセスを設定する方法は、環境と利用可能な AWS アクセスに応じて異なります。

認証方法を選択し、CDK Toolkit 用に設定するには、「 SDK とツールのリファレンスガイド」の「認証とアクセス」を参照してください。 AWS SDKs

雇用主から認証方法が与えられていないローカルで開発している新規ユーザーには、 を設定することをお勧めします AWS IAM Identity Center。この方法には、設定を容易に AWS CLI するために をインストールしたり、 AWS アクセスポータルに定期的にサインインしたりすることが含まれます。

この方法を選択した場合、AWS SDK とツールのリファレンスガイドの IAM Identity Center 認証の手順を完了したあと、環境には次の要素が含まれるはずです。

- アプリケーションを実行する前に AWS アクセスポータルセッションを開始 AWS CLIするために使用する 。
- から参照できる設定値のセットを持つ[default]プロファイルを持つ共有 AWSconfigファイル AWS CDK。このファイルの場所を確認するには、AWS SDK とツールのリファレンスガイドの「共有ファイルの場所」を参照してください。
- 共有 config ファイルは region 設定を設定します。これにより、 AWS CDK および CDK Toolkit AWS リージョン が AWS リクエストに使用するデフォルトの が設定されます。
- CDK Toolkit は、 にリクエストを送信する前に、プロファイルの SSO トークンプロバイダー設定を使用して認証情報を取得します AWS。IAM Identity Center アクセス許可セットに接続された IAM ロールである sso_role_name値では、アプリケーションで AWS のサービス 使用されている へのアクセスを許可する必要があります。

  次のサンプル config ファイルは、SSO トークンプロバイダー設定で設定されたデフォルトプロファイルを示しています。プロファイルの sso_session 設定は、指定された sso-session セクションを参照します。sso-session セクションには、 AWS アクセスポータルセッションを開始するための設定が含まれています。

```
[default]
sso_session = my-sso
sso_account_id = 111122223333
sso_role_name = SampleRole
region = us-east-1
output = json

[sso-session my-sso]
sso_region = us-east-1
sso_start_url = https://provided-domain.awsapps.com/start
sso_registration_scopes = sso:account:access
```

AWS アクセスポータルセッションを開始する

にアクセスする前に AWS のサービス、CDK Toolkit が IAM Identity Center 認証を使用して認証情報を解決するためのアクティブな AWS アクセスポータルセッションが必要です。設定したセッションの長さによっては、アクセスが最終的に期限切れになり、CDK Toolkit で認証エラーが発生します。で次のコマンドを実行して AWS CLI 、 AWS アクセスポータルにサインインします。

```
aws sso login
```

SSO トークンプロバイダーの設定で、デフォルトプロファイルではなく名前付きプロファイルを使用している場合、コマンドは ですaws sso login --profile *NAME*。--profile オプションまたはAWS_PROFILE環境変数を使用してcdkコマンドを発行する場合は、このプロファイルも指定します。

既にアクティブなセッションがあるかどうかをテストするには、次の AWS CLI コマンドを実行します。

```
aws sts get-caller-identity
```

このコマンドへの応答により、共有 config ファイルに設定されている IAM Identity Center アカウントとアクセス許可のセットが報告されます。

> **Note**
>
> 既にアクティブな AWS アクセスポータルセッションがあり、 を実行している場合はaws sso login、認証情報を入力する必要はありません。
> サインインプロセスにより、データ AWS CLI へのアクセスを許可するように求められる場合があります。 AWS CLI は SDK for Python 上に構築されているため、アクセス許可メッセージにはbotocore名前のバリエーションが含まれている可能性があります。

リージョンおよびその他の設定の指定

CDK Toolkit は、デプロイ先の AWS リージョンと、 で認証する方法を知る必要があります AWS。これは、デプロイオペレーションと合成中にコンテキスト値を取得するために必要です。アカウントとリージョンが一緒に環境 を構成します。

リージョンは、環境変数または設定ファイルで指定できます。これらは、 やさまざまな SDK などの他の AWS ツールで使用されるのと同じ変数 AWS CLI とファイルです。 AWS SDKs CDK Toolkit は、この情報を次の順序で検索します。

- AWS_DEFAULT_REGION 環境変数。
- 標準 AWS configファイルで定義され、 cdk コマンドの --profileオプションを使用して指定された名前付きプロファイル。
- 標準 AWS configファイルの [default]セクション。

[default] セクションで AWS 認証とリージョンを指定するだけでなく、1 つ以上の[profile *NAME*]セクションを追加することもできます。*NAME* はプロファイルの名前です。名前付きプロファイルの詳細については、 SDK およびツールリファレンスガイドの「共有設定ファイルと認証情報ファイル」を参照してください。 AWS SDKs

標準 AWS configファイルは、 ~/.aws/config (macOS /Linux) または %USERPROFILE%\.aws\config (Windows) にあります。詳細と代替の場所については、「 SDK とツールのリファレンスガイド」の「共有設定ファイルと認証情報ファイルの場所」を参照してください。 AWS SDKs

スタックの envプロパティを使用して AWS CDK アプリで指定した環境は、合成中に使用されます。これは環境固有の AWS CloudFormation テンプレートを生成するために使用され、デプロイ中に、前述の方法のいずれかで指定されたアカウントまたはリージョンを上書きします。詳細については、「the section called “環境”」を参照してください。

> **Note**
>
> AWS CDK は、 を含む他の AWS ツールや SDKsと同じソースファイルからの認証情報を使用しますAWS Command Line Interface。ただし、 AWS CDK の動作は、これらのツールとは多少異なる場合があります。内部 AWS SDK for JavaScript の を使用します。の認証情報の設定の詳細については AWS SDK for JavaScript、「認証情報の設定」を参照してください。

オプションで --role-arn (または -r) オプションを使用して、デプロイに使用する IAM ロールの ARN を指定できます。このロールは、使用する AWS アカウントによって引き受け可能である必要があります。

アプリケーションコマンドの指定

CDK Toolkit の多くの機能では、1 つ以上の AWS CloudFormation テンプレートを合成する必要があります。合成するには、アプリケーションの実行が必要です。は、さまざまな言語で記述されたプログラム AWS CDK をサポートしています。したがって、設定オプションを使用して、アプリケーションの実行に必要な正確なコマンドを指定します。このオプションは 2 つの方法で指定できます。

まず、最も一般的には、ファイル 内の appキーを使用して指定できますcdk.json。これは AWS CDK プロジェクトのメインディレクトリにあります。CDK Toolkit は、 で新しいプロジェクトを作成するときに適切なコマンドを提供しますcdk init。例えば、新しい TypeScript プロジェクトの cdk.jsonを次に示します。

```
{
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

CDK Toolkit は、アプリケーションの実行を試みるときに、現在の作業ディレクトリcdk.jsonで を検索します。このため、CDK Toolkit コマンドを発行するために、プロジェクトのメインディレクトリでシェルを開いたままにしておくことができます。

CDK Toolkit は、 でアプリキーが見つからない場合、 で ~/.cdk.json (つまり、ホームディレクトリで) アプリキーも検索します./cdk.json。アプリコマンドをここに追加すると、通常、同じ言語で CDK コードを操作する場合に便利です。

他のディレクトリにいる場合、または のコマンド以外のコマンドを使用してアプリケーションを実行する場合はcdk.json、 --app (または -a) オプションを使用して指定します。

```
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

デプロイ時に、 などの合成されたクラウドアセンブリを含むディレクトリを cdk.outの値として指定することもできます--app。指定されたスタックはこのディレクトリからデプロイされます。アプリは合成されません。

スタックの指定

多くの CDK Toolkit コマンド (などcdk deploy) は、アプリケーションで定義されたスタックで動作します。アプリケーションにスタックが 1 つだけ含まれている場合、スタックを明示的に指定しない場合、CDK Toolkit はスタックを前提としています。

それ以外の場合は、使用するスタックを指定する必要があります。これを行うには、コマンドラインで ID で目的のスタックを個別に指定します。ID は、スタックをインスタンス化するときに 2 番目の引数で指定された値であることを思い出してください。

```
cdk synth PipelineStack LambdaStack
```

ワイルドカードを使用して、パターンに一致する IDsを指定することもできます。

- ? は任意の 1 文字に一致します
- * は任意の数の文字に一致します (*単独ではすべてのスタックに一致します)
- ** は階層内のすべてのものと一致します。

--all オプションを使用して、すべてのスタックを指定することもできます。

アプリケーションが CDK Pipelines を使用している場合、CDK Toolkit はスタックとステージを階層として理解します。また、 --allオプションと*ワイルドカードは最上位スタックのみに一致します。すべてのスタックを照合するには、 を使用します**。また**、 を使用して、特定の階層のすべてのスタックを示します。

ワイルドカードを使用する場合は、パターンを引用符で囲むか、ワイルドカードを でエスケープします\。そうしないと、シェルがパターンを現在のディレクトリ内のファイルの名前に拡張しようとする可能性があります。最大で、これは期待どおりには動作しません。最悪の場合は、意図していなかったスタックをデプロイできます。はワイルドカードを拡張cmd.exeしないため、Windows では必須ではありませんが、それでも良い方法です。

```
cdk synth "*Stack"    # PipelineStack, LambdaStack, etc.
cdk synth 'Stack?'    # StackA, StackB, Stack1, etc.
cdk synth \*          # All stacks in the app, or all top-level stacks in a CDK
 Pipelines app
cdk synth '**'        # All stacks in a CDK Pipelines app
cdk synth 'PipelineStack/Prod/**'   # All stacks in Prod stage in a CDK Pipelines app
```

> **Note**
>
> スタックを指定する順序は、必ずしもスタックが処理される順序ではありません。Toolkit AWS CDK は、スタックの処理順序を決定する際に、スタック間の依存関係を考慮します。例えば、あるスタックが別のスタックによって生成された値 (2 番目のスタックで定義されたリソースの ARN など) を使用しているとします。この場合、この依存関係のため、2 番目のスタックは最初のスタックの前に合成されます。スタックの addDependency()メソッドを使用して、スタック間に依存関係を手動で追加できます。

AWS 環境のブートストラップ

CDK を使用してスタックをデプロイするには、特別な専用 AWS CDK リソースをプロビジョニングする必要があります。cdk bootstrap コマンドは、必要なリソースを作成します。これらの専用リソースを必要とするスタックをデプロイする場合にのみ、ブートストラップする必要があります。詳細については、「the section called “ブートストラッピング”」を参照してください。

```
cdk bootstrap
```

ここに示すように、引数なしで発行された場合、 `cdk bootstrap` コマンドは現在のアプリケーションを合成し、スタックがデプロイされる環境をブートストラップします。アプリケーションに環境に依存しないスタックが含まれており、環境を明示的に指定していない場合、デフォルトのアカウントとリージョンはブートストラップされるか、 を使用して指定された環境がブートストラップされます`--profile`。

アプリの外部では、ブートストラップする環境を明示的に指定する必要があります。また、アプリケーションまたはローカル AWS プロファイルで指定されていない環境をブートストラップすることもできます。認証情報は、指定されたアカウントとリージョンに対して設定する必要があります (例: で`~/.aws/credentials`)。必要な認証情報を含むプロファイルを指定できます。

```
cdk bootstrap ACCOUNT-NUMBER/REGION # e.g.
cdk bootstrap 1111111111/us-east-1
cdk bootstrap --profile test 1111111111/us-east-1
```

Important

このようなスタックをデプロイする各環境 (アカウント/リージョンの組み合わせ) は、個別にブートストラップする必要があります。

ブートストラップされたリソース内の AWS CDK ストアに対して AWS 料金が発生する場合があります。さらに、 を使用する場合`-bootstrap-customer-key`、AWS KMS キーが作成され、環境ごとに料金も発生します。

Note

以前のバージョンのブートストラップテンプレートでは、デフォルトで KMS キーが作成されていました。料金が発生しないようにするには、 を使用してブートストラップし直します`--no-bootstrap-customer-key`。

Note

CDK Toolkit v2 は、CDK v1 でデフォルトで使用されているレガシーテンプレートと呼ばれる元のブートストラップテンプレートをサポートしていません。

> **Important**
>
> 最新のブートストラップテンプレートは、 が暗示するアクセス許可`--cloudformation-execution-policies`を`--trust`リスト内の任意の AWS アカウントに効果的に付与します。デフォルトでは、これにより、ブートストラップされたアカウントの任意のリソースに対する読み取りと書き込みのアクセス許可が拡張されます。ブートストラップスタックには、使い慣れたポリシーと信頼できるアカウントを設定してください。

新しいアプリケーションの作成

新しいアプリケーションを作成するには、そのアプリケーション用のディレクトリを作成し、ディレクトリ内で を発行しますcdk init。

```
mkdir my-cdk-app
cd my-cdk-app
cdk init TEMPLATE --language LANGUAGE
```

サポートされている言語 (*##)* は次のとおりです。

Code	[言語]
typescript	TypeScript
javascript	JavaScript
python	Python
java	Java
csharp	C#

TEMPLATE はオプションのテンプレートです。目的のテンプレートがアプリケーション の場合、デフォルトは省略できます。使用可能なテンプレートは次のとおりです。

テンプレート	説明
app (デフォルト)	空の AWS CDK アプリケーションを作成します。
`sample-app`	Amazon SQS キューと Amazon SNS トピックを含むスタックを持つ AWS CDK アプリケーションを作成します。

テンプレートは、プロジェクトフォルダの名前を使用して、新しいアプリケーション内のファイルとクラスの名前を生成します。

スタックの一覧表示

AWS CDK アプリケーション内のスタックの IDs のリストを表示するには、次のいずれかの同等のコマンドを入力します。

```
cdk list
cdk ls
```

アプリケーションに CDK Pipelines スタックが含まれている場合、CDK Toolkit はパイプライン階層内の場所に応じてスタック名をパスとして表示します。(例えば、`PipelineStack`、`PipelineStack/Prod`、および など) `PipelineStack/Prod/MyService`。

アプリケーションに多数のスタックが含まれている場合は、一覧表示するスタックのフルまたは一部のスタック IDs を指定できます。詳細については、「the section called “スタックの指定”」を参照してください。

`--long` フラグを追加して、スタック名とその環境 (AWS アカウントとリージョン) など、スタックに関する詳細情報を表示します。

スタックの合成

`cdk synthesize` コマンド (通常は略称 `synth`) は、アプリケーションで定義されたスタックを CloudFormation テンプレートに合成します。

```
cdk synth       # if app contains only one stack
```

```
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*"     # all stacks in app
```

> **Note**
>
> CDK Toolkit は実際にアプリケーションを実行し、ほとんどのオペレーション (スタックのデプロイや比較時など) の前に新しいテンプレートを合成します。これらのテンプレートは、デフォルトで cdk.out ディレクトリに保存されます。cdk synth コマンドは、1 つ以上の指定されたスタックに対して生成されたテンプレートを単に出力します。

使用可能なすべてのオプションcdk synth --helpについては、「」を参照してください。最も頻繁に使用されるオプションのいくつかを次のセクションで説明します。

コンテキスト値の指定

--context または -cオプションを使用して、ランタイムコンテキスト値を CDK アプリケーションに渡します。

```
# specify a single context value
cdk synth --context key=value MyStack

# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

複数のスタックをデプロイする場合、指定されたコンテキスト値は通常、すべてのスタックに渡されます。必要に応じて、スタック名の前にコンテキスト値を付けることで、スタックごとに異なる値を指定できます。

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

表示形式の指定

デフォルトでは、合成されたテンプレートは YAML 形式で表示されます。--json フラグを追加して、代わりに JSON 形式で表示します。

```
cdk synth --json MyStack
```

出力ディレクトリの指定

--output (-o) オプションを追加して、合成されたテンプレートを 以外のディレクトリに書き込みますcdk.out。

```
cdk synth --output=~/templates
```

スタックのデプロイ

cdk deploy サブコマンドは、指定した 1 つ以上のスタックを AWS アカウントにデプロイします。

```
cdk deploy        # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"    # all stacks in app
```

> **Note**
>
> CDK Toolkit はアプリを実行し、新しい AWS CloudFormation テンプレートを合成してからデプロイします。したがって、 で使用できるほとんどのコマンドラインオプション cdk synth (など--context) は、 でも使用できますcdk deploy。

使用可能なすべてのオプションcdk deploy --helpについては、「」を参照してください。次のセクションでは、最も便利なオプションをいくつか紹介します。

合成のスキップ

cdk deploy コマンドは通常、デプロイ前にアプリケーションのスタックを合成して、デプロイがアプリケーションの最新バージョンを反映していることを確認します。前回の 以降にコードを変更していないことがわかっている場合はcdk synth、デプロイ時に冗長合成ステップを抑制できます。そのためには、 --appオプションでプロジェクトのcdk.outディレクトリを指定します。

```
cdk deploy --app cdk.out StackOne StackTwo
```

ロールバックの無効化

AWS CloudFormation には、デプロイがアトミックになるように変更をロールバックする機能があります。つまり、全体として成功または失敗します。は AWS CloudFormation テンプレートを合成してデプロイするため、この機能を AWS CDK 継承します。

ロールバックにより、リソースは常に一貫した状態になります。これは、本番稼働用スタックにとって不可欠です。ただし、まだインフラストラクチャを開発している間、一部の障害は避けられず、失敗したデプロイをロールバックすると速度が低下する可能性があります。

このため、CDK Toolkit では、 `cdk deploy` コマンド`--no-rollback`に を追加してロールバックを無効にできます。このフラグでは、失敗したデプロイはロールバックされません。代わりに、失敗したリソースの前にデプロイされたリソースはそのまま残り、次のデプロイは失敗したリソースから始まります。デプロイを待つ時間が大幅に短縮され、インフラストラクチャの開発に費やす時間が長くなります。

ホットスワップ

で `--hotswap`フラグ`cdk deploy`を使用して、 AWS CloudFormation 変更セットを生成してデプロイするのではなく、 AWS リソースを直接更新しようとします。ホットスワップが不可能な場合、デプロイは AWS CloudFormation デプロイにフォールバックします。

現在、ホットスワップは Lambda 関数、Step Functions ステートマシン、および Amazon ECS コンテナイメージをサポートしています。この`--hotswap`フラグはロールバックも無効にします (つまり、 を意味します`--no-rollback`) 。

> **Important**
>
> ホットスワップは、本番デプロイではお勧めしません。

監視モード

CDK Toolkit のウォッチモード (または略`cdk watch`して) は`cdk deploy --watch`、CDK アプリケーションのソースファイルとアセットの変更を継続的にモニタリングします。変更が検出されると、指定されたスタックのデプロイがすぐに実行されます。

デフォルトでは、これらのデプロイは `--hotswap`フラグを使用します。これにより、Lambda 関数への変更のデプロイが高速に追跡されます。また、インフラストラクチャ設定を変更

AWS CloudFormation した場合は、 を通じてデプロイされます。がcdk watch常にフル AWS CloudFormation デプロイを実行するには、 --no-hotswapフラグを に追加しますcdk watch。

cdk watch が既にデプロイを実行している間に加えられた変更は 1 つのデプロイに統合され、進行中のデプロイが完了するとすぐに開始されます。

監視モードでは、プロジェクトの の"watch"キーを使用してcdk.json、監視するファイルを決定します。デフォルトでは、これらのファイルはアプリケーションファイルとアセットですが、"watch"キーの "include"および "exclude"エントリを変更することで変更できます。次のcdk.jsonファイルは、これらのエントリの例を示しています。

```
{
  "app": "mvn -e -q compile exec:java",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
}
```

cdk watch は から "build" コマンドを実行してcdk.json、合成前にアプリケーションを構築します。デプロイで Lambda コードを構築またはパッケージ化するためのコマンド (または CDK アプリにないその他のコマンド) が必要な場合は、ここに追加します。

Git スタイルのワイルドカードは、 *と の両方で**、 "watch" および "build"キーで使用できます。各パスは、 の親ディレクトリに対して解釈されますcdk.json。のデフォルト値は includeです。つまり**/*、プロジェクトのルートディレクトリ内のすべてのファイルとディレクトリです。excludeはオプションです。

> **Important**
>
> 本番稼働用デプロイでは、監視モードはお勧めしません。

AWS CloudFormation パラメータの指定

Toolkit は、デプロイ時の AWS CloudFormation パラメータの指定 AWS CDK をサポートしています。これらは、 --parametersフラグの後のコマンドラインで指定できます。

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

複数のパラメータを定義するには、複数の--parameterフラグを使用します。

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

複数のスタックをデプロイする場合は、スタックごとに各パラメータの異なる値を指定できます。そのためには、 パラメータの名前の前にスタック名とコロンを付けます。それ以外の場合、同じ値がすべてのスタックに渡されます。

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

デフォルトでは、 は以前のデプロイのパラメータの値 AWS CDK を保持し、明示的に指定されていない場合は後のデプロイで使用します。--no-previous-parameters フラグを使用して、すべてのパラメータを指定する必要があります。

出力ファイルの指定

スタックが AWS CloudFormation 出力を宣言した場合、これらは通常、デプロイの終了時に画面に表示されます。JSON 形式でファイルに書き込むには、 --outputs-fileフラグを使用します。

```
cdk deploy --outputs-file outputs.json MyStack
```

セキュリティ関連の変更

セキュリティ体制に影響する意図しない変更から保護するために、 AWS CDK ツールキットは、セキュリティ関連の変更をデプロイする前に承認するように促します。承認が必要な変更のレベルを指定できます。

```
cdk deploy --require-approval LEVEL
```

LEVEL は次のいずれかになります。

言葉	意味
never	承認は不要
any-change	IAM または security-group-related 変更の承認が必要

言葉	意味
broadening （デフォルト）	IAM ステートメントまたはトラフィックルールが追加されると承認が必要。削除には承認が必要ではない

設定は cdk.json ファイルで設定することもできます。

```
{
  "app": "...",
  "requireApproval": "never"
}
```

スタックの比較

cdk diff コマンドは、アプリケーションで定義されているスタックの最新バージョン (およびその依存関係) を、既にデプロイされているバージョン、または保存された AWS CloudFormation テンプレートと比較し、変更のリストを表示します。

```
Stack HelloCdkStack
IAM Statement Changes
###############################################################################################
#   # Resource                     # Effect # Action                       # Principal
                  # Condition #
###############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole               #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                             #
                  #           #
#   # .Arn}                        #        #                             #
                  #           #
###############################################################################################
# + # ${MyFirstBucket.Arn}         # Allow  # s3:DeleteObject*             # AWS:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*       #        # s3:GetBucket*                #
 jectsCustomResourceProvider/ #           #
#   #                              #        # s3:GetObject*                # Role.Arn}
                  #           #
#   #                              #        # s3:List*                     #
                  #           #
```

```
###############################################################################################
IAM Policy Changes
###############################################################################################
#   # Resource                                                  # Managed Policy ARN
                                  #
###############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${AWS::Partition}:iam::aws:policy/serv #
#   # le}                                                     # ice-role/
AWSLambdaBasicExecutionRole"}                  #
###############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] AWS::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] AWS::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] AWS::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
```

```
#   ## [-] Retain
#   ## [+] Delete
## [~] UpdateReplacePolicy
    ## [-] Retain
    ## [+] Delete
```

アプリケーションのスタックを既存のデプロイと比較するには：

```
cdk diff MyStack
```

アプリケーションのスタックを保存済み CloudFormation テンプレートと比較するには：

```
cdk diff --template ~/stacks/MyStack.old MyStack
```

スタックへの既存リソースのインポート

`cdk import` コマンドを使用して、リソースを特定の AWS CDK スタック CloudFormation の の管理下に置くことができます。これは、 に移行する場合 AWS CDK、またはスタック間でリソースを移動する場合、または論理 ID を変更する場合に便利です。 は CloudFormation リソースのインポート `cdk import`を使用します。ここにインポートできるリソースのリストを参照してください。

既存のリソースを AWS CDK スタックにインポートするには、次のステップに従います。

- リソースが現在他の CloudFormation スタックによって管理されていないことを確認します。その場合は、まずリソースが現在存在するスタック`RemovalPolicy.RETAIN`の削除ポリシーを に設定し、デプロイを実行します。次に、スタックからリソースを削除し、別のデプロイを実行します。このプロセスにより、リソースは によって管理されなくなり CloudFormation 、削除されなくなります。
- を実行して`cdk diff`、リソースをインポートする AWS CDK スタックに保留中の変更がないことを確認します。「インポート」オペレーションで許可される変更は、インポートする新しいリソースの追加のみです。
- スタックにインポートするリソースのコンストラクトを追加します。例えば、Amazon S3 バケットをインポートする場合は、 のようなものを追加します`new s3.Bucket(this, 'ImportedS3Bucket', {});`。他のリソースに変更を加えないでください。

  また、リソースが現在定義に持っている状態を正確にモデル化する必要があります。バケットの例については、 AWS KMS キー、ライフサイクルポリシー、およびバケットに関連するその他のものを必ず含めてください。そうしないと、後続の更新オペレーションが期待どおりに動作しない可能性があります。

物理バケット名を含めるかどうかを選択できます。通常、リソースを複数回デプロイしやすくするために、 AWS CDK リソース定義にリソース名を含めないことをお勧めします。

- `cdk import` *`STACKNAME`* を実行します。
- リソース名がモデルにない場合、インポートするリソースの実際の名前を渡すように CLI から求められます。その後、インポートが開始されます。
- が成功`cdk import`を報告すると、リソースは AWS CDK と によって管理されるようになりました CloudFormation。 AWS CDK アプリケーション内のリソースプロパティに後続の変更を加えると、コンストラクト設定は次のデプロイに適用されます。
- AWS CDK アプリのリソース定義がリソースの現在の状態と一致することを確認するには、CloudFormation ドリフト検出オペレーション を開始できます。

この機能は現在、ネストされたスタックへのリソースのインポートをサポートしていません。

設定 (`cdk.json`)

多くの CDK Toolkit コマンドラインフラグのデフォルト値は、プロジェクトの `cdk.json` ファイルまたはユーザーディレクトリの `.cdk.json` ファイルに保存できます。以下は、サポートされている構成設定のアルファベット順のリファレンスです。

キー	メモ	CDK Toolkit オプション
`app`	CDK アプリケーションを実行するコマンド。	--app
`assetMetadata`	の場合`false`、CDK はアセットを使用するリソースにメタデータを追加しません。	--no-asset-metadata
`bootstrapKmsKeyId`	Amazon S3 デプロイバケットの暗号化に使用される AWS KMS キーの ID を上書きします。 Amazon S3	--bootstrap-kms-key-id
`build`	合成前に CDK アプリケーションをコンパイルまたは構築す	--build

キー	メモ	CDK Toolkit オプション
	るコマンド。では許可されません~/.cdk.json 。	
browser	cdk docs サブコマンドのウェブブラウザを起動するためのコマンド。	--browser
context	the section called “Context” を参照してください。設定ファイルのコンテキスト値は、 によって消去されませんcdk context --clear。(CDK Toolkit はキャッシュされたコンテキスト値を に配置します) cdk.context.json 。	--context
debug	の場合true、CDK Toolkit はデバッグに役立つより詳細な情報を出力します。	--debug
language	新しいプロジェクトの初期化に使用される言語。	--language
lookups	の場合false、コンテキスト検索は許可されません。コンテキスト検索を実行する必要がある場合、合成は失敗します。	--no-lookups
notices	の場合false、セキュリティの脆弱性、リグレッション、サポートされていないバージョンに関するメッセージの表示を抑制します。	--no-notices

キー	メモ	CDK Toolkit オプション
`output`	合成されたクラウドアセンブリが出力されるディレクトリの名前 (デフォルトは `"cdk.out"`)。	--output
`outputsFile`	デプロイされたスタックからの AWS CloudFormation 出力が書き込まれるファイル (JSON 形式)。	--outputs-file
`pathMetadata`	の場合`false`、CDK パスメタデータは合成されたテンプレートに追加されません。	--no-path-metadata
`plugin`	CDK を拡張するパッケージ名またはパッケージのローカルパスを指定する JSON 配列	--plugin
`profile`	リージョンとアカウントの認証情報の指定に使用されるデフォルト AWS プロファイルの名前。	--profile
`progress`	に設定すると`"events"`、CDK Toolkit はデプロイ中のすべての AWS CloudFormation イベントを、進行状況バーではなく表示します。	--progress
`requireApproval`	セキュリティ変更のデフォルトの承認レベル。「the section called “セキュリティ関連の変更”」を参照	--require-approval

キー	メモ	CDK Toolkit オプション
`rollback`	の場合`false`、失敗したデプロイはロールバックされません。	--no-rollback
`staging`	の場合`false`、アセットは出力ディレクトリにコピーされません (を使用したソースファイルのローカルデバッグに使用します AWS SAM) 。	--no-staging
`tags`	スタックのタグ (キーと値のペア) を含む JSON オブジェクト。	--tags
`toolkitBucketName`	Lambda 関数やコンテナイメージなどのアセットのデプロイに使用される Amazon S3 バケットの名前 (「」を参照してくださいthe section called “AWS 環境のブートストラップ”。	--toolkit-bucket-name
`toolkitStackName`	ブートストラップスタックの名前 (「」を参照してくださいthe section called “AWS 環境のブートストラップ”。	--toolkit-stack-name
`versionReporting`	の場合`false`、 はバージョンレポートをオプトアウトします。	--no-version-reporting

キー	メモ	CDK Toolkit オプション
`watch`	変更時にプロジェクトの再構築をトリガーする (またはトリガーしない) ファイルを示す `"include"` および `"exclude"` キーを含む JSON オブジェクト。「the section called “監視モード”」を参照してください。	--watch

AWS Toolkit for Visual Studio Code

AWS Toolkit for Visual Studio Code は、 でのアプリケーションの作成、デバッグ、デプロイを容易にする Visual Studio Code 用のオープンソースプラグインです AWS。このツールキットは、 AWS CDK アプリケーションを開発するための統合エクスペリエンスを提供します。これには、プロジェクトを AWS CDK 一覧表示し、CDK アプリケーションのさまざまなコンポーネントを参照する AWS CDK Explorer 機能が含まれています。Toolkit をインストール AWSし、 AWS CDK Explorer の使用に関する詳細を確認します。

AWS SAM 統合

AWS CDK と AWS Serverless Application Model （ AWS SAM) は連携して、CDK で定義されたサーバーレスアプリケーションをローカルで構築およびテストできます。詳細については、「 AWS SAM デベロッパーガイドAWS Cloud Development Kit (AWS CDK)」の「」を参照してください。SAM CLI をインストールするには、「 AWS SAM CLI のインストール」を参照してください。

コンストラクトのテスト

を使用すると AWS CDK、インフラストラクチャは、記述する他のコードと同じくらいテスト可能になります。 AWS CDK アプリケーションをテストするための標準的なアプローチでは、 AWS CDK のアサーションモジュールと、Jest for TypeScript や JavaScript Pytest for Python などの一般的なテストフレームワークを使用します。

AWS CDK アプリケーション用に記述できるテストには 2 つのカテゴリがあります。

- きめ細かなアサーションは、「このリソースにはこの値を持つこのプロパティがあります」など、生成された AWS CloudFormation テンプレートの特定の側面をテストします。これらのテストでは、回帰を検出できます。また、テスト駆動型開発を使用して新機能を開発する場合にも役立ちます。（最初にテストを記述し、正しい実装を記述することで合格させることができます）。きめ細かなアサーションは、最も頻繁に使用されるテストです。
- スナップショットテストでは、以前に保存したベースライン AWS CloudFormation テンプレートに対して合成されたテンプレートをテストします。スナップショットテストでは、リファクタリングされたコードが元のコードとまったく同じように動作することを確認できます。変更が意図的なものであった場合は、将来のテストのために新しいベースラインを受け入れることができます。ただし、CDK のアップグレードにより合成されたテンプレートが変更される可能性があるため、スナップショットだけに依存して実装が正しいことを確認することはできません。

> **Note**
>
> このトピックで例として使用される TypeScript、Python、Java アプリケーションの完全なバージョンは、 で入手できます GitHub。

開始

これらのテストの記述方法を説明するために、 AWS Step Functions ステートマシンと AWS Lambda 関数を含むスタックを作成します。Lambda 関数は Amazon SNS トピックにサブスクライブされ、メッセージをステートマシンに転送します。

まず、CDK Toolkit を使用して空の CDK アプリケーションプロジェクトを作成し、必要なライブラリをインストールします。使用するコンストラクトはすべてメイン CDK パッケージにあります。これは CDK Toolkit で作成されたプロジェクトのデフォルトの依存関係です。ただし、テストフレームワークをインストールする必要があります。

TypeScript

```
mkdir state-machine && cd state-machine
cdk init --language=typescript
npm install --save-dev jest @types/jest
```

テスト用のディレクトリを作成します。

```
mkdir test
```

プロジェクトの を編集package.jsonして、NPM に Jest の実行方法を伝え、収集するファイルの種類を Jest に伝えます。必要な変更は次のとおりです。

- scripts セクションに新しいtestキーを追加する
- Jest とそのタイプを devDependenciesセクションに追加する
- moduleFileExtensions 宣言を含む新しいjest最上位キーを追加する

これらの変更を次の概要に示します。「」で示されている場所に新しいテキストを配置しますpackage.json。「...」プレースホルダーは、変更すべきでないファイルの既存の部分を示します。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "@types/jest": "^24.0.18",
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

JavaScript

```
mkdir state-machine && cd state-machine
```

```
cdk init --language=javascript
npm install --save-dev jest
```

テスト用のディレクトリを作成します。

```
mkdir test
```

プロジェクトの を編集package.jsonして、NPM に Jest の実行方法を伝え、収集するファイルの種類を Jest に伝えます。必要な変更は次のとおりです。

- scripts セクションに新しいtestキーを追加する
- devDependencies セクションに Jest を追加する
- moduleFileExtensions 宣言を含む新しいjest最上位キーを追加する

これらの変更を次の概要に示します。「」で示されている場所に新しいテキストを配置しますpackage.json。「...」プレースホルダーは、変更すべきでないファイルの既存の部分を示します。

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

Python

```
mkdir state-machine && cd state-machine
cdk init --language=python
source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
python -m pip install -r requirements.txt
```

```
python -m pip install -r requirements-dev.txt
```

Java

```
mkdir state-machine && cd-state-machine
cdk init --language=java
```

任意の Java IDE でプロジェクトを開きます。(Eclipse では、ファイル > インポート > 既存の Maven プロジェクトを使用します。)

C#

```
mkdir state-machine && cd-state-machine
cdk init --language=csharp
```

Visual Studio で `src\StateMachine.sln` を開きます。

Solution Explorer でソリューションを右クリックし、追加 > 新しいプロジェクト を選択します。MSTest C# を検索し、C# の MSTest テストプロジェクトを追加します。(デフォルト名`TestProject1`は問題ありません)。

右クリック`TestProject1`して追加 > プロジェクトリファレンス を選択し、`StateMachine`プロジェクトをリファレンスとして追加します。

サンプルスタック

このトピックでテストするスタックは次のとおりです。前述のように、Lambda 関数と Step Functions ステートマシンが含まれており、1 つ以上の Amazon SNS トピックを受け入れます。Lambda 関数は Amazon SNS トピックにサブスクライブされ、ステートマシンに転送されます。

アプリケーションをテスト可能にするために特別な操作を行う必要はありません。実際、この CDK スタックは、このガイドの他のサンプルスタックと重要な点で変わりません。

TypeScript

次のコードを に配置します`lib/state-machine-stack.ts`。

```
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
```

```
import * as sns_subscriptions from "aws-cdk-lib/aws-sns-subscriptions";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as sfn from "aws-cdk-lib/aws-stepfunctions";
import { Construct } from "constructs";

export interface StateMachineStackProps extends cdk.StackProps {
  readonly topics: sns.Topic[];
}

export class StateMachineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: StateMachineStackProps) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}
```

JavaScript

次のコードを に配置しますlib/state-machine-stack.js。

```
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const sns_subscriptions = require("aws-cdk-lib/aws-sns-subscriptions");
const lambda = require("aws-cdk-lib/aws-lambda");
```

```
const sfn = require("aws-cdk-lib/aws-stepfunctions");

class StateMachineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}

module.exports = { StateMachineStack }
```

Python

次のコードを に配置しますstate_machine/state_machine_stack.py。

```
from typing import List

import aws_cdk.aws_lambda as lambda_
import aws_cdk.aws_sns as sns
import aws_cdk.aws_sns_subscriptions as sns_subscriptions
import aws_cdk.aws_stepfunctions as sfn
import aws_cdk as cdk

class StateMachineStack(cdk.Stack):
```

```
    def __init__(
        self,
        scope: cdk.Construct,
        construct_id: str,
        *,
        topics: List[sns.Topic],
        **kwargs
    ) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # In the future this state machine will do some work...
        state_machine = sfn.StateMachine(
            self, "StateMachine", definition=sfn.Pass(self, "StartState")
        )

        # This Lambda function starts the state machine.
        func = lambda_.Function(
            self,
            "LambdaFunction",
            runtime=lambda_.Runtime.NODEJS_18_X,
            handler="handler",
            code=lambda_.Code.from_asset("./start-state-machine"),
            environment={
                "STATE_MACHINE_ARN": state_machine.state_machine_arn,
            },
        )
        state_machine.grant_start_execution(func)

        subscription = sns_subscriptions.LambdaSubscription(func)
        for topic in topics:
            topic.add_subscription(subscription)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.sns.ITopicSubscription;
```

```
import software.amazon.awscdk.services.sns.Topic;
import software.amazon.awscdk.services.sns.subscriptions.LambdaSubscription;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;

import java.util.Collections;
import java.util.List;

public class StateMachineStack extends Stack {
    public StateMachineStack(final Construct scope, final String id, final
 List<Topic> topics) {
        this(scope, id, null, topics);
    }

    public StateMachineStack(final Construct scope, final String id, final
 StackProps props, final List<Topic> topics) {
        super(scope, id, props);

        // In the future this state machine will do some work...
        final StateMachine stateMachine = StateMachine.Builder.create(this,
 "StateMachine")
                .definition(new Pass(this, "StartState"))
                .build();

        // This Lambda function starts the state machine.
        final Function func = Function.Builder.create(this, "LambdaFunction")
                .runtime(Runtime.NODEJS_18_X)
                .handler("handler")
                .code(Code.fromAsset("./start-state-machine"))
                .environment(Collections.singletonMap("STATE_MACHINE_ARN",
 stateMachine.getStateMachineArn()))
                .build();
        stateMachine.grantStartExecution(func);

        final ITopicSubscription subscription = new LambdaSubscription(func);
        for (final Topic topic : topics) {
            topic.addSubscription(subscription);
        }
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.AWS.SNS.Subscriptions;
using Constructs;

using System.Collections.Generic;

namespace AwsCdkAssertionSamples
{
    public class StateMachineStackProps : StackProps
    {
        public Topic[] Topics;
    }

    public class StateMachineStack : Stack
    {

        internal StateMachineStack(Construct scope, string id,
 StateMachineStackProps props = null) : base(scope, id, props)
        {
            // In the future this state machine will do some work...
            var stateMachine = new StateMachine(this, "StateMachine", new
 StateMachineProps
            {
                Definition = new Pass(this, "StartState")
            });

            // This Lambda function starts the state machine.
            var func = new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "handler",
                Code = Code.FromAsset("./start-state-machine"),
                Environment = new Dictionary<string, string>
                {
                    { "STATE_MACHINE_ARN", stateMachine.StateMachineArn }
                }
            });
            stateMachine.GrantStartExecution(func);
```

```
            foreach (Topic topic in props?.Topics ?? new Topic[0])
            {
                var subscription = new LambdaSubscription(func);
            }

        }
    }
}
```

スタックを実際にインスタンス化しないように、アプリケーションのメインエントリポイントを変更します。誤ってデプロイしたくありません。テストでは、テスト用のアプリケーションとスタックのインスタンスが作成されます。これは、テスト駆動型開発と組み合わせると便利な戦術です。デプロイを有効にする前に、スタックがすべてのテストに合格していることを確認してください。

TypeScript

Eclipse `bin/state-machine.ts`:

```
#!/usr/bin/env node
import * as cdk from "aws-cdk-lib";

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

JavaScript

Eclipse `bin/state-machine.js`:

```
#!/usr/bin/env node
const cdk = require("aws-cdk-lib");

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

Python

Eclipse `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk  as cdk

app = cdk.App()

# Stacks are intentionally not created here -- this application isn't meant to
# be deployed.

app.synth()
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.amazon.awscdk.App;

public class SampleApp {
    public static void main(final String[] args) {
        App app = new App();

        // Stacks are intentionally not created here -- this application isn't meant
 to be deployed.

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;

namespace AwsCdkAssertionSamples
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();

            // Stacks are intentionally not created here -- this application isn't
 meant to be deployed.
```

```
            app.Synth();
        }
    }
}
```

Lambda 関数

サンプルスタックには、ステートマシンを起動する Lambda 関数が含まれています。CDK が Lambda 関数リソースの作成の一環としてバンドルしてデプロイできるように、この関数のソースコードを指定する必要があります。

- アプリのメインディレクトリ`start-state-machine`に フォルダを作成します。
- このフォルダに、少なくとも 1 つのファイルを作成します。例えば、次のコードを に保存できます`start-state-machines/index.js`。

```
exports.handler = async function (event, context) {
 return 'hello world';
};
```

ただし、実際にスタックをデプロイするわけではないため、どのファイルでも機能します。

テストを実行する

参考までに、 AWS CDK アプリでテストを実行するために使用するコマンドを次に示します。これらは、同じテストフレームワークを使用して任意のプロジェクトでテストを実行するときに使用するコマンドと同じです。ビルドステップが必要な言語の場合は、それを含めて、テストがコンパイルされていることを確認します。

TypeScript

```
tsc && npm test
```

JavaScript

```
npm test
```

Python

```
python -m pytest
```

Java

```
mvn compile && mvn test
```

C#

ソリューション (F6) を構築してテストを検出し、テストを実行します (テスト > すべてのテストを実行) 。実行するテストを選択するには、Test Explorer (テスト > Test Explorer) を開きます。

または:

```
dotnet test src
```

きめ細かなアサーション

きめ細かなアサーションを使用してスタックをテストする最初のステップは、生成された AWS CloudFormation テンプレートに対してアサーションを記述するため、スタックを合成することです。

ではStateMachineStackStack、ステートマシンに転送される Amazon SNS トピックを渡す必要があります。そのため、テストでは、トピックを含む別のスタックを作成します。

通常、CDK アプリケーションを記述するときに、スタックのコンストラクタで Amazon SNS トピックをサブクラスStack化してインスタンス化できます。テストでは、 Stack を直接インスタンス化し、このスタックを Topicのスコープとして渡して、スタックにアタッチします。これは機能的に同等であり、冗長性が低くなります。また、テストでのみ使用されるスタックを、デプロイする予定のスタックとは「外観が異なる」ようにすることもできます。

TypeScript

```
import { Capture, Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import { StateMachineStack } from "../lib/state-machine-stack";
```

```
describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const stateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);


}
```

JavaScript

```
const { Capture, Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const { StateMachineStack } = require("../lib/state-machine-stack");

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];
```

```
    // Create the StateMachineStack.
    const StateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
```

Python

```
from aws_cdk import aws_sns as sns
import aws_cdk as cdk
from aws_cdk.assertions import Template

from app.state_machine_stack import StateMachineStack

def test_synthesizes_properly():
    app = cdk.App()

    # Since the StateMachineStack consumes resources from a separate stack
    # (cross-stack references), we create a stack for our SNS topics to live
    # in here. These topics can then be passed to the StateMachineStack later,
    # creating a cross-stack reference.
    topics_stack = cdk.Stack(app, "TopicsStack")

    # Create the topic the stack we're testing will reference.
    topics = [sns.Topic(topics_stack, "Topic1")]

    # Create the StateMachineStack.
    state_machine_stack = StateMachineStack(
        app, "StateMachineStack", topics=topics  # Cross-stack reference
    )

    # Prepare the stack for assertions.
    template = Template.from_stack(state_machine_stack)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import software.amazon.awscdk.assertions.Capture;
```

```
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.services.sns.Topic;

import java.util.*;

import static org.assertj.core.api.Assertions.assertThat;

public class StateMachineStackTest {
    @Test
    public void testSynthesizesProperly() {
        final App app = new App();

        // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
        // for our SNS topics to live in here. These topics can then be passed to
 the StateMachineStack later, creating a
        // cross-stack reference.
        final Stack topicsStack = new Stack(app, "TopicsStack");

        // Create the topic the stack we're testing will reference.
        final List<Topic> topics =
 Collections.singletonList(Topic.Builder.create(topicsStack, "Topic1").build());

        // Create the StateMachineStack.
        final StateMachineStack stateMachineStack = new StateMachineStack(
                app,
                "StateMachineStack",
                topics // Cross-stack reference
        );

        // Prepare the stack for assertions.
        final Template template = Template.fromStack(stateMachineStack)
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.Assertions;
```

```
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest
    {
        [TestMethod]
        public void TestMethod1()
        {
            var app = new App();

            // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
            // for our SNS topics to live in here. These topics can then be passed
 to the StateMachineStack later, creating a
            // cross-stack reference.
            var topicsStack = new Stack(app, "TopicsStack");

            // Create the topic the stack we're testing will reference.
            var topics = new Topic[] { new Topic(topicsStack, "Topic1") };

            // Create the StateMachineStack.
            var StateMachineStack = new StateMachineStack(app, "StateMachineStack",
 new StateMachineStackProps
            {
                Topics = topics
            });

            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // test will go here
        }
    }
}
```

これで、Lambda 関数と Amazon SNS サブスクリプションが作成されたとアサートできます。

TypeScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

JavaScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

Python

```
# Assert that we have created the function with the correct properties
    template.has_resource_properties(
        "AWS::Lambda::Function",
        {
            "Handler": "handler",
            "Runtime": "nodejs14.x",
        },
    )

    # Assert that we have created a subscription
    template.resource_count_is("AWS::SNS::Subscription", 1)
```

Java

```
        // Assert it creates the function with the correct properties...
        template.hasResourceProperties("AWS::Lambda::Function", Map.of(
                "Handler", "handler",
                "Runtime", "nodejs14.x"
```

```
        ));

         // Creates the subscription...
        template.resourceCountIs("AWS::SNS::Subscription", 1);
```

C#

```
            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // Assert it creates the function with the correct properties...
            template.HasResourceProperties("AWS::Lambda::Function", new StringDict {
                { "Handler", "handler"},
                { "Runtime", "nodejs14x" }
            });

            // Creates the subscription...
            template.ResourceCountIs("AWS::SNS::Subscription", 1);
```

Lambda 関数テストでは、関数リソースの 2 つの特定のプロパティに特定の値があることをアサートします。デフォルトでは、 hasResourcePropertiesメソッドは合成された CloudFormation テンプレートで指定されたリソースのプロパティに対して部分一致を実行します。このテストでは、指定されたプロパティが存在し、指定された値を持っている必要がありますが、リソースにはテストされていない他のプロパティを含めることもできます。

Amazon SNS アサーションは、合成されたテンプレートにサブスクリプションが含まれていることをアサートしますが、サブスクリプション自体に関するものではありません。このアサーションは、主にリソース数をアサートする方法を説明するために含めました。Template クラスには、CloudFormation テンプレートの Resources、、Outputsおよび Mappingセクションに対してアサーションを書き込むためのより具体的なメソッドが用意されています。

マッチャー

のデフォルトの部分一致動作hasResourcePropertiesは、 Match クラスのマッチャーを使用して変更できます。

マッチャーの範囲は、寛容 (Match.anyValue) から厳密 () ですMatch.objectEquals。これらは、リソースプロパティの異なる部分に異なるマッチングメソッドを適用するためにネストできます。例えば、 Match.objectEqualsと をMatch.anyValue一緒に使用すると、ステートマシンの

IAM ロールをより完全にテストできますが、変更される可能性のあるプロパティに特定の値は必要ありません。

TypeScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
  "AWS::IAM::Role",
  Match.objectEquals({
    AssumeRolePolicyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "sts:AssumeRole",
          Effect: "Allow",
          Principal: {
            Service: {
              "Fn::Join": [
                "",
                ["states.", Match.anyValue(), ".amazonaws.com"],
              ],
            },
          },
        },
      ],
    },
  })
);
```

JavaScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
  "AWS::IAM::Role",
  Match.objectEquals({
    AssumeRolePolicyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "sts:AssumeRole",
          Effect: "Allow",
          Principal: {
            Service: {
```

```
                "Fn::Join": [
                  "",
                  ["states.", Match.anyValue(), ".amazonaws.com"],
                ],
              },
            },
          },
        ],
      },
    })
  );
```

Python

```
from aws_cdk.assertions import Match

    # Fully assert on the state machine's IAM role with matchers.
    template.has_resource_properties(
        "AWS::IAM::Role",
        Match.object_equals(
            {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Action": "sts:AssumeRole",
                            "Effect": "Allow",
                            "Principal": {
                                "Service": {
                                    "Fn::Join": [
                                        "",
                                        [
                                            "states.",
                                            Match.any_value(),
                                            ".amazonaws.com",
                                        ],
                                    ],
                                },
                            },
                        },
                    ],
                },
            }
```

```
        ),
    )
```

Java

```
        // Fully assert on the state machine's IAM role with matchers.
        template.hasResourceProperties("AWS::IAM::Role", Match.objectEquals(
                Collections.singletonMap("AssumeRolePolicyDocument", Map.of(
                        "Version", "2012-10-17",
                        "Statement", Collections.singletonList(Map.of(
                                "Action", "sts:AssumeRole",
                                "Effect", "Allow",
                                "Principal", Collections.singletonMap(
                                        "Service", Collections.singletonMap(
                                                "Fn::Join", Arrays.asList(
                                                        "",
                                                        Arrays.asList("states.",
Match.anyValue(), ".amazonaws.com")
                                                )
                                        )
                                )
                        ))
                ))
        ));
```

C#

```
            // Fully assert on the state machine's IAM role with matchers.
            template.HasResource("AWS::IAM::Role", Match.ObjectEquals(new ObjectDict
            {
                { "AssumeRolePolicyDocument", new ObjectDict
                    {
                        { "Version", "2012-10-17" },
                        { "Action", "sts:AssumeRole" },
                        { "Principal", new ObjectDict
                            {
                                { "Version", "2012-10-17" },
                                { "Statement", new object[]
                                    {
                                        new ObjectDict {
                                            { "Action", "sts:AssumeRole" },
                                            { "Effect", "Allow" },
                                            { "Principal", new ObjectDict
```

```
                                                {
                                                    { "Service", new ObjectDict
                                                        {
                                                            { "", new object[]
                                                                { "states",
Match.AnyValue(), ".amazonaws.com" }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }));
```

多くの CloudFormation リソースには、文字列として表されるシリアル化された JSON オブジェクトが含まれます。`Match.serializedJson()` マッチャーは、この JSON 内のプロパティを照合するために使用できます。

例えば、Step Functions ステートマシンは、JSON ベースの Amazon States Language の文字列を使用して定義されます。を使用して`Match.serializedJson()`、最初の状態が唯一のステップであることを確認します。ここでも、ネストされたマッチャーを使用して、オブジェクトのさまざまな部分にさまざまな種類のマッチングを適用します。

TypeScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
```

```
            End: true,
            // Make sure this state doesn't provide a next state -- we can't
            // provide both Next and set End to true.
            Next: Match.absent(),
          },
        },
      })
    ),
  });
```

JavaScript

```
  // Assert on the state machine's definition with the Match.serializedJson()
  // matcher.
  template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
    DefinitionString: Match.serializedJson(
      // Match.objectEquals() is used implicitly, but we use it explicitly
      // here for extra clarity.
      Match.objectEquals({
        StartAt: "StartState",
        States: {
          StartState: {
            Type: "Pass",
            End: true,
            // Make sure this state doesn't provide a next state -- we can't
            // provide both Next and set End to true.
            Next: Match.absent(),
          },
        },
      })
    ),
  });
```

Python

```
    # Assert on the state machine's definition with the serialized_json matcher.
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                # Match.object_equals() is the default, but specify it here for
clarity
                Match.object_equals(
```

```
                    {
                        "StartAt": "StartState",
                        "States": {
                            "StartState": {
                                "Type": "Pass",
                                "End": True,
                                # Make sure this state doesn't provide a next state
--
                                # we can't provide both Next and set End to true.
                                "Next": Match.absent(),
                            },
                        },
                    }
                )
            ),
        },
    )
```

Java

```
        // Assert on the state machine's definition with the Match.serializedJson()
matcher.
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                        Match.objectEquals(Map.of(
                                "StartAt", "StartState",
                                "States", Collections.singletonMap(
                                        "StartState", Map.of(
                                                "Type", "Pass",
                                                "End", true,
                                                // Make sure this state doesn't
provide a next state -- we can't provide
                                                // both Next and set End to true.
                                                "Next", Match.absent()
                                        )
                                )
                        ))
                )
        ));
```

C#

```
            // Assert on the state machine's definition with the
Match.serializedJson() matcher
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                    Match.ObjectEquals(new ObjectDict {
                        { "StartAt", "StartState" },
                        { "States", new ObjectDict
                        {
                            { "StartState", new ObjectDict {
                                { "Type", "Pass" },
                                { "End", "True" },
                                // Make sure this state doesn't provide a next state
-- we can't provide
                                // both Next and set End to true.
                                { "Next", Match.Absent() }
                            }}
                        }}
                    })
                )}});
```

キャプチャ

多くの場合、プロパティをテストして、特定の形式に従っているか、別のプロパティと同じ値を持っていることを確認すると、正確な値を事前に知る必要がなくなります。assertions モジュールは、 Capture クラスでこの機能を提供します。

の値の代わりにCaptureインスタンスを指定することでhasResourceProperties、その値は Capture オブジェクトに保持されます。実際のキャプチャ値は、、、asNumber()asString()および を含むオブジェクトの asメソッドを使用して取得できasObject、テストの対象となります。をマッチャーCaptureとともに使用して、シリアル化された JSON プロパティなど、リソースのプロパティ内でキャプチャされる値の正確な場所を指定します。

次の例では、ステートマシンの開始状態が で始まる名前になっていることをテストしますStart。また、この状態がマシン内の状態のリスト内に存在することをテストします。

TypeScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
      StartAt: startAtCapture,
      States: statesCapture,
    })
  ),
});

// Assert that the start state starts with "Start".
expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the
// state machine definition.
expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

JavaScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
      StartAt: startAtCapture,
      States: statesCapture,
    })
  ),
});

// Assert that the start state starts with "Start".
expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the
// state machine definition.
expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

Python

```
import re

    from aws_cdk.assertions import Capture

    # ...

    # Capture some data from the state machine's definition.
    start_at_capture = Capture()
    states_capture = Capture()
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                Match.object_like(
                    {
                        "StartAt": start_at_capture,
                        "States": states_capture,
                    }
                )
            ),
        },
    )

    # Assert that the start state starts with "Start".
    assert re.match("^Start", start_at_capture.as_string())

    # Assert that the start state actually exists in the states object of the
    # state machine definition.
    assert start_at_capture.as_string() in states_capture.as_object()
```

Java

```
        // Capture some data from the state machine's definition.
        final Capture startAtCapture = new Capture();
        final Capture statesCapture = new Capture();
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
 Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        Match.objectLike(Map.of(
                                "StartAt", startAtCapture,
                                "States", statesCapture
```

```
                        ))
                )
        ));

        // Assert that the start state starts with "Start".
        assertThat(startAtCapture.asString()).matches("^Start.+");

        // Assert that the start state actually exists in the states object of the
state machine definition.
        assertThat(statesCapture.asObject()).containsKey(startAtCapture.asString());
```

C#

```
            // Capture some data from the state machine's definition.
            var startAtCapture = new Capture();
            var statesCapture = new Capture();
            template.HasResourceProperties("AWS::StepFunctions::StateMachine", new
ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    new ObjectDict
                    {
                        { "StartAt", startAtCapture },
                        { "States", statesCapture }
                    }
                )}
            });

            Assert.IsTrue(startAtCapture.ToString().StartsWith("Start"));

Assert.IsTrue(statesCapture.AsObject().ContainsKey(startAtCapture.ToString()));
```

スナップショットテスト

スナップショットテストでは、合成された CloudFormation テンプレート全体を、以前に保存したベースライン (多くの場合、「マスター」と呼ばれます) テンプレートと比較します。きめ細かなアサーションとは異なり、スナップショットテストはリグレッションのキャッチには役立ちません。これは、スナップショットテストがテンプレート全体に適用され、コード変更以外のことが合成結果に小さな (または not-so-small) 差異を引き起こす可能性があるためです。これらの変更はデプロイには影響しないかもしれませんが、スナップショットテストは失敗します。

例えば、CDK コンストラクトを更新して新しいベストプラクティスを組み込むと、合成されたリソースやそれらの構成に変更が生じる可能性があります。または、CDK ツールキットを、追加のメタデータをレポートするバージョンに更新することもできます。コンテキスト値を変更すると、合成されたテンプレートにも影響する可能性があります。

ただし、合成されたテンプレートに影響を与える可能性のある他のすべての要素を一定に維持すれば、スナップショットテストはリファクタリングに非常に役立ちます。行った変更がテンプレートを意図せずに変更したかどうかはすぐにわかります。変更が意図的なものである場合は、新しいテンプレートをベースラインとして受け入れるだけです。

例えば、次のDeadLetterQueueコンストラクトがあるとします。

TypeScript

```
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}
```

JavaScript

```
class DeadLetterQueue extends sqs.Queue {

  constructor(scope, id) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
```

```
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}

module.exports = { DeadLetterQueue }
```

Python

```
class DeadLetterQueue(sqs.Queue):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        self.messages_in_queue_alarm = cloudwatch.Alarm(
            self,
            "Alarm",
            alarm_description="There are messages in the Dead Letter Queue.",
            evaluation_periods=1,
            threshold=1,
            metric=self.metric_approximate_number_of_messages_visible(),
        )
```

Java

```
public class DeadLetterQueue extends Queue {
    private final IAlarm messagesInQueueAlarm;

    public DeadLetterQueue(@NotNull Construct scope, @NotNull String id) {
        super(scope, id);

        this.messagesInQueueAlarm = Alarm.Builder.create(this, "Alarm")
                .alarmDescription("There are messages in the Dead Letter Queue.")
                .evaluationPeriods(1)
                .threshold(1)
                .metric(this.metricApproximateNumberOfMessagesVisible())
                .build();
    }

    public IAlarm getMessagesInQueueAlarm() {
        return messagesInQueueAlarm;
    }
}
```

C#

```
namespace AwsCdkAssertionSamples
{
    public class DeadLetterQueue : Queue
    {
        public IAlarm messagesInQueueAlarm;

        public DeadLetterQueue(Construct scope, string id) : base(scope, id)
        {
            messagesInQueueAlarm = new Alarm(this, "Alarm", new AlarmProps
            {
                AlarmDescription = "There are messages in the Dead Letter Queue.",
                EvaluationPeriods = 1,
                Threshold = 1,
                Metric = this.MetricApproximateNumberOfMessagesVisible()
            });
        }
    }
}
```

次のようにテストできます。

TypeScript

```
import { Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import { DeadLetterQueue } from "../lib/dead-letter-queue";

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

JavaScript

```
const { Match, Template } = require("aws-cdk-lib/assertions");
```

```
const cdk = require("aws-cdk-lib");
const { DeadLetterQueue } = require("../lib/dead-letter-queue");

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

Python

```
import aws_cdk_lib as cdk
from aws_cdk_lib.assertions import Match, Template

from app.dead_letter_queue import DeadLetterQueue

def snapshot_test():
    stack = cdk.Stack()
    DeadLetterQueue(stack, "DeadLetterQueue")

    template = Template.from_stack(stack)
    assert template.to_json() == snapshot
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import au.com.origin.snapshots.Expect;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.Stack;

import java.util.Collections;
import java.util.Map;

public class DeadLetterQueueTest {
    @Test
```

```
    public void snapshotTest() {
        final Stack stack = new Stack();
        new DeadLetterQueue(stack, "DeadLetterQueue");

        final Template template = Template.fromStack(stack);
        expect.toMatchSnapshot(template.toJSON());
    }
}
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest

    [TestClass]
    public class DeadLetterQueueTest
    {
    [TestMethod]
        public void SnapshotTest()
        {
            var stack = new Stack();
            new DeadLetterQueue(stack, "DeadLetterQueue");

            var template = Template.FromStack(stack);

            return Verifier.Verify(template.ToJSON());
        }
    }
}
```

テストのヒント

テストはテストするコードと同じ期間存続し、同じ頻度で読み取られ、変更されることに注意してください。したがって、それらを書き込むのに最適な方法を検討するために少し時間がかかることになります。

セットアップラインや一般的なアサーションをコピーして貼り付けないでください。代わりに、このロジックをフィクスチャまたはヘルパー関数にリファクタリングします。各テストが実際にテストしたものを反映した良い名前を使用してください。

1 回のテストでやりすぎないでください。テストでは、1 つの動作のみをテストする必要があります。この動作を誤って中断した場合、1 つのテストだけが失敗し、テストの名前に失敗した内容が示されます。ただし、これは、複数の動作をテストするテストをやむを得ず (または誤って) 書き込む場合もあり、これを試みるのが理想的です。スナップショットテストは、すでに説明した理由で、特にこの問題が発生しやすいため、慎重に使用してください。

のセキュリティ AWS Cloud Development Kit (AWS CDK)

クラウドセキュリティは Amazon Web Services (AWS) の最優先事項です。お客様は AWS 、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャからメリットを得られます。セキュリティは、 AWS とユーザーの間で共有される責任です。責任共有モデルでは、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

クラウドのセキュリティ — AWS クラウドで提供されるすべてのサービスを実行するインフラストラクチャ AWS を保護し、安全に使用できるサービスを提供します。当社のセキュリティ責任は で最優先事項であり AWS、当社のセキュリティの有効性は、AWS コンプライアンスプログラム の一環として、サードパーティーの監査者によって定期的にテストおよび検証されています。

クラウドにおけるセキュリティ — お客様の責任は、使用しているサービス、およびデータの機密性、組織の要件、適用される法律や規制などのその他の要因によって AWS 決まります。

は、サポートする特定の Amazon Web Services (AWS) サービスを通じて責任共有モデル AWS CDK に従います。 AWS サービスセキュリティ情報については、AWS 「サービスセキュリティドキュメント」ページとAWS 、コンプライアンスプログラム によるコンプライアンスの取り組みの対象となる AWS サービスを参照してください。

トピック

- の Identity and Access Management AWS Cloud Development Kit (AWS CDK)
- のコンプライアンス検証 AWS Cloud Development Kit (AWS CDK)
- の耐障害性 AWS Cloud Development Kit (AWS CDK)
- のインフラストラクチャセキュリティ AWS Cloud Development Kit (AWS CDK)

の Identity and Access Management AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービス するのに役立つ です。IAM 管理者は、誰を認証 (サインイン) し、誰に AWS リソースの使用を承認する (アクセス許可を付与する) かを制御します。IAM は、追加料金なしで AWS のサービス 使用できる です。

対象者

AWS Identity and Access Management (IAM) の使用方法は、 で行う作業によって異なります AWS。

サービスユーザー – AWS のサービス を使用してジョブを実行する場合、管理者から必要な認証情報とアクセス許可が与えられます。さらに多くの AWS 機能を使用して作業を行う場合は、追加のアクセス許可が必要になることがあります。アクセスの管理方法を理解しておくと、管理者に適切な許可をリクエストするうえで役立ちます。

サービス管理者 – 社内の AWS リソースを担当している場合は、通常、 AWS リソースへのフルアクセスがあります。サービスユーザーがどの AWS のサービス とリソースにアクセスするかを決めるのは管理者の仕事です。その後、IAM 管理者にリクエストを送信して、サービスユーザーの権限を変更する必要があります。このページの情報を点検して、IAM の基本概念を理解してください。

IAM 管理者 - 管理者は、 AWS のサービスへのアクセスを管理するポリシーの書き込み方法の詳細について確認する場合があります。

アイデンティティを使用した認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。として、IAM ユーザーとして AWS アカウントのルートユーザー、または IAM ロールを引き受けて認証 (にサインイン AWS) される必要があります。

ID ソースを介して提供された認証情報を使用して、フェデレーティッド ID AWS として にサインインできます。 AWS IAM Identity Center （ IAM Identity Center) ユーザー、会社のシングルサインオン認証、Google または Facebook の認証情報は、フェデレーション ID の例です。フェデレーティッドアイデンティティとしてサインインする場合、IAM ロールを使用して、前もって管理者により ID フェデレーションが設定されています。フェデレーション AWS を使用して にアクセスすると、間接的にロールを引き受けることになります。

ユーザーのタイプに応じて、 AWS Management Console または AWS アクセスポータルにサインインできます。へのサインインの詳細については AWS、「 ユーザーガイド」の「 へのサインイン AWS アカウント方法AWS サインイン 」を参照してください。

AWS プログラムで にアクセスするには、、 Software Development Kit (SDKs) AWS CDK、およびコマンドラインインターフェイス (CLI) AWS を提供し、認証情報を使用してリクエストに暗号で署名します。 AWS ツールを使用しない場合は、リクエストに自分で署名する必要があります。推奨される方法を使用して自分でリクエストに署名する方法の詳細については、「AWS 全般のリファレンス」の「署名バージョン 4 の署名プロセス」を参照してください。

使用する認証方法を問わず、追加セキュリティ情報の提供をリクエストされる場合もあります。例えば、 AWS では、多要素認証 (MFA) を使用してアカウントのセキュリティを向上させることをお勧めします。詳細については、『AWS IAM Identity Center ユーザーガイド』の「Multi-factor authentication」(多要素認証) および『IAM ユーザーガイド』の「AWSにおける多要素認証 (MFA) の使用」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、アカウント内のすべての AWS のサービス およびリソースへの完全なアクセス権を持つ 1 つのサインインアイデンティティから始めます。この ID は AWS アカウント ルートユーザーと呼ばれ、アカウントの作成に使用した E メールアドレスとパスワードでサインインすることでアクセスできます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザーの認証情報は保護し、ルートユーザーでしか実行できないタスクを実行するときに使用します。ルートユーザーとしてサインインする必要があるタスクの完全なリストについては、『IAM ユーザーガイド』の「ルートユーザー認証情報が必要なタスク」を参照してください。

フェデレーティッドアイデンティティ

ベストプラクティスとして、管理者アクセスを必要とするユーザーを含む人間のユーザーに、一時的な認証情報を使用して にアクセスするための ID プロバイダーとのフェデレーションの使用を要求 AWS のサービス します。

フェデレーティッド ID は、エンタープライズユーザーディレクトリ、ウェブ ID プロバイダー、AWS Directory Service、Identity Center ディレクトリのユーザー、または ID ソースを通じて提供された認証情報 AWS のサービス を使用して にアクセスするユーザーです。フェデレーティッド ID がにアクセスすると AWS アカウント、ロールを引き受け、ロールは一時的な認証情報を提供します。

アクセスを一元管理する場合は、 AWS IAM Identity Centerを使用することをお勧めします。IAM Identity Center でユーザーとグループを作成することも、独自の ID ソース内のユーザーとグループのセットに接続して同期して、すべての AWS アカウント とアプリケーションで使用することもできます。IAM Identity Center の詳細については、『AWS IAM Identity Center ユーザーガイド』の「What is IAM Identity Center?」(IAM Identity Center とは) を参照してください。

IAM ユーザーとグループ

IAM ユーザーは、単一のユーザーまたはアプリケーションに対して特定のアクセス許可 AWS アカウント を持つ 内のアイデンティティです。可能であれば、パスワードやアクセスキーなどの長期的な認証情報を保有する IAM ユーザーを作成する代わりに、一時認証情報を使用することをお勧めしま

す。ただし、IAM ユーザーでの長期的な認証情報が必要な特定のユースケースがある場合は、アクセスキーをローテーションすることをお勧めします。詳細については、IAM ユーザーガイドの「長期的な認証情報を必要とするユースケースのためにアクセスキーを定期的にローテーションする」を参照してください。

IAM グループは、IAM ユーザーの集団を指定するアイデンティティです。グループとしてサインインすることはできません。グループを使用して、複数のユーザーに対して一度に権限を指定できます。多数のユーザーグループがある場合、グループを使用することで権限の管理が容易になります。例えば、IAMAdmins という名前のグループを設定して、そのグループに IAM リソースを管理する権限を与えることができます。

ユーザーは、ロールとは異なります。ユーザーは 1 人の人または 1 つのアプリケーションに一意に関連付けられますが、ロールはそれを必要とする任意の人が引き受けるようになっています。ユーザーには永続的な長期の認証情報がありますが、ロールでは一時的な認証情報が提供されます。詳細については、『IAM ユーザーガイド』の「IAM ユーザー (ロールではなく) の作成が適している場合」を参照してください。

IAM ロール

IAM ロールは、特定のアクセス許可 AWS アカウント を持つ 内のアイデンティティです。これは IAM ユーザーに似ていますが、詳細のユーザーに関連付けられていません。ロール を切り替える AWS Management Console ことで、 で IAM ロールを一時的に引き受けることができます。ロールを引き受けるには、 AWS CLI または AWS API オペレーションを呼び出すか、カスタム URL を使用します。ロールを使用する方法の詳細については、「IAM ユーザーガイド」の「IAM ロールの使用」を参照してください。

IAM ロールと一時的な認証情報は、次の状況で役立ちます:

- フェデレーションユーザーアクセス – フェデレーティッドアイデンティティに権限を割り当てるには、ロールを作成してそのロールの権限を定義します。フェデレーティッドアイデンティティが認証されると、そのアイデンティティはロールに関連付けられ、ロールで定義されている権限が付与されます。フェデレーションの詳細については、『IAM ユーザーガイド』の「サードパーティーアイデンティティプロバイダー向けロールの作成」 を参照してください。IAM アイデンティティセンターを使用する場合、権限セットを設定します。アイデンティティが認証後にアクセスできるものを制御するため、IAM Identity Center は、権限セットを IAM のロールに関連付けます。権限セットの詳細については、『AWS IAM Identity Center ユーザーガイド』の「権限セット」を参照してください。
- 一時的な IAM ユーザー権限 - IAM ユーザーまたはロールは、特定のタスクに対して複数の異なる権限を一時的に IAM ロールで引き受けることができます。

- クロスアカウントアクセス - IAM ロールを使用して、自分のアカウントのリソースにアクセスすることを、別のアカウントの人物 (信頼済みプリンシパル) に許可できます。クロスアカウントアクセス権を付与する主な方法は、ロールを使用することです。ただし、一部の では AWS のサービス、(ロールをプロキシとして使用する代わりに) ポリシーをリソースに直接アタッチできます。クロスアカウントアクセスにおけるロールとリソースベースのポリシーの違いについては、『IAM ユーザーガイド』の「IAM ロールとリソースベースのポリシーとの相違点」を参照してください。
- クロスサービスアクセス — 一部の は、他の の機能 AWS のサービス を使用します AWS のサービス。例えば、あるサービスで呼び出しを行うと、通常そのサービスによって Amazon EC2 でアプリケーションが実行されたり、Amazon S3 にオブジェクトが保存されたりします。サービスでは、呼び出し元プリンシパルの権限、サービスロール、またはサービスにリンクされたロールを使用してこれを行う場合があります。
  - サービスロール - サービスがユーザーに代わってアクションを実行するために引き受ける IAM ロールです。IAM 管理者は、IAM 内からサービスロールを作成、変更、削除できます。詳細については、「IAM ユーザーガイド」の「AWS のサービスにアクセス許可を委任するロールの作成」を参照してください。
  - サービスにリンクされたロール – サービスにリンクされたロールは、 にリンクされたサービスロールの一種です AWS のサービス。サービスは、ユーザーに代わってアクションを実行するロールを引き受けることができます。サービスにリンクされたロールは に表示され AWS アカウント 、サービスによって所有されます。IAM 管理者は、サービスにリンクされたロールの権限を表示できますが、編集することはできません。
- Amazon EC2 で実行されているアプリケーション – IAM ロールを使用して、EC2 インスタンスで実行され、 AWS CLI または AWS API リクエストを行うアプリケーションの一時的な認証情報を管理できます。これは、EC2 インスタンス内でのアクセスキーの保存に推奨されます。 AWS ロールを EC2 インスタンスに割り当て、そのすべてのアプリケーションで使用できるようにするには、インスタンスにアタッチされたインスタンスプロファイルを作成します。インスタンスプロファイルにはロールが含まれ、EC2 インスタンスで実行されるプログラムは一時的な認証情報を取得できます。詳細については、『IAM ユーザーガイド』の「Amazon EC2 インスタンスで実行されるアプリケーションに IAM ロールを使用して権限を付与する」を参照してください。

IAM ロールと IAM ユーザーのどちらを使用するかについては、『IAM ユーザーガイド』の「(IAM ユーザーではなく) IAM ロールをいつ作成したら良いのか?」を参照してください。

のコンプライアンス検証 AWS Cloud Development Kit (AWS CDK)

は、サポートする特定の Amazon Web Services (AWS) サービスを通じて責任共有モデル AWS CDK に従います。 AWS サービスセキュリティ情報については、AWS 「サービスセキュリティドキュメント」ページとAWS 、コンプライアンスプログラム によるコンプライアンスの取り組みの対象となる AWS サービスを参照してください。

AWS サービスのセキュリティとコンプライアンスは、複数の AWS コンプライアンス プログラムの一環として、サードパーティーの監査者によって評価されます。これには、SOC、PCI、FedRAMP、HIPAA などが含まれます。 は、コンプライアンスプログラム による対象範囲内の AWS のサービスで、特定のコンプライアンスプログラムAWS の範囲内で頻繁に更新される のサービスのリスト AWS を提供します。

AWS Artifact を使用してサードパーティーの監査レポートをダウンロードすることができます。詳細については、「 でのレポートのダウンロード AWS Artifact」を参照してください。

AWS コンプライアンスプログラムの詳細については、AWS 「 コンプライアンスプログラム」を参照してください。

を使用して AWS サービス AWS CDK にアクセスする場合のお客様のコンプライアンス責任は、お客様のデータの機密性、組織のコンプライアンス目的、適用可能な法律および規制によって決まります。サービスの使用 AWS が HIPAA、PCI、FedRAMP などの標準に準拠していることを前提としている場合、 は以下に役立つリソース AWS を提供します。

- セキュリティとコンプライアンスのクイックスタートガイド – アーキテクチャ上の考慮事項について説明し、セキュリティとコンプライアンスに重点を置いたベースライン環境を にデプロイする手順を説明するデプロイガイド AWS。
- AWS コンプライアンスリソース – お客様の業界や地域に適用される可能性のあるワークブックとガイドのコレクション。
- AWS Config – 自社プラクティス、業界ガイドライン、および規制に対するリソースの設定の準拠状態を評価するサービス。
- AWS Security Hub – セキュリティ業界標準とベストプラクティスへの準拠を確認するのに役立つ AWS 、 内のセキュリティ状態の包括的なビュー。

の耐障害性 AWS Cloud Development Kit (AWS CDK)

Amazon Web Services (AWS) グローバルインフラストラクチャは、 AWS リージョンとアベイラビリティーゾーンを中心に構築されています。

AWS リージョンは、低レイテンシー、高スループット、および高度に冗長なネットワークで接続された、物理的に分離および分離された複数のアベイラビリティーゾーンを提供します。

アベイラビリティーゾーンでは、アベイラビリティーゾーン間で中断せずに、自動的にフェイルオーバーするアプリケーションとデータベースを設計および運用することができます。アベイラビリティーゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、耐障害性、およびスケーラビリティが優れています。

AWS リージョンとアベイラビリティーゾーンの詳細については、AWS 「 グローバルインフラストラクチャ」を参照してください。

は、サポートする特定の Amazon Web Services (AWS) サービスを通じて責任共有モデル AWS CDK に従います。 AWS サービスセキュリティ情報については、AWS 「サービスセキュリティドキュメント」ページとAWS 、コンプライアンスプログラム によるコンプライアンスの取り組みの対象となる AWS サービスを参照してください。

のインフラストラクチャセキュリティ AWS Cloud Development Kit (AWS CDK)

は、サポートする特定の Amazon Web Services (AWS) サービスを通じて責任共有モデル AWS CDK に従います。 AWS サービスセキュリティ情報については、AWS 「サービスセキュリティドキュメント」ページとAWS 、コンプライアンスプログラム によるコンプライアンスの取り組みの対象となる AWS サービスを参照してください。

一般的な AWS CDK 問題のトラブルシューティング

このトピックでは、 に関する以下の問題のトラブルシューティング方法について説明します AWS CDK。

- を更新した後 AWS CDK、 AWS CDK ツールキット (CLI) AWS は コンストラクトライブラリとの不一致を報告します。
- AWS CDK スタックをデプロイするときに NoSuchBucket エラーが表示される
- AWS CDK スタックをデプロイするときに、 というforbidden: nullメッセージが表示されます。
- AWS CDK スタックを合成するときに、 というメッセージが表示されます。 --app is required either in command-line, in cdk.json or in ~/.cdk.json
- AWS CDK スタックを合成するときに、テンプレートに含まれるリソースが多すぎるため AWS CloudFormation 、エラーが発生します。
- Auto Scaling グループまたは VPC に 3 つ (またはそれ以上) のアベイラビリティーゾーンを指定しましたが、デプロイされたのは 2 つのみです
- S3 バケット、DynamoDB テーブル、またはその他のリソースが問題発生時に削除されない cdk destroy

を更新した後 AWS CDK、 AWS CDK ツールキット (CLI) AWS は コンストラクトライブラリとの不一致を報告します。

Toolkit のバージョン AWS CDK (cdk コマンドを提供) は、メインの Construct Library AWS モジュールのバージョン と少なくとも同じである必要がありますaws-cdk-lib。ツールキットは下位互換性を目的としています。ツールキットの最新の 2.x バージョンは、ライブラリの任意の 1.x または 2.x リリースで使用できます。このため、このコンポーネントをグローバルにインストールし、最新の状態に保つことをお勧めします。

```
npm update -g aws-cdk
```

複数のバージョンの AWS CDK Toolkit を使用する必要がある場合は、プロジェクトフォルダにローカルに特定のバージョンのツールキットをインストールします。

TypeScript または を使用している場合 JavaScript、プロジェクトディレクトリには CDK Toolkit のバージョン管理されたローカルコピーが既に含まれています。

別の言語を使用している場合は、 を使用して AWS CDK Toolkit をインストールnpmし、 -gフラグを省略して目的のバージョンを指定します。例:

```
npm install aws-cdk@2.0
```

ローカルにインストールされた AWS CDK Toolkit を実行するには、 npx aws-cdkだけでなく、 コマンドを使用しますcdk。例:

```
npx aws-cdk deploy MyStack
```

npx aws-cdk ローカルバージョンの AWS CDK Toolkit が存在する場合、 はそれを実行します。プロジェクトにローカルインストールがない場合、グローバルバージョンにフォールバックします。cdk 常にこの方法で が呼び出されるようにシェルエイリアスを設定すると便利な場合があります。

macOS/Linux

```
alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

(のリストに戻る)

AWS CDK スタックをデプロイするときに **NoSuchBucket** エラーが表示される

AWS 環境はブートストラップされていないため、デプロイ中にリソースを保持する Amazon S3 バケットがありません。次のコマンドを使用して、ステージングバケットおよびその他の必要なリソースを作成できます。

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

予想外の AWS 料金が発生しないように、 AWS CDK は環境を自動的にブートストラップしません。デプロイ先の各環境を明示的にブートストラップする必要があります。

デフォルトでは、ブートストラップリソースは、現在の AWS CDK アプリケーションのスタックで使用される 1 つまたは複数のリージョンに作成されます。または、ローカル AWS プロファイルで指

定されたリージョン (で設定aws configure) に、そのプロファイルのアカウントを使用して作成されます。次のように、コマンドラインで別のアカウントとリージョンを指定できます。(アプリのディレクトリにない場合は、アカウントとリージョンを指定する必要があります)。

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

詳細については、「the section called “ブートストラッピング”」を参照してください。

(のリストに戻る)

AWS CDK スタックをデプロイするときに、 というforbidden: null**メッセージが表示されます。**

ブートストラップリソースを必要とするスタックをデプロイしていますが、そのスタックに書き込むアクセス許可がない IAM ロールまたはアカウントを使用しています。(ステージングバケットは、アセットを含むスタックや 50K を超える AWS CloudFormation テンプレートを合成するスタックをデプロイするときに使用されます)。エラーメッセージに記載されているバケットs3:*に対してアクションを実行するアクセス許可を持つアカウントまたはロールを使用します。

(のリストに戻る)

AWS CDK スタックを合成するときに、 というメッセージが表示されます。 **--app is required either in command-line, in cdk.json or in ~/.cdk.json**

このメッセージは通常、 を発行したときに AWS CDK プロジェクトのメインディレクトリにないことを意味しますcdk synth。cdk init コマンドによって作成されたcdk.jsonこのディレクトリの ファイルには、 AWS CDK アプリケーションの実行 (および合成) に必要なコマンドラインが含まれています。例えば、 TypeScript アプリの場合、デフォルトは次のcdk.jsonようになります。

```
{
  "app": "npx ts-node bin/my-cdk-app.ts"
}
```

AWS CDK ツールキットがcdk.jsonそこで見つけてアプリケーションを正常に実行できるように、プロジェクトのメインディレクトリでのみcdkコマンドを発行することをお勧めします。

何らかの理由でこれが実用的でない場合、 AWS CDK ツールキットは他の 2 つの場所でアプリケーションのコマンドラインを探します。

- ホームディレクトリcdk.jsonの 内

- `-a` オプションを使用した`cdk synth`コマンド自体で

例えば、次のように TypeScript アプリケーションからスタックを合成できます。

```
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

(のリストに戻る)

AWS CDK スタックを合成するときに、テンプレートに含まれるリソースが多すぎるため AWS CloudFormation 、エラーが発生します。

は AWS CloudFormation templates AWS CDK を生成してデプロイします。 AWS CloudFormation スタックに含めることができるリソースの数には厳しい制限があります。を使用すると AWS CDK、この制限に対して予想よりも迅速に実行できます。

> **Note**
>
> この書き込みの AWS CloudFormation リソース制限は 500 です。現在のリソース制限のAWS CloudFormation クォータを参照してください。

AWS Construct Library の上位レベルのインテントベースのコンストラクトは、ログ記録、キー管理、認可、およびその他の目的に必要な補助リソースを自動的にプロビジョニングします。例えば、あるリソースへのアクセス権を別のリソースに付与すると、関連するサービスが通信するために必要な IAM オブジェクトが生成されます。

経験上、インテントベースのコンストラクトを実際に使用すると、コンストラクトごとに 1～5 AWS CloudFormation リソースになりますが、これは異なる場合があります。サーバーレスアプリケーションでは、API エンドポイントあたり 5～8 AWS リソースが一般的です。

より高いレベルの抽象化を表すパターンを使用すると、コードの数を減らしてさらに多くの AWS リソースを定義できます。例えばthe section called “ECS”、 の AWS CDK コードは、3 つのコンストラクトのみを定義しながら 50 を超える AWS CloudFormation リソースを生成します。

AWS CloudFormation リソース制限を超えると、 AWS CloudFormation 合成中にエラーが発生します。スタックが制限の 80% を超えると AWS CDK 、警告が表示されます。スタックに`maxResources`プロパティを設定することで別の制限を使用することも、 を 0 `maxResources`に設定して検証を無効にすることもできます。

> **Tip**
>
> 次のユーティリティスクリプトを使用して、合成された出力内のリソースの正確な数を取得できます。(すべての AWS CDK デベロッパーには Node.js が必要なため、スクリプトは で記述されます) JavaScript。

```
// rescount.js - count the resources defined in a stack
// invoke with: node rescount.js <path-to-stack-json>
// e.g. node rescount.js cdk.out/MyStack.template.json

import * as fs from 'fs';
const path = process.argv[2];

if (path) fs.readFile(path, 'utf8', function(err, contents) {
  console.log(err ? `${err}` :
  `${Object.keys(JSON.parse(contents).Resources).length} resources defined in
 ${path}`);
}); else console.log("Please specify the path to the stack's output .json
 file");
```

スタックのリソース数が制限に近づいたら、再設計してスタックに含まれるリソースの数を減らすことを検討してください。例えば、いくつかの Lambda 関数を組み合わせたり、スタックを複数のスタックに分割したりします。CDK はスタック 間の参照をサポートしているため、アプリの機能をさまざまなスタックに分割できます。

> **Note**
>
> AWS CloudFormation エキスパートは多くの場合、リソース制限のソリューションとしてネストされたスタックの使用を提案します。は、 `NestedStack`コンストラクトを介してこのアプローチ AWS CDK をサポートします。

(のリストに戻る)

Auto Scaling グループまたは VPC に 3 つ (またはそれ以上) のアベイラビリティーゾーンを指定しましたが、デプロイされたのは 2 つのみです

リクエストするアベイラビリティーゾーンの数を取得するには、スタックの envプロパティでアカウントとリージョンを指定します。両方を指定しない場合、 は AWS CDKデフォルトでスタックを

環境に依存しないものとして合成します。その後、 を使用してスタックを特定のリージョンにデプロイできます AWS CloudFormation。一部のリージョンには 2 つのアベイラビリティーゾーンしかないため、環境に依存しないテンプレートでは 2 つ以上のアベイラビリティーゾーンを使用しません。

> **Note**
>
> 以前は、リージョンは 1 つのアベイラビリティーゾーンのみで起動していました。環境に依存しない AWS CDK スタックをそのようなリージョンにデプロイすることはできません。ただし、この記事では、すべての AWS リージョンに少なくとも 2 つの AZs。

スタックの `availablilityZones` (Python: `availability_zones`) プロパティを上書きして、使用するゾーンを明示的に指定することで、この動作を変更できます。

スタックのアカウントとリージョンを合成時に指定し、どのリージョンにもデプロイできる柔軟性を維持する方法の詳細については、「」を参照してくださいthe section called “環境”。

(のリストに戻る)

S3 バケット、DynamoDB テーブル、またはその他のリソースが問題発生時に削除されない **cdk destroy**

デフォルトでは、ユーザーデータを含むことができるリソースには `removalPolicy` (Python: `removal_policy`) プロパティ がありRETAIN、スタックが破棄されてもリソースは削除されません。代わりに、リソースはスタックから孤立します。その後、スタックが破棄されたら、リソースを手動で削除する必要があります。これを行うまでは、スタックの再デプロイは失敗します。これは、デプロイ中に作成される新しいリソースの名前が、孤立したリソースの名前と競合するためです。

リソースの削除ポリシーを に設定した場合DESTROY、スタックが破棄されると、そのリソースは削除されます。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
```

```
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

```
software.amazon.awscdk.*;
import software.amazon.awscdk.services.s3.*;
```

```
import software.constructs;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY
    });
}
```

> **Note**
>
> AWS CloudFormation は空でない Amazon S3 バケットを削除することはできません。Amazon S3 バケットの削除ポリシーを に設定しDESTROY、データが含まれている場合、バケットを削除できないため、スタックを破棄しようとすると失敗します。バケットの autoDeleteObjectsprop を に設定することで、破棄を試みる前にバケット内のオブジェクト AWS CDK を削除できますtrue。

(のリストに戻る)

AWS CDK および jsii の OpenPGP キー

このトピックには、 および jsii の現在 AWS CDK および過去の OpenPGP キーが含まれています。

現在のキー

これらのキーは、 AWS CDK および jsii の現在のリリースを検証するために使用される必要があります。

AWS CDK OpenPGP キー

キー ID:	0x42B9CF2286CD987A
タイプ:	RSA
サイズ：	4096/4096
作成済み：	2022-07-05
有効期限：	2026-07-04
ユーザー ID:	AWS クラウド開発キット <aws-cdk@amazon.com>
キーフィンガープリント：	69B5 2D5B A295 1D11 FA65 413B 42B9 CF22 86CD 987A

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGLEgOsBEADCoAMwvnszMLybJ+AD9cHhVyX6+rYIUEXYSgVnfkl6Z7qawIwv
wgd/a5fEs9Kiz2XJmfwS9Rxb4d+0+Y1ls1A+gnpw9FMLcZlqkC9KLnS2MqvuXWLB
t3z4kjZaL9fQ+58PoD4gy/M2hDg6gZrYqR3gtJuw8FcFpb/1KlkzRQUM8eAMFxf2
TyfjP0V0tSHwcB+84oushX7fUXVMyc3+OHsCPOe/WBFMIlWgKA+n33JKIQlUUC8f
kCWBAsAFupil0lCveT6mZu5slNRlc1I3iBLjUZ3/MtLygfqAMKwUVXeawtDvRIZe
PrAFc2NyODEhly2JG6K0FW7eIcvBqR3rg8U49t9Y74ELTM0kKnfd+flvq35xWqQC
0zghnk3kDppRTN4zWBgTKiCMxBcsHXGOoGn57t4B9VY9Zy3vkeySigeiwl/Tw9nJ
```

```
PE0SRnwEc/HnjTTfX+GTG1aQVE0xSVyZ4m5ymRNCu6+rNH8lKwo5FujlXJ+GXPkp
qT+Lx6Ix/Ny7PaoweWxwtZUkLRS4pWUsg0yotZrGyIbS+X3yMEG8WBTFI9hf6HTq
0ryfi5/TsBrdrGKqWB99EC9xYEGgtHp4fKO5X0ynOagVOhf0jSe8t1uyuJPGb2Gc
MQagSys5xMhdG/ZnEY4Cb+JDtH/4jc3tca0+4Z5RQ7kF9IhCncFtrbjJbwARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJixIDrAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
F4AACgkQQrnPIobNmHo2qg//Zt9p/kN1DevflzxWKouUX0AS7UmUtRYXu5k/EEbu
wkYNHpUr7+lZ+Me5YyjcIpt6UwuG9cW4SvwuxIfXucyKAWiwEbydCQauvnrYDxDa
J6Yr/ntk7Sii6An9re99qic3IsvX+xlUXh+qJ/34ooP/1PHziCMqykvW/DwAIyhx
2qvTXy+9+OlOWSUbhkCnNz5XKb4XQGq73DqalZX1nH4dG6fckZmYRX+dpw2njfTw
ZLdZ7bkrfiL84FI4A21RfSbEU4s4ngiV17lZ9ivilBKTbDv3da7+yc919M7C5N4J
yrlxvtyYNDoqKAD2WYZAnpEbG/shu3f56RyOJd56tXGwl9nKPh+F9y+379XthSwA
xZTURFtjWf7wWHaDZadU0DKi+Oeeszjg2f/VJaGmmS8PIg7q6GiSHHpqHqNvACHm
ZXMw12QFd3qt3xu0JMmE11ZC5VBgblwpkQTrO04Sq1rOpJwXI9ODMS/ZEhAIoYmT
OR7ouknlAx6mj9fwpavWDAAJHLdVUMYBZTXiQYFzDvx51ivvTRWkB1zTJcFdqShY
B37+Jz2jLDNdMrcHk2yfVp/VvfbxKcexg8wEwrrtQUslTUenl5jBZJouoz/wW81s
Y4U1nCPCdTK5/C7JCKzR2gVnCpe6uaxAWkkM2feQhjqJZkTC4cFVgBT+4M6WcT1r
yq4=
=ahbs
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP キー

キー ID:	0x056C4E15DAE3D8D9
タイプ:	RSA
サイズ：	4096/4096
作成済み：	2022-07-05
有効期限：	2026-07-04
ユーザー ID:	AWS JSII チーム <aws-jsii@amazon.com>
キーフィンガープリント：	1E07 31D4 57E5 FE87 87E5 530A 056C 4E15 DAE3 D8D9

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGLEgOkBEAD27EPVG9g2mHQ3+M6tF6le+tfhARJ2EV7m7NKIrTdSlCZATLWn
AVLlxG1unW34NlkKZbcbR86gAxRnnAhuEhPuloU/S5wAqPGbRiFl58YjYZDNJw6U
1SSMpE4O1sfjxv9yAbiRihLYtvksyHHZmaDhYner2aK1PdeWu+BKq/tjfm3Yzsd2
uuVEduJ72YoQk/29dEiGOHfT+2kUKxUX+0tJSJ9MGlEf4NtQE4WLzrT6Xqb2SG4+
alIiIVxIEi0XKDn7n8ZLjFwfJwOYxVYLtEUkqFWM8e8vgoc9/nYc+vDXZVED2g3Z
FWrwSnDSXbQpnMa2cLhD4xLpDHUS3i2p7r3dkJQGLo/5JGOopLibrOAbYZ72izhu
H/TuPFogSz0mNFPglrWdnLF04UIjIq420+06V4WQZC9n55Zjcbki/OhnC3B9pAdU
tiy8zg070bWq45dPGf5STkPPn7G8A2zmKefyO51iLIi26ZzW78siB+FvcGRhdg25
39sHJ1cmrTeC+B+k4KeV5sQ/m3UucimrZnk1xdaiVp8mWzRqWb8bB6Rs8K9RMrMV
tFBOKOBAT2QxOQtRGAantVgm193E1T1cmNpD0FKAKkDdPs64rKBEwFiHxccXHbah
eMd1weVwn3AKFD6uAm8ZRMV+dyssfcQxqpo/kfT1XpA6cQeOmGDOcKBfdwARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
YsSA6QIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEAVsThXa
49jZjU4QANoyqOJUT4gRrXshE3N0mW5Ad4i8Ke09GA62HyvTtfbsA+2nkNVGJpXm
sFMzdaFO95Q65RkLS9vW4nhhjXBEc2XYNCt2AnARudA/41ykjDPwU112z9ZTB9he
y4ItIeNGpHvMWr51fihl0y2nkpODOBeiv44jscLbHyOmZfki1f5fuIu2U2IbUGK3
5FtYyeHcgRHnpYkzLuzK4PfayOywqQPJ7M9DWrHf+v5Cu4ZCZDOIKfzF+ew7MWwc
6KaoWHCYbFpX8jxFppbGsSFOQ8Sl2quoP0TLz9Wsq70Khi6C2P8JI6lm0HRLO+1M
jFbQxNOwAcN3k4HSwunAjXBlmT/6oc1RsdBdpXBaZ2AWseIXwSYZqNXp+5L179uZ
vSiD3DSSUqLJbdQRVOsJi3/87V5QU59byq2dToHveRjtSbVnK0TkTx9Zlgkcpjv M
BwHNqWhratV6af2Upjq2YQ0fdSB42f3pgopInxNJPMvlAb+cCfr0Pfwu7ge7UooQ
WHTxbpCvwtn/HNctMGpWscO02WsWgoYVjnVFay/XphE77pQ9rRUkhMe6VKXfxj/n
OCZJKrydluIIwR8vvONNqO+QwZ1xDEhO7MaSZlOm1AuUZIXFPgaWQkPZHKiiwFA/
QWnL/+shuRtMH2geTjkev198Jgb5HyXFm4SyYtZferQROyliEhik
=BuGv
-----END PGP PUBLIC KEY BLOCK-----
```

履歴キー

これらのキーは、2022-07-05 より前の AWS CDK および jsii のリリースを検証するために使用できます。

> **Important**
>
> 新しいキーは、以前のキーの有効期限が切れる前に作成されます。その結果、特定の時点で、複数のキーが有効になる可能性があります。キーは、アーティファクトが作成された日から署名するために使用されるため、キーの有効性が重複する最近発行されたキーを使用します。

AWS CDK OpenPGP キー (2022-04-07)

> **Note**
>
> このキーは、2022-07-05 以降の AWS CDK アーティファクトの署名には使用されませんでした。

キー ID:	0x015584281F44A3C3
タイプ:	RSA
サイズ：	4096/4096
作成済み：	2022-04-07
有効期限：	2026-04-06
ユーザー ID:	AWS クラウド開発キット <aws-cdk@amazon.com>
キーフィンガープリント：	EAE1 1A24 82B0 AA86 456E 6C67 0155 8428 1F44 A3C3

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGJPLgUBEADtlR5jQtxtBmROQvmWlPOViqqnJNhk0dULc3tXnq8NS/l6X81r
wHk+/CHG5kBunwvM0qaqLFRC6z9NnnNDxEHcTi47n+OAjWyDM6unxxWOPz8Dfaps
Uq/ZWa4by292ZeqRC9Ir2wdrizb69JbRjeshBwlJDAS/qtqCAqBRH/f7Zw7QSD6/
XTxyIy+KOVjZwFPFNHMRQ/NmgUc/Rfxsa0pUjk1YAj/AkvQlwwD8DEnASoBhO0DP
QonZxouLqIpgp4LsGo8TZdQv30ocIj0C9DuYUiUXWlCPlYPgDj6IWf3rgpMQ6nB9
wC9lx4t/L3Zg1HUD52y8aymndmbdHVn90mzlNg4XWyc58rioYrEk57YwbDnea/Kk
Hv4kVHZRfJ4/OFPyqs5ex1X3X6rb07VvA1tfLgPywO9XF2Xws8YWOWcEobaWTcnb
AzyVC6wKya8rEQzxkYJ6UkJlhDB6g6bZwIpsI2zlimG+kSBsyFvE2oRYMS0cXPqU
o+tX0+4TvxEyW3RrUQzQHIpqXrb0X1Q8Z2idPn5dwsipDEa4gsFXtrSXmbB/0Cee
eJVvKWQAsxol3+NE9L/yozq3cz5PWh0SSbmCLRcs78lMJ23MmzbMWV7BWC9DXdY+
TywY5IkDUPjGCKlD8VlrI3TgC222bH6qaua6LYCiTtRtvpDYuJNAlUjhawARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
```

```
iQI/BBMBAgApBQJiTy4FAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
F4AACgkQAVWEKB9Eo8NpbxAAiBF0kR/lVw3vuam60mk4l0iGMVsP8Xq6g/buzbEO
2MEB4Ftk04qOnoa+93S0ZiLR9PqxrwsGSp4ADDX3Vtc4uxwzUlKUi1ywEhQ1cwyL
YHQI3Hd75K1J81ozMEu6qJH+yF0TtTDZMeZHtH/XvuIYJW3Lx4o5ZFlsEegFPAgX
YCCpUS+k9qC6M8g2VjcltQJpyjGswsKm6FWaKHW+B9dfjdOHlImB9E2jaknJ8eoY
zb9zHgFANluMzpZ6rYVSiCuXiEgYmazQWCvlPcMOP7nX+1hq1z11LMqeSnfE09gX
H+OYho9cMEJkb1dzx1H9MRpylFIn9tL+2iCp4UPJjnqi6uawWyLZ2tp4G11haqQq
1yAh69u233I8GZKFUySzjHwH5qWGRgBTjrZ6FdcjSS2w/wMkVKuCPkWtdvo/TJrm
msCd1Reye8SEKYqrs0ujTwmlvWmUZmO06AdUjo1kWiBKeslTJrWEuG7Yk4pFOoA4
dsaq83gxpOJNVCh6M3y4DLNrvl7dhF95NwTWMROPj2otw7NIjF4/cdzve2+P7YNN
pVAtyCtTJdD3eZbQPVaL3T8cf1VGqt6++pnLGnWJ0+X3TyvfmTohdJvN3TE+tq7A
7cprDX/q9c56HaXdJzVpxEzuf/YC+JuYKeHwsX3QouDhyRg3PsigdZES/02Wr8so
l6U=
=MQI4
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP キー (2022-04-07)

> **Note**
>
> このキーは、2022-07-05 以降の jsii アーティファクトの署名には使用されませんでした。

キー ID:	0x985F5BC974B79356
タイプ:	RSA
サイズ：	4096/4096
作成済み：	2022-04-07
有効期限：	2026-04-06
ユーザー ID:	AWS JSII チーム <aws-jsii@amazon.com>
キーフィンガープリント：	35A7 1785 8FA6 282D C5AC CD95 985F 5BC9 74B7 9356

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGJPLewBEADHH4TXup/gOlHrKDZRbj8MvsMTdM6eDteA6/c32UYV/YsK9rDA
jN8Jv/xlfosOebcHrfnFpHF9VTkmjuOpN695XdwMrW/NvlEPISTGEJf21x6ZTQ2r
1xWFYzC3sl3FZmvj9XAXTmygdv+XM3TqsFgZeCaBkZVdiLbQf+FhYrovUlgotb5D
YiCQI3ofV5QTE+141jhO5Pkd3ZIoBG+P826LaT8NXhwS0o1XqVk39DCZNoFshNmR
WFZpkVCTHyv5ZhVey1NWXnD8opO375htGNV4AeSmSIH9YkURD1g5F+2t7RiosKFo
kJrfPmUjhHn8IFpReGc8qmMMZX0WaV3t+VAWfOHGGyrXDfQ4xz1VCot75C2+qypM
+qhwOAOOP0zA7CfI96ULZzSH/j8HuQk3O0DsUCybpMuKEazEMxP3tgGtRerwDaFG
jQvAlK8Rbq3v8buBI6YJuXTwSzJE8KLjleUiTFumE6WP4rsAvlP/5rBvubeMfa3n
NIMm5Rkl36Z+jt3e2Z2ZqWDPpBRta8m7QHccrZhkvqu3YC3Gl6kdnm4Vio3Xfpg2
qtWhIQutQ6DmItewV+weQHas3hl88RPJtSrfWWIIMkpbF7Y4vbX9xcnsYCLlp2Mz
tWbbnU+EWATNSsufml/Kdnu9iEEuLmeovE11I69nwjNOq9P+GJ3r/FUb2wARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
Yk8t7AIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEJhfW8l0
t5NWo64P/2y7gcMRylLLW/wbrCjton2O4+YRocwQxKm1cBml9FVDUR5967YczNuu
EwEOfH/Pu3UAlrBfKAfxPNhKchLwYiOBNh2Wk5UUxRcldNHTLb5jn5gxCeWNAsl/
Tc46qY+ObdBMdOf2Vu33UCOg83WLbg1bfBoA8Bm1cd0XObtLGucu606EBt1dBrKq
9UTcbJfuGivY2Xjy5r4kEiMHBoLKcFrSo2Mm7VtYlE4Mabjyj9+orqUio7qxOl60
aa7Psa6rMvs1Ip9IOrAdG7o5Y29tQpeINH0R1/u47BrlTEAgG63Dfy49w2h/1g0G
c9KPXVuN55OWRIu0hsiySDMk/2ERsF348TU3NURZltnCOxp6pHlbPJIxRVTNa9Cn
f8tbLB3y3HfA80516g+qwNYIYiqksDdV2bz+VbvmCWcO+FellDZli831gyMGa5JJ
rq7d0lEr6nqjcnKiVwItTQXyFYmKTAXweQtVC72g1sd3oZIyqa7T8pvhWpKXxoJV
WP+OPBhGg/JEVC9sguhuv53tzVwayrNwb54JxJsD2nemfhQm1Wyvb2bPTEaJ3mrv
mhPUvXZj/I9rgsEq3L/sm2Xjy09nra4o3oe3bhEL8nOj11wkIodi17VaGP0y+H3s
I5zB5UztS6dy+cH+J7DoRaxzVzq7qtH/ZY2quClt30wwqDHUX1ef
=+iYX
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK OpenPGP キー (2018-06-19)

キー ID:	0x0566A784E17F3870
タイプ:	RSA
サイズ：	4096/4096
作成済み：	2018-06-19
有効期限：	2022-06-18
ユーザー ID:	AWS CDK チーム <aws-cdk@amazon.com>

キーフィンガープリント：	E88B E3B6 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFsovE8BEADEFVCHeAVPvoQgsjVu9FPUczxy9P+2zGIT/MLI3/vPLiULQwRy
IN2oxyBNDtcDToNa/fTkW3Ev0NTP4V1h+uBoKDZD/p+dTmSDRfByECMI0sGZ3UsG
Ohhyl2Of44s0sL8gdLtDnqSRLf+ZrfT3gpgUnplW7VitkwLxr78jDpW4QD8p8dZ9
WNm3JgB55jyPgaJKqA1Ln4Vduni/1XkrG42nxrrU71uUdZPvPZ2ELLJa6n0/raG8
jq3le+xQh45gAIs6PGaAgy7jAsfbwkGTBHjjujITAY1DwvQH5iS31OaCM9n4JNpc
xGZeJAVYTLilznf2QtS/a50t+ZOmpq67Ssp2j6qYpiumm0Lo9q3K/R4/yF0FZ8SL
1TuNX0ecXEptiMVUfTiqrLsANg18EPtLZZOYW+ZkbcVytkDpiqj7bMwA7mI7zGCJ
1gjaTbcEmOmVdQYS1G6ZptwbTtvrgA6AfnZxX1HUxLRQ7tT/wvRtABfbQKAh85Ff
a3U9W4oC3c1MP5IyhNV1Wo8Zm0flZiZc0iZnojTtSG6UbcxNNL4Q8e08FWjhungj
yxSsIBnQO1Aeo1N4BbzlI+n9iaXVDUN7Kz1QEyS4PNpjvUyrUiQ+a9C5sRA7WP+x
IEOaBBGpoAXB3oLsdTNO6AcwcDd9+r2NlXlhWC4/uH2YHQUIegPqHmPWxwARAQAB
tCFBV1MgQ0RLIFRlYW0gPGF3cy1jZGtAYW1hem9uLmNvbT6JAj8EEwEIACkFAlso
vE8CGy8FCQeEzgAHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRAFZqeE4X84
cLGxD/0XHnhoR2xvz38GM8HQlwlZy9W1wVhQKmNDQUavw8Zx7+iRR3m7nq3xM7Qq
BDbcbKSg1lVLSBQ6H2V6vRpysOhkPSH1nN2dO8DtvSKIPcxK48+1x7lmO+ksSs/+
oo1UvOmTDaRzOitYh3kOGXHHXk/l11GtF2FGQzYssX5iM4PHcjBsK1unThs56IMh
OJeZezEYzBaskTu/ytRJ236bPP2kZIEXfzAvhmTytuXWUXEftxOxc6fIAcYiKTha
aofG7WyR+Fvb1j5gNLcbY552QMxa23NZd5cSZH7468WEW1SGJ3AdLA7k5xvsPPOC
2YvQFD+vUOZ1JJuu6B5rHkiEMhRTLklkvqXEShTxuXiCp7iTOo6TBCmrWAT4eQr7
htLmqlXrgKi8qPkWmRdXXG+MQBzI/UyZq2q8KC6cx2md1PhANmeefhiM7FZZfeNM
WLonWfh8gVCsNH5h8WJ9fxsQCADd3Xxx3NelS2zDYBPRoaqZEEBbgUP6LnWFprA2
EkSlc/RoDqZCpBGgcoy1FFWvV/ZLgNU6OTQlYH6oYOWiylSJnaTDyurrktsxJI6d
4gdsFb6tqwTGecuUPvvZaEuvhWExLxAebhu780FdAPXgVTX+YCLI2zf+dWQvkFQf
80RE7ayn7BsiaLzFBVux/zz/WgvudsZX18r8tDiVQBL51ORmqw==
=0wuQ
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP キー (2018-08-06)

キー ID:	0x1C7ACE4CB2A1B93A
タイプ:	RSA

サイズ：	4096/4096
作成済み：	2018-08-06
有効期限：	2022-08-05
ユーザー ID:	AWS JSII チーム <aws-jsii@amazon.com>
キーフィンガープリント：	85EF 6522 4CE2 1E8C 72DB 28EC 1C7A CE4C B2A1 B93A

「コピー」アイコンを選択して、次の OpenPGP キーをコピーします。

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFtoSs0BEAD6WweLD0B26h0F7Jo9iR6tVQ4PgQBK1Va5H/eP+A2Iqw79UyxZ
WNzHYhzQ5MjYYI1SgcPavXy5/LV1N8HJ7QzyKszybnLYpNTLPYArWE8ZM9ZmjvIR
p1GzwnVBGQfoOlxyeutE9T5ZkAn45dTS5jlno4unji4gHjnwXKf2nP1APU2CZfdK
8vDpLOgj9LeeGlerYNbx+7xtY/I+csFIQvK09FPLSNMJQLlkBhY0r6Rt9ZQG+653
tJn+AUjyM237w0UIX1IqyYc5IONXu8HklPGu0NYuX9AY/63Ak2Cyfj0w/PZlvueQ
noQNM3j0nkOEsTOEXCyaLQw9iBKpxvLnm5RjMSODDCkj8c9uu0LHr7J4EOtgt2S1
pem7Y/c/N+/Z+Ksg9fP8fVTfYwRPvdI1x2sCiRDfLoQSG9tdrN5VwPFi4sGV04sI
x7Al8Vf/OBjAGZrDaJgM/gVvb9SKAQUA6t3ofeP14gDrS0eYodEXZ+lamnxFglxF
Sn8NRC4JFNmkXSUaTNGUdFf//F0D69PRNT8CnFfmniGj0CphN5037PCA2LC/Buq2
3+K6mTPkCcCHYPC/SwItp/xIDAQsGuDc1i1SfDYXrjsK7uOuwC5jLA9X6wZ/jgXQ
4umRRJBAV1aW8b1+yfaYYCO2AfXXO6caObv8IvH7Pc4leC2DoqylD3KklQARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQgAKQUC
W2hKzQIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEBx6zkyy
obk6B34P/iNb5QjKyhT0glZiq1wK7tuDDRpR6fC/sp6Jd/GhaNjO4BzlDbUPSjW5
950VT+qwaHXbIma/QVP7EIRztfwWy7m8eOodjpiu7JyJprhwG9nocXiNsLADcMoH
BvabkDRWXWIWSurq2wbcFMlTVwxjHPIQs6kt2oojpzP985CDS/KTzyjow6/gfMim
DLdhSSbDUM34STEgew79L2sQzL7cvM/N59k+AGyEMHZDXHkEw/Bge5Ovz50YOnsp
lisH4BzPRIw7uWqPlkVPzJKwMuo2WvMjDfgbYLbyjfvs5mqDxT2GTwAx/rd2taU6
iSqP0QmLM54BtTVVdoVXZSmJyTmXAAGlITq8ECZ/coUW9K2pUSgVuWyu63lktFP6
MyCQYRmXPh9aSd4+ielteXM9Y39snlyLgEJBhMxioZXVO2oszwluPuhPoAp4ekwj
/umVsBf6As6PoAchg7Qzr+lRZGmV9YTJOgDn2Z7jf/7tOes0g/mdiXTQMSGtp/Fp
ggnifTBx3iXkrQhqHlwtam8XTHGHy3MvX17ZslNuB8Pjh+07hhCxv0VUVZPUHJqJ
ZsLa398LMteQ8UMxwJ3t06jwDWAd7mbr2tatIilLHtWWBFoCwBh1XLe/03ENCpDp
njZ7OsBsBK2nVVcN0H2v5ey0T1yE93o6r7xOwCwBiVp5skTCRUob
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK デベロッパーガイドの履歴

リリースの詳細については、 AWS CDK「リリース」を参照してください。 AWS CDK は約 1 週間に 1 回更新されます。メンテナンスバージョンは、重要な問題に対処するために、毎週のリリースの間にリリースされる可能性があります。各リリースには、一致する AWS CDK ツールキット (CDK CLI)、 AWS コンストラクトライブラリ、および API リファレンスが含まれています。通常、このガイドの更新は AWS CDK リリースと同期しません。

> **Note**
>
> 次の表は、ドキュメントの重要なマイルストーンを表しています。エラーを修正し、コンテンツを継続的に改善します。

変更	説明	日付
CDK 移行機能のドキュメントを追加する	`cdk migrate` コマンドを使用して、デプロイされた AWS リソース、デプロイされた AWS CloudFormation スタック、およびローカル AWS CloudFormation テンプレートを に移行します AWS CDK CLI AWS CDK。詳細については、「 への移行 AWS CDK」を参照してください。	2024 年 2 月 21 日
IAM ベストプラクティスの更新	IAM ベストプラクティスに沿ってガイドを更新しました。詳細については、「IAM のセキュリティのベストプラクティス」を参照してください。	2023 年 3 月 23 日
ドキュメント `cdk.json`	`cdk.json` 設定値のドキュメントを追加します。	2022 年 4 月 20 日

依存関係管理	を使用した依存関係の管理に関するトピックを追加しますAWS CDK。	2022 年 4 月 7 日
Java から二重括弧を削除する例	このアンチパターンは、`Map.of`全体で Java 9 に置き換えてください。	2022 年 3 月 9 日
AWS CDK v2 リリース	AWS CDK デベロッパーガイドのバージョン 2 がリリースされました。CDK v1 のドキュメント履歴。	2021 年 12 月 4 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。