TypeScript AWS CDK で を使用して IaC プロジェクトを作成するためのベストプラクティス

AWS 規範ガイダンス

AWS 規範ガイダンス: TypeScript AWS CDK で を使用して IaC プロジェクトを作成するためのベストプラクティス

Table of Contents

TypeScript AWS CDK で を使用して IaC プロジェクトを作成するためのベストプラクティス

Sandeep Gawande、Mason Cahill、Sandip Gangapadhyay、Siamak Heshmati、Rajneesh Tyagi (Amazon Web Services (AWS))

2025 年 10 月 (ドキュメント履歴)

このガイドでは、TypeScript で AWS Cloud Development Kit (AWS CDK) を使用し、大規模な Infrastructure as Code (IaC) プロジェクトをビルドしデプロイする際の推奨事項とベストプラクティスについて説明します。 AWS CDK は、コードでクラウドインフラストラクチャを定義し、 を通じてそのインフラストラクチャをプロビジョニングするためのフレームワークです AWS CloudFormation。明確に定義されたプロジェクト構造がない場合は、大規模なプロジェクトの AWS CDK コードベースの構築と管理が困難な場合があります。こうした課題に対処するために、大規模プロジェクトでアンチパターンを使用する組織もありますが、そうしたパターンはプロジェクトの進行を遅らせ、組織に悪影響を及ぼす他の問題を引き起こす可能性があります。例えば、アンチパターンは、開発者のオンボーディング、バグ修正、新機能の採用を複雑にし、遅らせる可能性があります。

このガイドでは、アンチパターンに代わる方法を示し、スケーラビリティ、テスト、セキュリティのベストプラクティスとの整合性を考慮してコードを整理する方法を説明します。このガイドを参考にして、IaC プロジェクトのコード品質を向上させ、ビジネスのアジリティを最大化することも可能です。このガイドは、アーキテクト、テクニカルリード、インフラストラクチャエンジニア、および大規模な AWS CDK プロジェクト用に適切に設計されたプロジェクトを構築しようとしているその他のロールを対象としています。

目的

- コストの削減 – を使用して AWS CDK 、組織のセキュリティ、コンプライアンス、ガバナンス要件を満たす独自の再利用可能なコンポーネントを設計できます。また、組織全体でコンポーネントを簡単に共有できるため、デフォルトでベストプラクティスに沿った新しいプロジェクトを迅速に立ち上げることができます。
- 市場投入までの時間を短縮 – で使い慣れた機能を活用して AWS CDK 、開発プロセスを高速化します。これにより、デプロイの再利用性が向上し、開発作業が軽減されます。

- 開発者の生産性の向上 – 開発者は使い慣れた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定義できます。これにより、デベロッパーは AWS リソースを表現して維持できます。これにより、開発者の効率とコラボレーションが向上します。

ベストプラクティス

このセクションでは、以下のベストプラクティスの概要を説明します。

- 大規模プロジェクトのコード編成
- 再利用可能なパターンの開発
- コンストラクトの作成または拡張
- TypeScript のベストプラクティスを順守
- セキュリティの脆弱性やフォーマットエラーのスキャン
- ドキュメントの作成と改善
- テスト駆動型の開発アプローチを採用
- コンストラクトにリリースとバージョン管理を使用する
- ライブラリのバージョン管理を強制する

大規模プロジェクトのコード編成

コード編成が重要な理由

大規模な AWS CDK プロジェクトでは、高品質で明確に定義された構造を持つことが重要です。プロジェクトが大きくなり、サポートの対象となる機能やコンストラクトの数が増えるにつれて、コードナビゲーションはより難しくなります。この難しさが生産性に影響を及ぼし、開発者のオンボーディングを遅らせることにもなりかねません。

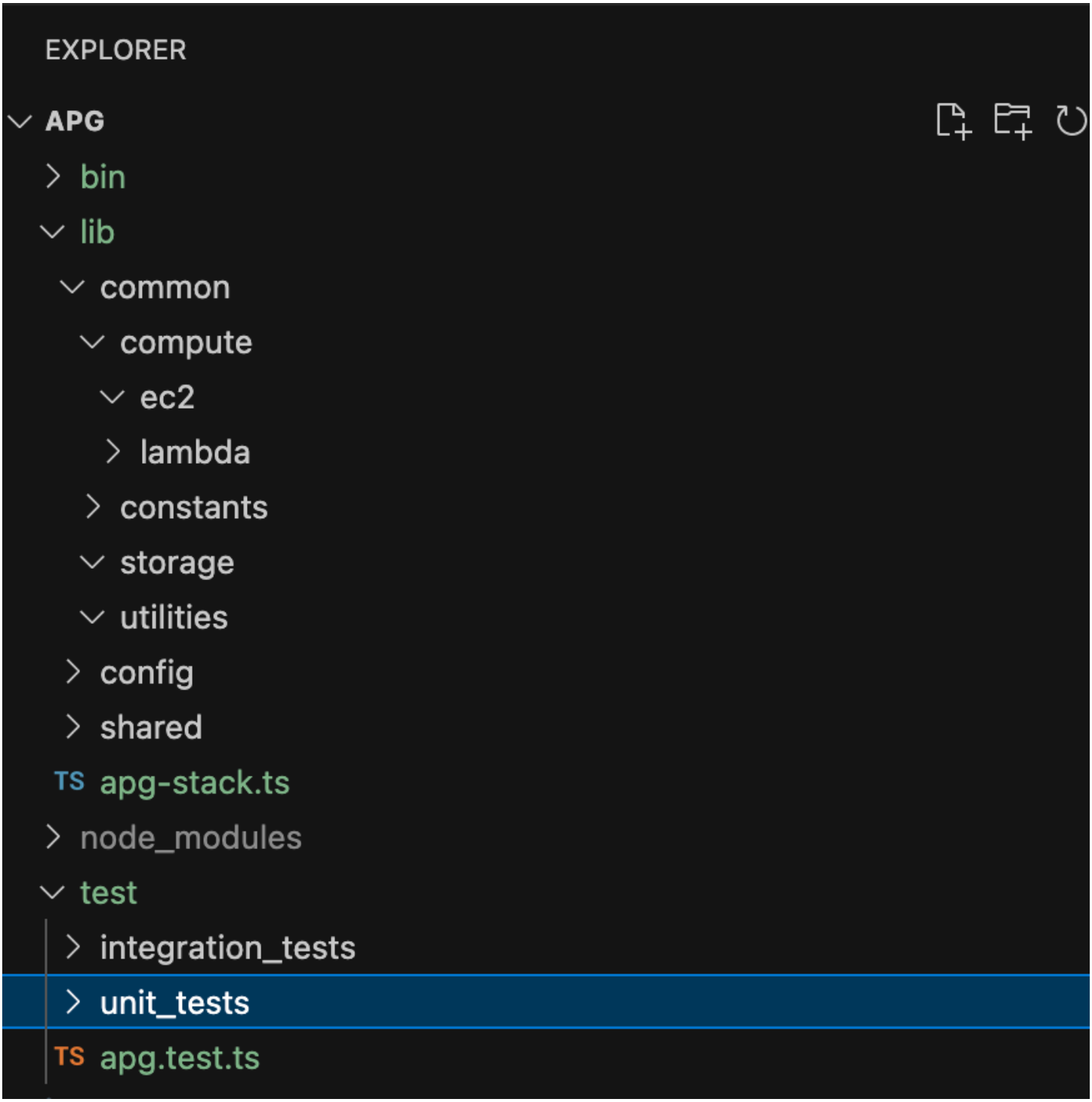
規模に見合ったコード編成の方法

高いレベルの柔軟性と可読性を持つコードを実現するには、コードを機能に基づいて論理的に分割することを推奨します。このような分割は、ほとんどのコンストラクトがさまざまなビジネスドメインで使用されているという事実を反映するものです。たとえば、フロントエンドアプリケーションとバックエンドアプリケーションの両方が AWS Lambda 関数を必要とし、同じソースコードを消費する可能性があります。ファクトリーでは、作成ロジックをクライアントに公開せずにオブジェクトを作成し、共通のインターフェイスを使用して新しく作成されたオブジェクトを参照できます。コードベースで整合性のある動作を作成するための効果的なパターンとして、ファクトリーを使用できます。さらに、ファクトリーは信頼できる単一のソースとしても機能し、コードの繰り返しを避け、トラブルシューティングを容易にします。

ファクトリーの仕組みをよりよく理解するために、自動車メーカーを例に考えてみましょう。自動車メーカーには、タイヤ製造に必要な知識とインフラストラクチャは必要ありません。自動車メーカーはその専門知識をタイヤの専門メーカーに外注し、必要に応じてそのメーカーにタイヤを注文すればよいのです。同じ原則がコードにも当てはまります。例えば、高品質の Lambda 関数を構築できる Lambda ファクトリーを作成しておけば、Lambda 関数を作成する必要があるときはいつでも、コード内の Lambda ファクトリーを呼び出すことができます。同様に、これと同じアウトソーシングプロセスを使用してアプリケーションを切り離し、モジュール型コンポーネントを構築できます。

サンプルコードの構成

次の図で示す TypeScript サンプルプロジェクトには、すべてのコンストラクトや共通機能を保存できる共通フォルダーがあります。



例えば、コンピューティングフォルダー (共通フォルダー内にあります) には、さまざまなコンピューティングコンストラクトのあらゆるロジックが格納されます。新任の開発者は、他のリソースに影響を与えることなく、新しいコンピューティングコンストラクトを簡単に追加できます。他のすべてのコンストラクトは、内部で新しいリソースを作成する必要はありません。これらのコンストラ

クトが共通コンストラクトファクトリーを呼び出すだけでよいのです。ストレージなど他のコンストラクトも同様に編成可能です。

設定には環境ベースのデータが含まれるため、ロジックを保存している共通フォルダーから切り離す必要があります。共有するフォルダー内には、共通の設定データを格納することを推奨します。また、ユーティリティフォルダーを使用して、すべてのヘルパー関数とクリーンアップスクリプトを提供することも推奨しています。

再利用可能なパターンの開発

ソフトウェアのデザインパターンは、ソフトウェア開発に共通する問題に対する再利用可能なソリューションです。これらは、ソフトウェアエンジニアがベストプラクティスに従った製品を開発するのに役立つガイドまたはパラダイムとして機能します。このセクションでは、 AWS CDK コードベースで使用できる Abstract Factory パターンと Chain of Responsibility パターンの 2 つの再利用可能なパターンの概要を説明します。各パターンをブループリントとして使用し、コード内にある特定の設計上の問題に合わせてカスタマイズできます。デザインパターンの詳細については、Refactoring.Guru ドキュメントの「デザインパターン」を参照してください。

Abstract Factory

Abstract Factory パターンは、具体的なクラスを指定せずに関連オブジェクトや依存関係オブジェクトのファミリーを作成するためのインターフェイスを提供します。このパターンは、次のようなユースケースに適用されます。

- クライアントが、システム内のオブジェクトの作成方法や構成方法に依存しない場合
- システムが複数のオブジェクトファミリーで構成されており、これらのファミリーを一緒に使用できるように設計されている場合
- 特定の依存関係を構築するためのランタイム値が必要な場合

Abstract Factory パターンの詳細については、Refactoring.Guru ドキュメントの「Abstract Factory を TypeScript で」を参照してください。

以下のコード例は、Amazon Elastic Block Store (Amazon EBS) ストレージファクトリーを構築するために Abstract Factory パターンを使用する方法を示しています。

```
abstract class EBSStorage {
    abstract initialize(): void;
}
```

```
class ProductEbs extends EBSStorage{
    constructor(value: String) {
        super();
        console.log(value);
    }
    initialize(): void {}
}

abstract class AbstractFactory {
    abstract createEbs(): EBSStorage
}

class EbsFactory extends AbstractFactory {
    createEbs(): ProductEbs{
        return new ProductEbs('EBS Created.')
    }
}

const ebs = new EbsFactory();
ebs.createEbs();
```

Chain of Responsibility

Chain of Responsibility とは、ハンドラー候補チェーンの 1 人がリクエストを処理するまで、ハンドラー候補チェーンに沿ってリクエストを渡せるようにする行動デザインパターンです。Chain of Responsibility パターンは、次のようなユースケースに適用されます。

- 実行時に決定された複数のオブジェクトがリクエストを処理する候補となる場合
- コードでハンドラーを明示的に指定しない場合
- レシーバーを明示的に指定せずに、複数のオブジェクトのうちの 1 つにリクエストを発行したい場合

Chain of Responsibility パターンの詳細については、Refactoring.Guru ドキュメントの「Chain of Responsibility を TypeScript で」を参照してください。

以下のコードは、Chain of Responsibility パターンを使用して、タスクの完了に必要な一連のアクションを構築する方法の例を示しています。

```
interface Handler {
    setNext(handler: Handler): Handler;
```

```
    handle(request: string): string;
}
abstract class AbstractHandler implements Handler
{
    private nextHandler: Handler;
    public setNext(handler: Handler): Handler {
        this.nextHandler = handler;
        return handler;
    }

    public handle(request: string): string {
        if (this.nextHandler) {
            return this.nextHandler.handle(request);
        }
        return '';
    }
}

class KMSHandler extends AbstractHandler {
    public handle(request: string): string {
        return super.handle(request);
    }
}
```

コンストラクトの作成または拡張

コンストラクトとは

コンストラクトは、 AWS CDK アプリケーションの基本的な構成要素です。コンストラクトは、Amazon Simple Storage Service (Amazon S3) バケットなどの単一の AWS リソースを表すことも、複数の AWS 関連リソースで構成される高レベルの抽象化にすることもできます。コンストラクトのコンポーネントには、関連するコンピューティング能力を持つワーカーキュー、またはモニタリングリソースとダッシュボードを持つスケジュールされたジョブが含まれることがあります。には、コンストラクトライブラリと呼ばれる AWS コンストラクトのコレクション AWS CDK が含まれています。ライブラリには、すべての のコンストラクトが含まれています AWS のサービス。Construct Hub を使用して、、サードパーティー AWS、オープンソース AWS CDK コミュニティから追加のコンストラクトを検出できます。

コンストラクトのさまざまなタイプ

には 3 つの異なるタイプのコンストラクトがあります AWS CDK。

- L1 コンストラクト — レイヤー 1 (L1) のコンストラクトは、まさに CloudFormation によって定義されたリソースであり、それ以上でもそれ以下でもありません。設定に必要なリソースは自分で用意しなければなりません。これらの L1 コンストラクトは非常に基本的なため、手動で設定する必要があります。 L1 コンストラクトには`Cfn`プレフィックスがあり、CloudFormation 仕様に直接対応しています。CloudFormation AWS のサービス がこれらのサービスをサポートする AWS CDK とすぐに、新しい が でサポートされます。CfnBucket は L1 コンストラクトの良い例です。このクラスは S3 バケットを表し、すべてのプロパティを明示的に設定する必要があります。L1 コンストラクトは、L2 または L3 コンストラクトが見つからない場合にのみ使用することをお勧めします。
- L2 コンストラクト— レイヤー 2 (L2) のコンストラクトには、共通の定型コードとグルーロジックがあります。これらのコンストラクトには便利なデフォルトがあり、それらについて知っておく必要がある知識の量を減らします。L2 コンストラクトは、インテントベースの APIs を使用してリソースを構築し、通常は対応する L1 モジュールをカプセル化します。L2 コンストラクトの良い例がバケットです。このクラスは、bucket.addLifeCycleRule() のようなデフォルトのプロパティとメソッドを使用して S3 バケットを作成し、ライフサイクルルールをバケットに追加します。
- L3 コンストラクト — レイヤー 3 (L3) のコンストラクトはパターンと呼ばれます。L3 コンストラクトは、 で一般的なタスクを完了するのに役立つように設計されており AWS、多くの場合、複数の種類のリソースが含まれます。これらは L2 コンストラクトよりもさらに限定された特定用途向けとなっていて、ある決まったユースケースに使用されます。たとえば、 aws-ecs-patterns.ApplicationLoadBalancedFargateService コンストラクトは、Application Load Balancer を使用するコンテナクラスターを含む AWS Fargate アーキテクチャを表します。もう 1 つの例として、aws-apigateway.LambdaRestApi コンストラクトがあります。このコンストラクトは Lambda 関数によってサポートされている Amazon API Gateway API を表しています。

コンストラクトレベルが高くなるにつれて、これらのコンストラクトがどのように使用されるかについて、より多くの仮定がなされます。これにより、極めて特殊なユースケースに対して、より効果的なデフォルトをインターフェイスに提供できるようになります。

独自のコンストラクトを作成する方法

独自のコンストラクトを定義するには、特定の方法に従う必要があります。これは、すべてのコンストラクトが `Construct` クラスを拡張するためです。`Construct` クラスはコンストラクトツリーの構成要素です。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。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。

- スコープ – コンストラクトの親または所有者。コンストラクトツリー内の場所を決定するスタックまたは別のコンストラクト。通常は `this` (または Python の `self`) にパスしなければならず、それがスコープの現在のオブジェクトを表します。
- id — このスコープ内で一意でなければならない識別子です。識別子は、現在の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、リソース名や CloudFormation 論理 IDs などの一意の ID を割り当てるために使用されます。
- Props – コンストラクトの初期設定を定義する一連のプロパティ。

次の例は、コンストラクトを定義する方法を示しています。

```
import { Construct } from 'constructs';

export interface CustomProps {
  // List all the properties
  Name: string;
}
export class MyConstruct extends Construct {
  constructor(scope: Construct, id: string, props: CustomProps) {
    super(scope, id);

    // TODO
  }
}
```

L2 コンストラクトの作成または拡張

L2 コンストラクトは「クラウドコンポーネント」を表し、CloudFormation がコンポーネントを作成するために必要なすべてのものをカプセル化します。L2 コンストラクトには 1 つ以上の AWS リソースを含めることができ、自分でコンストラクトをカスタマイズできます。L2 コンストラクトを作成または拡張することの利点は、コードを再定義しなくても CloudFormation スタックのコンポーネントを再利用できることです。コンストラクトをクラスとしてインポートするだけで済みます。

既存のコンストラクトとの「is a」関係がある場合は、既存のコンストラクトを拡張してデフォルトの機能を追加できます。既存の L2 コンストラクトのプロパティを再利用するのがベストプラクティスです。コンストラクターでプロパティを直接変更することで、プロパティを上書きできます。

次の例は、ベストプラクティスに沿って、`s3.Bucket` という既存の L2 コンストラクトを拡張する方法を示しています。この拡張は、`versioned`、`publicReadAccess`、`blockPublicAccess` の

ようなデフォルトプロパティを設定し、この新しいコンストラクトから作成されたすべてのオブジェクト (この例では S3 バケット) に、これらのデフォルト値が常に設定されるようにします。

```
import * as s3 from 'aws-cdk-lib/aws-s3';
import { Construct } from 'constructs';
export class MySecureBucket extends s3.Bucket {
  constructor(scope: Construct, id: string, props?: s3.BucketProps) {

    super(scope, id, {
      ...props,
      versioned: true,
      publicReadAccess: false,
      blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL
    });
  }
}
```

L3 コンストラクトの作成

コンポジションは、既存のコンストラクトコンポジションと「関係がある」場合に適しています。コンポジションとは、他の既存のコンストラクトに独自のコンストラクトを構築することです。独自のパターンを作成して、すべてのリソースとそのデフォルト値を、共有可能な単一上位レベルの L3 コンストラクトにカプセル化できます。独自の L3 コンストラクト (パターン) を作成する利点は、コードを再定義しなくてもコンポーネントをスタックで再利用できることです。コンストラクトをクラスとしてインポートするだけで済みます。これらのパターンは、消費者が共通のパターンに基づいて、限られた知識で複数のリソースを簡潔にプロビジョニングできるように設計されています。

次のコード例では、 という AWS CDK コンストラクトを作成しますExampleConstruct。このコンストラクトをテンプレートとして使用して、クラウドコンポーネントを定義できます。

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

export interface ExampleConstructProps {
  //insert properties you wish to expose
}

export class ExampleConstruct extends Construct {
  constructor(scope: Construct, id: string, props: ExampleConstructProps) {
    super(scope, id);
    //Insert the AWS components you wish to integrate
```

```
  }
}
```

次の例は、新しく作成されたコンストラクトを AWS CDK アプリケーションまたはスタックにインポートする方法を示しています。

```
import { ExampleConstruct } from './lib/construct-name';
```

次の例は、基本クラスから拡張したコンストラクトのインスタンスをインスタンス化する方法を示しています。

```
import { ExampleConstruct } from './lib/construct-name';

new ExampleConstruct(this, 'newConstruct', {
  //insert props which you exposed in the interface `ExampleConstructProps`
});
```

詳細については、AWS CDK ワークショップドキュメントの AWS CDK 「ワークショップ」を参照してください。

エスケープハッチ

でエスケープハッチを使用して抽象化レベル AWS CDK を上げると、より低いレベルのコンストラクトにアクセスできます。エスケープハッチは、 の最新バージョンでは公開されていない AWS が CloudFormation で利用可能な機能のコンストラクトを拡張するために使用されます。

次のようなシナリオでは、エスケープハッチの使用を推奨します。

- AWS のサービス 機能は CloudFormation を通じて使用できますが、`Construct`コンストラクトはありません。
- AWS のサービス 機能は CloudFormation を通じて利用でき、サービスの`Construct`コンストラクトがありますが、まだこの機能を公開していません。と言うのも、`Construct` コンストラクトの開発は「手作業で」行われるため、CloudFormation リソースコンストラクトよりも遅れる場合があるからです。

次のコード例は、エスケープハッチを使用する一般的なユースケースを示しています。この例では、上位レベルのコンストラクトにはまだ実装されていない機能を、オートスケーリング `LaunchConfiguration` 用の `httpPutResponseHopLimit` に追加するためのものです。

```
const launchConfig = autoscaling.onDemandASG.node.findChild("LaunchConfig") as
 CfnLaunchConfiguration;
            launchConfig.metadataOptions = {
                  httpPutResponseHopLimit: autoscalingConfig.httpPutResponseHopLimit||
 2
            }
```

前述のコード例は、次のワークフローを示しています。

1. L2 コンストラクトを使用して AutoScalingGroup を定義します。L2 コンストラクトは の更新をサポートしていないためhttpPutResponseHopLimit、エスケープハッチを使用する必要があります。
2. node.findChild() メソッドを使用して L2 AutoScalingGroupコンストラクトの特定のLaunchConfig子を見つけ、CfnLaunchConfigurationリソースとしてキャストします。
3. これで、L1 CfnLaunchConfiguration の launchConfig.metadataOptions のプロパティを設定できるようになりました。

カスタムリソース

カスタムリソースを使用すると、テンプレートにカスタムのプロビジョニングロジックを書き込むことができ、スタックを作成、更新 (カスタムリソースを変更した場合)、削除するたびに CloudFormation がそれを実行します。たとえば、 で使用できないリソースを含める場合は、カスタムリソースを使用できます AWS CDK。この方法により、すべての関連リソースを 1 つのスタックで管理できます。

カスタムリソースを構築するには、リソースの CREATE、UPDATE、DELETE ライフサイクルイベントに応答する Lambda 関数を書き込む必要があります。カスタムリソースが API を呼び出せるのが 1 回だけである場合には、AwsCustomResource コンストラクトの使用を検討してください。これにより、CloudFormation のデプロイ中に任意の SDK 呼び出しを実行することが可能となります。それ以外の場合は、必要な作業を実行するための独自の Lambda 関数を作成することを推奨します。

カスタムリソースの詳細については、CloudFormation ドキュメントの「カスタムリソース」を参照してください。カスタムリソースの使用例については、GitHub の「Custom Resource」リポジトリを参照してください。

次の例は、Lambda 関数を開始して CloudFormation に成功または失敗シグナルを送信する、カスタムリソースクラスの作成方法を示しています。

```
import cdk = require('aws-cdk-lib');
import customResources = require('aws-cdk-lib/custom-resources');
import lambda = require('aws-cdk-lib/aws-lambda');
import { Construct } from 'constructs';

import fs = require('fs');

export interface MyCustomResourceProps {
  /**
   * Message to echo
   */
  message: string;
}

export class MyCustomResource extends Construct {
  public readonly response: string;

  constructor(scope: Construct, id: string, props: MyCustomResourceProps) {
    super(scope, id);

    const fn = new lambda.SingletonFunction(this, 'Singleton', {
      uuid: 'f7d4f730-4ee1-11e8-9c2d-fa7ae01bbebc',
      code: new lambda.InlineCode(fs.readFileSync('custom-resource-handler.py',
 { encoding: 'utf-8' })),
      handler: 'index.main',
      timeout: cdk.Duration.seconds(300),
      runtime: lambda.Runtime.PYTHON_3_6,
    });

    const provider = new customResources.Provider(this, 'Provider', {
      onEventHandler: fn,
    });

    const resource = new cdk.CustomResource(this, 'Resource', {
      serviceToken: provider.serviceToken,
      properties: props,
    });

    this.response = resource.getAtt('Response').toString();
    }
}
```

次の例は、カスタムリソースの主なロジックです。

```
def main(event, context):
    import logging as log
    import cfnresponse
    log.getLogger().setLevel(log.INFO)

    # This needs to change if there are to be multiple resources in the same stack
    physical_id = 'TheOnlyCustomResource'

    try:
        log.info('Input event: %s', event)

        # Check if this is a Create and we're failing Creates
        if event['RequestType'] == 'Create' and
 event['ResourceProperties'].get('FailCreate', False):
            raise RuntimeError('Create failure requested')

        # Do the thing
        message = event['ResourceProperties']['Message']
        attributes = {
            'Response': 'You said "%s"' % message
        }

        cfnresponse.send(event, context, cfnresponse.SUCCESS, attributes, physical_id)
    except Exception as e:
        log.exception(e)
        # cfnresponse's error message is always "see CloudWatch"
        cfnresponse.send(event, context, cfnresponse.FAILED, {}, physical_id)
```

次の例は、 AWS CDK スタックがカスタムリソースを呼び出す方法を示しています。

```
import cdk = require('aws-cdk-lib');
import { MyCustomResource } from './my-custom-resource';

/**
 * A stack that sets up MyCustomResource and shows how to get an attribute from it
 */
class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const resource = new MyCustomResource(this, 'DemoResource', {
      message: 'CustomResource says hello',
    });
```

```
    // Publish the custom resource output
    new cdk.CfnOutput(this, 'ResponseMessage', {
      description: 'The message that came back from the Custom Resource',
      value: resource.response
    });
  }
}

const app = new cdk.App();
new MyStack(app, 'CustomResourceDemoStack');
app.synth();
```

TypeScript のベストプラクティスを順守

TypeScript は JavaScript の機能を拡張する言語です。これは厳密に型指定されたオブジェクト指向の言語です。TypeScript を使用することでコード内で渡されるデータのタイプを指定でき、タイプが一致しない場合はエラーを報告できます。このセクションでは、TypeScript のベストプラクティスの概要を説明します。

データの説明

TypeScript を使用して、コード内のオブジェクトと関数の形状を説明できます。any タイプを使用することは、変数の型検査をオプトアウトすることと同じです。コードに any を使用しないことを推奨します。以下はその例です。

```
type Result = "success" | "failure"
function verifyResult(result: Result) {
    if (result === "success") {
        console.log("Passed");
    } else {
        console.log("Failed")
    }
}
```

列挙型を使用する

列挙型を使用して名前付き定数のセットを定義し、さらにコードベースで再利用できる標準を定義できます。列挙型をグローバルレベルで一度エクスポートし、他のクラスにその列挙型をインポートして使用することを推奨します。コードベース内のイベントをキャプチャするための、一連のアクショ

ンを作成すると仮定します。TypeScript には、数値ベースと文字列ベース、両方の列挙型が用意されています。次の例では列挙型を使用します。

```
enum EventType {
    Create,
    Delete,
    Update
}

class InfraEvent {
    constructor(event: EventType) {
        if (event === EventType.Create) {
            // Call for other function
            console.log(`Event Captured :${event}`);
        }
    }
}

let eventSource: EventType = EventType.Create;
const eventExample = new InfraEvent(eventSource)
```

インターフェイスを使用する

インターフェイスはクラスに関する契約です。契約を作成する場合、ユーザーはその契約を順守する必要があります。次の例では、インターフェイスを使用して props を標準化し、このクラスを使用する際には、予想されるパラメータを呼び出し元が確実に提供できるようにしています。

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

interface BucketProps {
    name: string;
    region: string;
    encryption: boolean;

}

class S3Bucket extends Stack {
    constructor(scope: Construct, props: BucketProps) {
        super(scope);
        console.log(props.name);
```

```
    }
}
const app = App();
const myS3Bucket = new S3Bucket(app, {
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false
})
```

プロパティの中には、オブジェクトが最初に作成されたときにしか変更できないものもあります。これを指定するには、次の例で示しているように、プロパティ名の前に readonly を入力します。

```
interface Position {
    readonly latitude: number;
    readonly longitute: number;
}
```

インターフェイスを拡張する

インターフェイスを拡張すると、インターフェイス間でプロパティをコピーする必要がなくなるため、重複が減ります。また、コードの読者がアプリケーション内の関係を容易に理解できるようになります。

```
 interface BaseInterface{
    name: string;
  }
  interface EncryptedVolume extends BaseInterface{
      keyName: string;
  }
  interface UnencryptedVolume extends BaseInterface {
      tags: string[];
  }
```

空のインターフェイスを回避する

空のインターフェイスはリスクを生じる可能性があるため、使用しないことをお勧めします。次の例では、 という空のインターフェイスがありますBucketProps。myS3Bucket1 と myS3Bucket2 のオブジェクトはどちらも有効ですが、インターフェイスは契約を強制しないため、異なる標準に従っています。次のコードはプロパティをコンパイルして出力しますが、これによりアプリケーションに矛盾が生じます。

```
interface BucketProps {}

class S3Bucket implements BucketProps {
    constructor(props: BucketProps){
        console.log(props);
    }
}

const myS3Bucket1 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false,
});

const myS3Bucket2 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
});
```

ファクトリーを使用する

Abstract Factory パターンでは、インターフェイスはクラスを明示的に指定しなくても、関連するオブジェクトのファクトリーを作成します。例えば、Lambda 関数を作成するための Lambda ファクトリーを作成できます。コンストラクト内に新しい Lambda 関数を作成する代わりに、作成プロセスをファクトリに委任します。このデザインパターンの詳細については、Refactoring.Guru ドキュメントの「Abstract Factory を TypeScript で」を参照してください。

プロパティにデストラクチャリングを使用する

ECMAScript 6 (ES6) で導入されたデストラクチャリングは、配列やオブジェクトから複数のデータを抽出し、それらを独自の変数に割り当てることができる JavaScript の機能です。

```
const object = {
    objname: "obj",
    scope: "this",
};

const oName = object.objname;
const oScop = object.scope;

const { objname, scope } = object;
```

標準の命名規則を定義する

命名規則を適用することでコードベースの整合性が保たれ、変数の命名方法を考える際のオーバーヘッドが軽減されます。次の構成を推奨します。

- 変数名と関数名には camelCase を使用します。
- グローバル定数に UPPER_CASE を使用して、変更不可能なコンパイル時間値を明確に指定します。
- クラス名とインターフェイス名には PascalCase を使用します。
- インターフェイスメンバーには camelCase を使用します。
- 型名と列挙型名には PascalCase を使用します。
- camelCase を使用してファイルに名前 (例えば、`ebsVolumes.tsx` または `storage.ts`) を付けます

以下に、これらの推奨命名規則の例を示します。

```
// Variables and functions
const userName = 'john';
function getUserData() { }

// Global constants
const MAX_RETRY_ATTEMPTS = 3;
const API_BASE_URL = 'https://api.example.com';

// Classes and interfaces
class DatabaseConnection { }
interface UserProfile { }

// Types and enums
type ResponseStatus = 'success' | 'error';
enum HttpStatusCode { }
```

var キーワードを使用しない

`let` のステートメントは、TypeScript でローカル変数を宣言するために使用されます。`var` キーワードに似ていますが、`var`キーワードと比較してスコープに制限があります。`let` のあるブロック内で宣言された変数は、そのブロック内でのみ使用できます。`var` キーワードはブロックスコープにすることはできません。つまり、特定のブロック (で表される{}) の外部からアクセスできま

すが、定義されている関数の外部からはアクセスできません。var 変数を再宣言および更新できます。var キーワードを使用しないことがベストプラクティスです。

ESLint と Prettier の使用を検討する

ESLint はコードを静的に分析して問題をすばやく見つけます。ESLint を使用して、コードの見た目や動作を定義する一連のアサーション (lint ルールと呼ばれるもの) を作成できます。ESLint には、コードの改善に役立つ自動修正候補もあります。最後に、ESLint を使って共有プラグインから lint ルールを読み込むことができます。

Prettier は、さまざまなプログラミング言語をサポートする有名なコードフォーマッタです。Prettier を使用してコードスタイルを設定できるため、コードを手動でフォーマットしないで済みます。インストール後、package.json ファイルを更新し npm run format と npm run lint のコマンドを実行できます。

次の例は、 AWS CDK プロジェクトの ESLint と Prettier フォーマッターを有効にする方法を示しています。

```
"scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk",
    "lint": "eslint --ext .js,.ts .",
    "format": "prettier --ignore-path .gitignore --write '**/*.+(js|ts|json)'"
}
```

アクセス修飾子を使用する

TypeScript のプライベート修飾子は、可視性を同じクラスのみに制限します。プライベート修飾子をプロパティまたはメソッドに追加すると、同じクラス内でそのプロパティまたはメソッドにアクセスできます。

パブリック修飾子を使用すると、クラスのプロパティとメソッドにあらゆる場所からアクセスできます。プロパティとメソッドにアクセス修飾子を指定しない場合、デフォルトでパブリック修飾子が使用されます。

保護された修飾子を使うと、同一クラス内およびサブクラス内でクラスのプロパティとメソッドにアクセスできるようになります。 AWS CDK アプリケーションでサブクラスを作成する場合は、保護された修飾子を使用します。

ユーティリティタイプを使用する

TypeScript のユーティリティタイプは、既存のタイプに対して変換とオペレーションを実行する事前定義されたタイプ関数です。これにより、既存のタイプに基づいて新しいタイプを作成できます。たとえば、プロパティを変更または抽出したり、プロパティをオプションまたは必須にしたり、タイプのイミュータブルバージョンを作成したりできます。ユーティリティタイプを使用すると、より正確なタイプを定義し、コンパイル時に潜在的なエラーをキャッチできます。

部分 < タイプ>

Partial は、入力タイプのすべてのメンバーをオプションTypeとしてマークします。このユーティリティは、特定のタイプのすべてのサブセットを表すタイプを返します。Partial の例を次に示します。

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight: number;
}

let partialDog: Partial<Dog> = {};
```

必須<Type>

Required は の逆を実行しますPartial。これにより、入力タイプのすべてのメンバーがオプションTypeでなくなります (つまり、必須) 。Required の例を次に示します。

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight?: number;
}

let dog: Required<Dog> = {
  name: "scruffy",
  age: 5,
  breed: "labrador",
  weight: 55 // "Required" forces weight to be defined
```

```
};
```

セキュリティの脆弱性やフォーマットエラーのスキャン

Infrastructure as Code (IaC) と自動化は、企業にとって不可欠なものとなっています。IaC は非常に堅牢であるため、セキュリティリスクを管理する責任は大きくなります。IaC の一般的なセキュリティリスクには次のようなものがあります。

- 過剰許可 AWS Identity and Access Management (IAM) 権限
- オープンなセキュリティグループ
- 暗号化されていないリソース
- オンになっていないアクセスログ

セキュリティのアプローチとツール

以下のセキュリティアプローチの実装をお勧めします。

- 開発中の脆弱性検出 — ソフトウェアパッチの開発と配布は複雑なため、本番稼働環境での脆弱性の修復には費用と時間がかかります。さらに、本番稼働環境の脆弱性には悪用されるリスクもあります。本番稼働環境にリリースする前に脆弱性を検出して修正できるように、IaC リソースでコードスキャンを使用することを推奨します。
- コンプライアンスと自動修復 — AWS はマネージドルール AWS Config を提供します。これらのルールは、コンプライアンスを適用し、AWS Systems Manager 自動化を使用して自動修復を試行するのに役立ちます。 AWS Config ルールを使用してカスタムオートメーションドキュメントを作成して関連付けることもできます。

一般的な開発ツール

このセクションで説明するツールは、独自のカスタムルールを持つ組み込み機能の拡張に役立ちます。カスタムルールを組織の標準に合わせることをお勧めします。以下は一般的な開発ツールの一部です。

- cdk-nag を使用して、特定のスコープ内のコンストラクトが定義されたルールのセットに準拠していることを検証します。cdk-nag はルール抑制やコンプライアンスレポートにも使用できます。cdk-nag ツールは、 の側面を拡張してコンストラクトを検証します AWS CDK。詳細につい

ては、 AWS DevOps ブログの「Manage application security and compliance with the AWS Cloud Development Kit (AWS CDK) 」および「cdk-nag」を参照してください。

- オープンソースツールの Checkov を使用して、IaC 環境で静的分析を実行します。Checkov は、Kubernetes、Terraform、または CloudFormation のインフラストラクチャコードをスキャンすることで、クラウドの構成ミスを特定できます。Checkov を使用すると、JSON、JUnit XML、CLI など、さまざまな形式の出力を取得できます。Checkov は、動的なコード依存関係を示すグラフを作成することで、変数を効果的に処理できます。詳細については、Bridgecrew の GitHub「Checkov」リポジトリを参照してください。
- TFLint を使用すると、エラーや廃止された構文がチェックされ、ベストプラクティスを実施しやすくなります。TFLint はプロバイダー固有の問題を検証できない場合がありますのでご注意ください。TFLint の詳細については、Terraform Linters の GitHub「TFLint」リポジトリを参照してください。
- Amazon Q Developer を使用してセキュリティスキャンを実行します。統合開発環境 (IDE) で使用すると、Amazon Q Developer は AI を活用したソフトウェア開発支援を提供します。コードに関するチャット、インラインコード補完の提供、新しいコードの生成、セキュリティの脆弱性のスキャン、コードのアップグレードと改善を行うことができます。

ドキュメントの作成と改善

プロジェクトの成功にはドキュメントが不可欠です。ドキュメントはコードの仕組みを説明するだけでなく、開発者がアプリケーションの特徴や機能をよりよく理解するのにも役立ちます。質の高いドキュメントを作成· 改善することで、ソフトウェア開発プロセスの強化や、高品質なソフトウェアの維持が可能となり、開発者間の知識移転にも役立ちます。

ドキュメントには、コード内のドキュメントと、コードに関するサポートドキュメントという 2 つのカテゴリがあります。コード内のドキュメントはコメント形式です。コードに関するサポートドキュメントには、README ファイルや外部ドキュメントがあります。コード自体は理解しやすいため、開発者がドキュメントをオーバーヘッドと考えることは珍しくありません。小規模なプロジェクトにはそうした考えが当てはまるかもしれませんが、複数のチームが関与する大規模なプロジェクトではドキュメントは不可欠です。

コードの作成者は、その機能をよく理解しているため、ドキュメントを作成するのがベストプラクティスです。開発者は、個別のサポートドキュメントの管理から生じる追加のオーバーヘッドに苦労することがあります。この課題を解決するために、開発者はコードにコメントを追加し、それらのコメントを自動的に抽出して、すべてのバージョンのコードとドキュメントを同期させることができます。

開発者がコードからコメントを抽出してドキュメントを生成する際に役立つ、さまざまなツールがあります。このガイドでは、 AWS CDK コンストラクトの推奨ツールとして TypeDoc に焦点を当てています。

AWS CDK コンストラクトにコードドキュメントが必要な理由

AWS CDK 共通コンストラクトは、組織内の複数のチームによって作成され、異なるチーム間で共有されて消費されます。優れたドキュメントがあれば、コンストラクトライブラリーの利用者は最小限の労力で容易にコンストラクトを統合し、インフラストラクチャを構築できます。すべてのドキュメントを同期させることは大変な作業です。TypeDoc ライブラリを使用して抽出されるコード内のドキュメントを管理することを推奨します。

コンストラクトライブラリでの TypeDoc AWS の使用

TypeDoc は TypeScript 用のドキュメントジェネレーターです。TypeDoc を使用して TypeScript のソースファイルを読み取り、ファイル内のコメントを解析し、コードのドキュメントを含む静的サイトを生成できます。

次のコードは、TypeDoc を AWS コンストラクトライブラリと統合し、次のパッケージを の `package.json` ファイルに追加する方法を示していますdevDependencies。

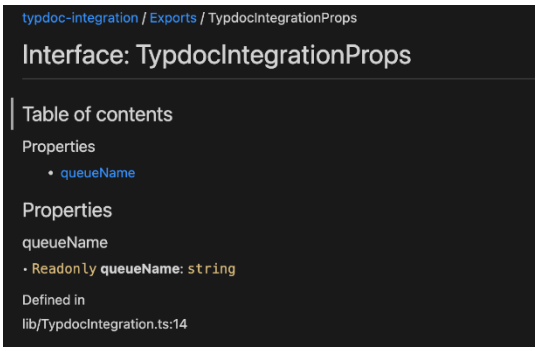
```
{

  "devDependencies": {

    "typedoc-plugin-markdown": "^3.11.7",
    "typescript": "~3.9.7"
  },

}
```

CDK ライブラリフォルダに `typedoc.json` を追加するには、次のコードを使用します。

```
{
    "$schema": "https://typedoc.org/schema.json",
    "entryPoints": ["./lib"],
}
```

README ファイルを生成するには、 AWS CDK コンストラクトライブラリプロジェクトのルートディレクトリで `npx typedoc` コマンドを実行します。

次のサンプルドキュメントは TypeDoc によって生成されます。



TypeDoc 統合オプションの詳細については、TypeDoc ドキュメントの「Doc Comments」を参照してください。

テスト駆動型の開発アプローチを採用

では、テスト駆動型開発 (TDD) アプローチに従うことをお勧めします AWS CDK。TDD は、コードを指定して検証するためのテストケースを開発するソフトウェア開発アプローチです。簡単に言うと、まず機能ごとにテストケースを作成し、テストが失敗したら、テストをパスする新しいコードを書き、コードをシンプルでバグのないものにします。

TDD を使って最初にテストケースを書くことができます。これにより、リソースにセキュリティポリシーを適用し、プロジェクト固有の命名規則に従うという点で、さまざまな設計上の制約があるインフラストラクチャを検証できます。 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は、 AWS CDK アサーション モジュールとTypeScript 用の Jest や Python 用の JavaScript や pytest などの一般的なテストフレームワークを使用することです。

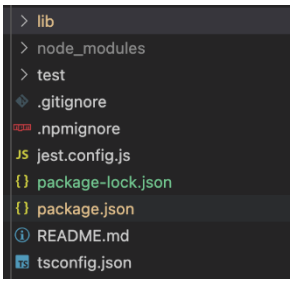
アプリケーション用に記述できるテストには 2 つのカテゴリがあります AWS CDK 。

- きめ細かいアサーションを使用して、例えば「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 CloudFormation テンプレートの特定の側面をテストします。これらのテストはリグレッションを検出できるだけでなく、TDD を使用して新機能を開発する場合にも役立ちます (最初にテストを書いてから、正しい実装を書いてパス合格させます)。きめ細かいアサーションは、最も多く作成するテストです。
- スナップショットテストを使用して、合成されたCloudFormation テンプレートを、以前に保存したベースラインテンプレートと照合してテストします。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るため、自由にリファクタリングできます。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、 AWS CDK アップグレードによって合成された

テンプレートが変更される可能性があるため、実装が正しいことを確認するためにスナップショットのみに依存することはできません。

ユニットテスト

このガイドでは、特に TypeScript のユニットテスト統合を中心に説明します。テストを有効にするには、 `package.json` `ts-jest`ファイルに `@types/jest`、 `jest`、 のライブラリがあることを確認しますdevDependencies。これらのパッケージを追加するには、`cdk init lib --language=typescript` のコマンドを実行します。上記のコマンドを実行すると、次の構造が表示されます。



次のコードは、Jest ライブラリで有効になっている`package.json`ファイルの例です。

```
{
  ...
  "scripts": {
    "build": "npm run lint && tsc",
    "watch": "tsc -w",
    "test": "jest",
  },
  "devDependencies": {
    ...
    "@types/jest": "27.5.2",
    "jest": "27.5.1",
    "ts-jest": "27.1.5",
    ...
  }
}
```

テストフォルダの下にテストケースを書き込めます。次の例は、 AWS CodePipeline コンストラクトのテストケースを示しています。

```
import { Stack } from 'aws-cdk-lib';
import { Template } from 'aws-cdk-lib/assertions';
```

```
import * as CodePipeline from 'aws-cdk-lib/aws-codepipeline';
import * as CodePipelineActions from 'aws-cdk-lib/aws-codepipeline-actions';
import { MyPipelineStack } from '../lib/my-pipeline-stack';
test('Pipeline Created with GitHub Source', () => {
  // ARRANGE
  const stack = new Stack();
  // ACT
  new MyPipelineStack(stack, 'MyTestStack');
  // ASSERT
  const template = Template.fromStack(stack);
  // Verify that the pipeline resource is created
  template.resourceCountIs('AWS::CodePipeline::Pipeline', 1);
  // Verify that the pipeline has the expected stages with GitHub source
  template.hasResourceProperties('AWS::CodePipeline::Pipeline', {
    Stages: [
      {
        Name: 'Source',
        Actions: [
          {
            Name: 'SourceAction',
            ActionTypeId: {
              Category: 'Source',
              Owner: 'ThirdParty',
              Provider: 'GitHub',
              Version: '1'
            },
            Configuration: {
              Owner: {
                'Fn::Join': [
                  '',
                  [
                    '{{resolve:secretsmanager:',
                    {
                      Ref: 'GitHubTokenSecret'
                    },
                    ':SecretString:owner}}'
                  ]
                ]
              },
              Repo: {
                'Fn::Join': [
                  '',
                  [
                    '{{resolve:secretsmanager:',
```

```
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:repo}}'
                ]
              ]
            },
            Branch: 'main',
            OAuthToken: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:token}}'
                ]
              ]
            }
          },
          OutputArtifacts: [
            {
              Name: 'SourceOutput'
            }
          ],
          RunOrder: 1
        }
      ]
    },
    {
      Name: 'Build',
      Actions: [
        {
          Name: 'BuildAction',
          ActionTypeId: {
            Category: 'Build',
            Owner: 'AWS',
            Provider: 'CodeBuild',
            Version: '1'
          },
          InputArtifacts: [
            {
              Name: 'SourceOutput'
```

```
              }
            ],
            OutputArtifacts: [
              {
                Name: 'BuildOutput'
              }
            ],
            RunOrder: 1
          }
        ]
      }
      // Add more stage checks as needed
    ]
  });
  // Verify that a GitHub token secret is created
  template.resourceCountIs('AWS::SecretsManager::Secret', 1);
});
);
```

テストを実行するには、プロジェクト内の `npm run test` コマンドを実行します。テストは次の結果を返します。

```
PASS  test/codepipeline-module.test.ts (5.972 s)
  # Code Pipeline Created (97 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        6.142 s, estimated 9 s
```

テストケースの詳細については、 AWS Cloud Development Kit (AWS CDK) デベロッパーガイドの「コンストラクトのテスト」を参照してください。

統合テスト

AWS CDK コンストラクトの統合テストは、 `integ-tests`モジュールを使用して含めることもできます。統合テストは AWS CDK アプリケーションとして定義する必要があります。統合テストと AWS CDK アプリケーションの間には one-to-one の関係が必要です。詳細については、 AWS CDK API リファレンスの integ-tests-alpha モジュールを参照してください。

コンストラクトにリリースとバージョン管理を使用する

のバージョン管理 AWS CDK

AWS CDK 共通コンストラクトは、複数のチームによって作成され、組織全体で共有して使用できます。通常、開発者は共通の AWS CDK コンストラクトで新機能やバグ修正をリリースします。これらのコンストラクトは、依存関係の一部としてアプリケーションまたは他の既存の AWS CDK コンストラクトによって AWS CDK 使用されます。このため、開発者はコンストラクトを適切なセマンティックバージョンで個別に更新しリリースすることが重要です。ダウンストリーム AWS CDK アプリケーションやその他の AWS CDK コンストラクトは、新しくリリースされた AWS CDK コンストラクトバージョンを使用するように依存関係を更新できます。

セマンティックバージョニング (Semver) とは、コンピュータソフトウェアに一意のソフトウェア番号を付与するための一連のルールまたはメソッドです。バージョンは次のように定義されます。

- メジャーバージョンは、互換性のない API の変更または重大な変更で構成されます。
- マイナーバージョンは、下位互換性のある方法で追加された機能で構成されます。
- パッチバージョンは、下位互換性のあるバグ修正で構成されます。

セマンティックバージョニングの詳細については、セマンティックバージョニングドキュメントの「Semantic Versioning Specification (SemVer)」を参照してください。

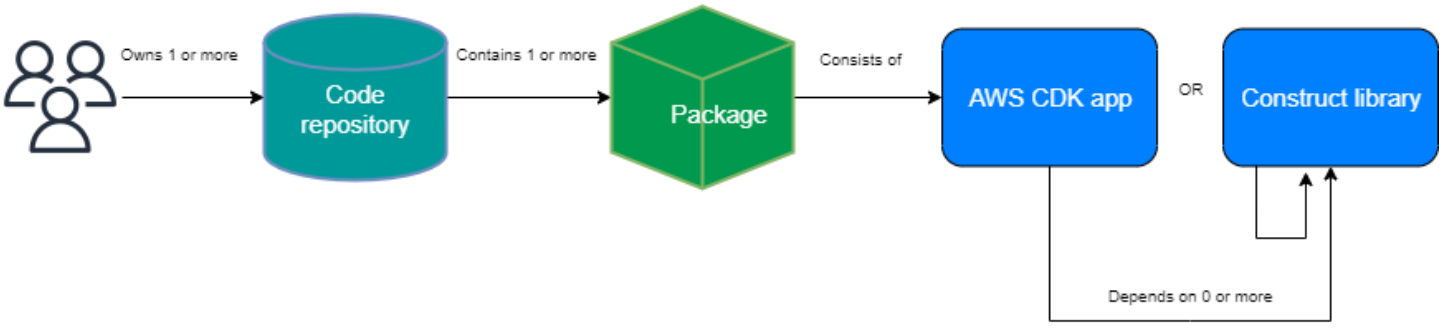
AWS CDK コンストラクトのリポジトリとパッケージ化

AWS CDK コンストラクトは異なるチームによって開発され、複数の AWS CDK アプリケーションで使用されるため、 AWS CDK コンストラクトごとに個別のリポジトリを使用できます。これはアクセスコントロールの強化にも役立ちます。各リポジトリには、同じ AWS CDK コンストラクトに関連するすべてのソースコードとそのすべての依存関係を含めることができます。1 つのアプリケーション (コンストラクト AWS CDK) を 1 つのリポジトリに保持することで、デプロイ中の変更の影響範囲を減らすことができます。

は、インフラストラクチャをデプロイするための CloudFormation テンプレートを生成する AWS CDK 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をバンドルし、インフラストラクチャと一緒にデプロイします。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結合できるだけでなく、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。

リポジトリの境界を越えてパッケージを使用するには、npm、PyPI、Maven Central に似たプライベートパッケージリポジトリを組織の内部に持つ必要があります。また、パッケージを構築し、テストし、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。ローカルの仮想マシン (VM) または Amazon S3 を使用して、PyPI サーバーなどのプライベートリポジトリを作成できます。プライベートパッケージレジストリを設計または作成するときは、高い可用性とスケーラビリティによってサービスが中断されるリスクを考慮することが重要です。パッケージを保存するためにクラウドでホストされているサーバーレスマネージドサービスは、メンテナンスのオーバーヘッドを大幅に削減できます。たとえば、 を使用して、ほとんどの一般的なプログラミング言語のパッケージAWS CodeArtifactをホストできます。さらに、CodeArtifact を使用して外部リポジトリ接続を設定し、それらを CodeArtifact 内に複製することもできます。

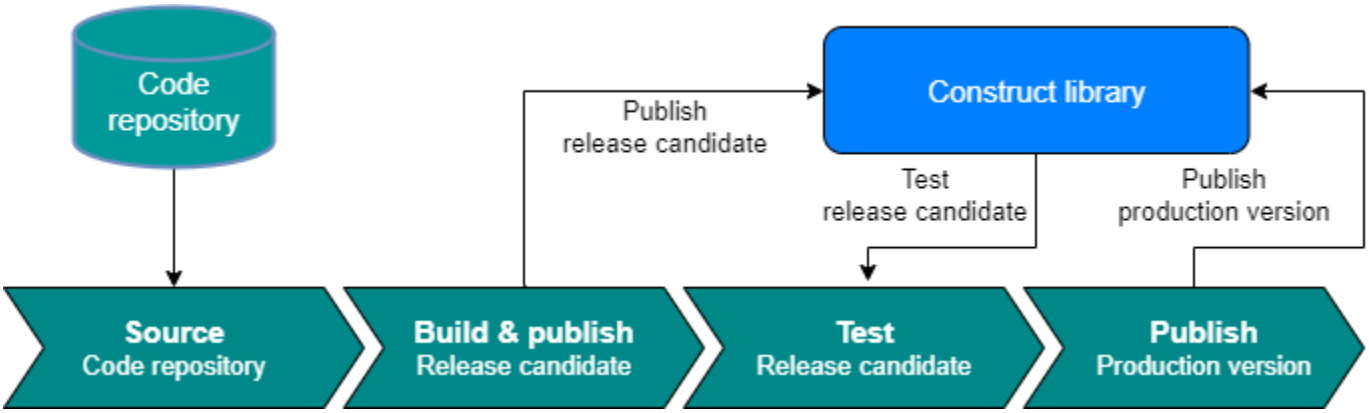
パッケージリポジトリ内のパッケージへの依存関係は、使用する言語のパッケージマネージャー (TypeScript や JavaScript アプリケーションの場合は npm など) によって管理されます。パッケージマネージャーは、アプリケーションが依存するすべてのパッケージの特定のバージョンを記録し、次の図のように制御された方法で依存関係をアップグレードできるようにすることで、ビルドが繰り返し可能であることを確認します。



のコンストラクトリリース AWS CDK

新しい AWS CDK コンストラクトバージョンを構築およびリリースするには、独自の自動パイプラインを作成することをお勧めします。プルリクエストの承認プロセスを適切に行い、ソースコードをコミットしてリポジトリのメインブランチにプッシュすれば、パイプラインはリリース候補バージョンを構築· 作成できるようになります。そのバージョンを CodeArtifact にプッシュして、本番稼働環境対応バージョンをリリースする前にテストできます。必要に応じて、コードをメインブランチとマージする前に、新しい AWS CDK コンストラクトバージョンをローカルでテストできます。これにより、パイプラインから本番稼働環境に対応したバージョンがリリースされます。共有コンストラクトやパッケージは、あたかも一般に公開されているかのように、利用側アプリケーションとは独立してテストする必要があることを考慮します。

次の図は、 AWS CDK バージョンリリースパイプラインの例を示しています。



以下のサンプルコマンドを使用すると npm パッケージのビルド、テスト、公開が可能です。まず、以下のコマンドを実行してアーティファクトリポジトリにサインインします。

```
aws codeartifact login --tool npm --domain <Domain Name> --domain-owner $(aws sts get-caller-identity --output text --query 'Account')  \
--repository <Repository Name> --region <AWS Region Name>
```

以下のステップを実行します。

1. package.json ファイル npm install に基づいて必要なパッケージをインストールする
2. リリース候補バージョン npm version prerelease --preid rc を作成する
3. npm パッケージ npm run build をビルドする
4. npm パッケージ npm run test をテストする
5. npm パッケージ npm publish を公開する

ライブラリのバージョン管理を強制する

ライフサイクル管理は、 AWS CDK コードベースを維持する場合の重要な課題です。たとえば、バージョン 1.97 で AWS CDK プロジェクトを開始し、後でバージョン 1.169 が利用可能になるとします。バージョン 1.169 には新機能とバグ修正が含まれていますが、古いバージョンを使用してインフラストラクチャをデプロイしています。現在、新しいバージョンでは重大な変更が加えられる可能性があるため、このギャップが拡大するにつれ、コンストラクトの更新が難しくなります。環境に多くのリソースがある場合、これは難しい場合があります。このセクションで紹介するパターンは、自動化を使用して AWS CDK ライブラリのバージョンを管理するのに役立ちます。このパターンのワークフローは次のとおりです。

1. 新しい CodeArtifact Service Catalog 製品を起動すると、 AWS CDK ライブラリのバージョンとその依存関係が `package.json` ファイルに保存されます。
2. すべてのリポジトリを追跡する共通のパイプラインをデプロイして、重大な変更がない場合はリポジトリに自動アップグレードを適用できるようにします。
3. AWS CodeBuild ステージは依存関係ツリーをチェックし、重大な変更を探します。
4. パイプラインは機能ブランチを作成し、エラーがないことを確認するために新しいバージョンで `cdk synth` を実行します。
5. 新しいバージョンをテスト環境にデプロイし、最後に統合テストを実行してデプロイが正常であることを確認します。
6. Amazon Simple Queue Service (Amazon SQS) キューを 2 つ使用してスタックを追跡できます。ユーザーは例外キューのスタックを手動で確認することで、重大な変更に対処できます。統合テストをパスしたアイテムは、マージおよびリリースが可能となります。

よくある質問

TypeScript はどのような問題を解決できますか?

通常、自動テストを作成し、コードが想定どおりに動作することを手動で検証し、最後に別の人にコードを検証してもらうことで、コードのバグを排除できます。プロジェクトのすべての部分間の接続を検証するには時間がかかります。検証プロセスをスピードアップするために、TypeScript のような型検査言語を使用してコード検証を自動化すれば、開発中に即座にフィードバックを提供できます。

なぜ TypeScript を使うべきなのでしょうか?

TypeScript は、JavaScript コードを簡素化し、読み取りとデバッグを容易にするオープンソース言語です。また、TypeScript は JavaScript IDE やプラクティス向けの生産性の高い開発ツール (静的チェックなど) も提供しています。さらに、TypeScript には ECMAScript 6 (ES6) の利点もあり、生産性を向上させることができます。最後に、TypeScript は、コードの型検査を行うことで、開発者が JavaScript を書くときによく遭遇する厄介なバグの回避に役立ちます。

AWS CDK または CloudFormation を使用する必要がありますか?

組織で を活用するための開発の専門知識がある場合は AWS CloudFormation、 AWS Cloud Development Kit (AWS CDK) 代わりに を使用することをお勧めします AWS CDK。これは、プログラミング言語と OOP の概念を使用できるため、 AWS CDK が CloudFormation よりも柔軟であるためです。CloudFormation を使用して、整った予測可能な方法で AWS リソースを作成できることに注意してください。CloudFormation では、リソースは JSON または YAML 形式を使用してテキストファイルに書き込まれます。

AWS CDK が新しく起動された をサポートしていない場合はどうなりますか AWS のサービス?

RAW 型の上書きまたは CloudFormation のカスタムリソースを使用できます。

でサポートされているさまざまなプログラミング言語は何ですか AWS CDK?

AWS CDK は、JavaScript、TypeScript、Python、Java、C#、Go (デベロッパープレビュー) で一般利用可能です。

AWS CDK コストはいくらですか?

には追加料金はかかりません AWS CDK。リソース AWS (Amazon EC2 インスタンスや Elastic Load Balancing ロードバランサーなど) は、手動で作成した場合と同様に AWS CDK 、 の使用時に作成される料金が発生します。使用した分だけをお支払いいただきます。最低料金や前払いの義務はありません。

次のステップ

TypeScript AWS Cloud Development Kit (AWS CDK) で を使用して構築を開始することをお勧めします。詳細については、AWS CDK Immersion Day Workshop を参照してください。

リソース

リファレンス

- AWS ソリューションコンストラクト (AWS ソリューション)
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS コンストラクトライブラリ API リファレンス (AWS CDK リファレンスドキュメント)
- AWS CDK リファレンスドキュメント (AWS CDK リファレンスドキュメント)
- AWS CDK Immersion Day Workshop (AWS Workshop Studio)

ツール

- cdk-nag(GitHub)
- TypeScript ESLint (TypeScript ESLint ドキュメント)

ガイドとパターン

- AWS ソリューション パターンの構築 (AWS ドキュメント)

ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、RSS フィード をサブスクライブできます。

変更	説明	日付
ベストプラクティスの更新	「共通開発ツール」セクションで cfn-nag を使用するための推奨事項を削除しました。「標準命名規則の定義」セクションで、グローバル定数の標準命名規則を追加し、命名例を追加しました。	2025 年 10 月 23 日
コードの更新	コードサンプルは、TypeScript のベストプラクティスに従う」セクションで更新しました。	2024 年 2 月 16 日
セクションを追加する	ユーティリティタイプの使用と統合テストのセクションを追加しました。	2024 年 1 月 10 日
マイナーな更新	L3 コンストラクトを作成するコード例を更新しました。	2023 年 6 月 16 日
初版発行	—	2022 年 12 月 8 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。