AWS CDK

はじめに

本ページは、AWS に関する個人の勉強および勉強会で使用することを目的に、AWS ドキュメントなどを参照し作成しておりますが、記載の誤り等が含まれる場合がございます。

最新の情報については、AWS 公式ドキュメントをご参照ください。

Contents

AWS CDK とは

AWS Cloud Development Kit のことで、使い慣れたプログラミング言語を使用してクラウドアプリケーションのリソースを定義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。

【AWS Black Belt Online Seminar】AWS Cloud Development Kit (CDK)(YouTube)(1:01:23)

 blackbelt-cdk_3

【AWS Black Belt Online Seminar】AWS CDK 概要 (Basic #1)(YouTube)(0:33:07)PDF

 blackbelt-cdk_1

【AWS Black Belt Online Seminar】AWS CDK の基本的なコンポーネントと機能 (Basic #2)(YouTube)(0:28:13)PDF


blackbelt-cdk_2

【AWS Black Belt Online Seminar】AWS CDK の開発を効率化する機能 (Basic #3)(YouTube)(0:29:20)PDF


blackbelt-cdk_3

AWS クラウド開発キット

AWS CDK Reference Documentation

AWS CDK DEVELOPER GUIDE

AWS CDK API REFERENCE

AWS Cloud Development Kit のよくある質問

CDK v2

CDK の v2 は、2021 年 5 月のプレビューが実施され、 2021 年 12 月 2 日に GA されました。

AWS Cloud Development Kit v2 開発者プレビューのお知らせ
https://aws.amazon.com/jp/blogs/news/announcing-aws-cloud-development-kit-v2-developer-preview/

AWS Cloud Development Kit (AWS CDK) v2 の一般提供開始 https://aws.amazon.com/jp/about-aws/whats-new/2021/12/aws-cloud-development-kit-cdk-generally-available/

v2 では Construct ライブラリが aws-cdk-lib に単一化されたため、v1 で実施していた個々のパッケージインストールが不要になりました。

v1 では個別にパッケージをインストールする必要があったため、後からインストールしたパッケージはバージョンを指定しないとバージョンが異なってしまうことがあります。バージョンを合わせるために、「npm install @aws-cdk/aws-lambda@1.111.0」とする必要がありました。

```
npm install @aws-cdk/aws-lambda
npm install @aws-cdk/aws-cloudfront
npm install @aws-cdk/aws-iam
npm install @aws-cdk/aws-s3


import * as cdk from "@aws-cdk/core";
import * as cloudfront from "@aws-cdk/aws-cloudfront";
import * as s3 from "@aws-cdk/aws-s3";
import * as iam from "@aws-cdk/aws-iam";
```

v2 では単一のパッケージに統合されたため、個別パッケージをインストールすることなく使用できます。

```
npm install aws-cdk-lib

import {
  aws_s3 as s3,
  aws_iam as iam,
```

```
  aws_cloudfront as cloudfront,
} from "aws-cdk-lib";
```

対応言語

- TypeScript
- JavaScript
- Python
- Java
- C#/.NET

CDK ワークショップ

手を動かして学ぶことができます。

https://cdkworkshop.com/

CDK の構成要素

 AppStacks.png

引用：開発者ガイド>CDK とは

App

CDK の最上位層で、複数のスタックの依存関係などを定義します。

Stack(s)

CloudFormation のスタックに対応します。AWS へのデプロイはこのスタック単位で行います。

Construct

Stack 層に AWS リソースの定義を作成します。

Construct には 3 つの種類があります。

- L1 Construct(Low Level Construct)
- L2 Construct(High Level Construct)
- L3 Construct(Patterns)

L1 Construct / L2 Construct

L1 Construct とは、Low Level Construct のことです。CloudFormation の各リソースと 1:1 の関係になっています。`Cfn`で始まるものが L1 です。

CloudFormation で定義するのと同じレベルでの記載になります。L2 Construct が存在するリソースは、L2 Construct を利用することを推奨しますが、要件に応じた細かい設定が必要な場合は、L1 Construct を利用します。

https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/constructs.html#constructs_l1_using

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

L2 Construct とは、High Level Construct と呼ばれるもので、L1 Construct をラップした Construct です。

https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/constructs.html#constructs_using

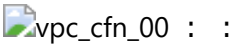
```
import * as s3 from "aws-cdk-lib/aws-s3";

const bucket = new s3.Bucket(this, "MyBucket", {
  versioned: true,
});
```

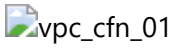
TypeScript を使用して、VPC を作る場合は下記のようにするだけで、VPC、ルートテーブル、インターネットゲートウェイ、NAT ゲートウェイ、サブネット 6 つ（3 種類 × maxAzs 数分） の CloudFormation 定義を作成してくれます。CloudFormation で記述すると 430 行にもなる定義が 12 行だけで完了します。

cidrMask の数字だけ指定しているので、サブネットの CIDR ブロックは自動的に生成されます。CIDR ブロックを特定の値で指定したい場合は、L1 Construct を使用します。

```
const vpc = new ec2.Vpc(this, "vpc", {
  vpcName: "vpc",
  cidr: "10.0.0.0/16",
  natGateways: 2,
  natGatewaySubnets: { subnetType: ec2.SubnetType.PUBLIC },
  maxAzs: 2,
  subnetConfiguration: [
    { subnetType: ec2.SubnetType.PUBLIC, name: "public", cidrMask: 20 },
    {
      subnetType: ec2.SubnetType.PRIVATE_WITH_NAT,
      name: "private",
      cidrMask: 19,
    },
    {
      subnetType: ec2.SubnetType.PRIVATE_ISOLATED,
      name: "protected",
      cidrMask: 21,
    },
  ],
});
```

この定義を実行した場合に生成される CloudFormation のコードは次の通りです。



： ： 

Patterns

L3 Construct として、ECS Patterns のように ECS 関連のリソースを簡単に生成できるものがあります。

次のようにするだけで、ECS サービス、ALB、関連するセキュリティグループ、タスク用の IAM ロールなど 200 ～ 300 行の CloudFormation のコードを出力してくれます。

```
// クラスタを作成
const cluster = new ecs.Cluster(this, "MyCluster", {
  vpc: props.vpc,
});

const loadBalancedFargateService =
  new ecs_patterns.ApplicationLoadBalancedFargateService(
    this,
    "MyFargateService",
    {
      cluster: cluster,
      cpu: 4096,
      memoryLimitMiB: 30720,
      publicLoadBalancer: true,
      desiredCount: 6,
      taskImageOptions: {
        image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"),
      },
    }
  );
```

CDK のテスト

Snapshot test

前回生成された CloudFormation テンプレートと比較して差分をチェックするスナップショットテストが実施できます。 スナップショットテストを行うことで、意図しないテンプレートへの変更があるかどうかを検知できます。

```
test('snapshot validation test',() =>{
    const stack = new MyTestStack(app, 'MyTestStack', {
        pjName: projectName,
        envName: envName,
        description: 'xxxxxxxx',
        :
        isAutoDeleteObject: false,
        env: defaultEnv,
        terminationProtection: false, // Enabling deletion protection
    });
    // add tag
    cdk.Tags.of(app).add('Project', projectName);
    cdk.Tags.of(app).add('Environment', envName);
    // test with snapshot
    expect(Template.fromStack(stack)).toMatchSnapshot();
```

```
})
```

Unit Test

Jest を使った Unit Test も実施できます。これにより、リソース単位の細かなテストを行うことができます。

VPC の場合は、以下のようにして VPC やサブネットの数、ルートテーブルの状態などをテストすることができます。

ただ、"AWS::EC2::VPC" のように AWS のリソースを知っていないといけないので慣れないうちは手間取るかもしれません。

```
test("create the vpc", () => {
  // GIVEN
  const app = new App({
    context: {
      PJName: "junit",
      EnvName: "prod",
      CidrBlock: "10.0.0.0/16",
    },
  });
  const stack = new VPCStack(app, "testing-vpcstack", {});
  // WHEN
  const template = Template.fromStack(stack);

  // THEN
  // VPC
  template.resourceCountIs("AWS::EC2::VPC", 1);
  template.hasResourceProperties("AWS::EC2::VPC", {
    CidrBlock: "10.0.0.0/16",
  });

  // Subnet
  template.resourceCountIs("AWS::EC2::Subnet", 6);
  // Internet Gateway
  template.resourceCountIs("AWS::EC2::InternetGateway", 1);
  template.resourceCountIs("AWS::EC2::VPCGatewayAttachment", 1);
  template.hasResourceProperties("AWS::EC2::VPCGatewayAttachment", {
    VpcId: Match.anyValue(),
    InternetGatewayId: Match.anyValue(),
  });
  // Elastic IP
  template.resourceCountIs("AWS::EC2::EIP", 2);
  // NatGateway
  template.resourceCountIs("AWS::EC2::NatGateway", 2);
  template.hasResourceProperties("AWS::EC2::NatGateway", {
    AllocationId: Match.anyValue(),
    SubnetId: Match.anyValue(),
  });
  template.hasResourceProperties("AWS::EC2::NatGateway", {
```

```
      AllocationId: Match.anyValue(),
      SubnetId: Match.anyValue(),
    });
    // Route Table
    template.resourceCountIs("AWS::EC2::RouteTable", 6);
    template.resourceCountIs("AWS::EC2::Route", 4);
    template.hasResourceProperties("AWS::EC2::Route", {
      RouteTableId: Match.anyValue(),
      DestinationCidrBlock: "0.0.0.0/0",
      GatewayId: Match.anyValue(),
    });
    template.hasResourceProperties("AWS::EC2::Route", {
      RouteTableId: Match.anyValue(),
      DestinationCidrBlock: "0.0.0.0/0",
      NatGatewayId: Match.anyValue(),
    });
    template.resourceCountIs("AWS::EC2::SubnetRouteTableAssociation", 6);
    template.hasResourceProperties("AWS::EC2::SubnetRouteTableAssociation", {
      RouteTableId: Match.anyValue(),
      SubnetId: Match.anyValue(),
    });
  });
```

テストを実行した結果は次のようになります。


jest

CDK のコマンド

- cdk deploy https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/cli.html#cli-deploy CDK で定義されたリソースを AWS 環境にデプロイするコマンドです。 すべてのスタックをデプロイしたい場合は、`--all` オプションを指定します。 指定したスタックをデプロイしたい場合は、deploy の後にスタック名を指定します。
- cdk diff https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/cli.html#cli-diff 現在の CDK とすでにデプロイされているバージョンとの差分を確認し、変更点を一覧で取得するコマンドです。
- cdk synth https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/cli.html#cli-synth CDK で定義されたスタックを CloudFormation テンプレートにするコマンドです。 リソースに付与される Metadata を削除したい場合は、`--path-metadata false` オプションを付与します。 テンプレートは出力せずに、プログラム内に記述した console.log() だけ確認したい場合は、 `--quit` または `-q` オプションを付与します。
- cdk list / cdk ls https://docs.aws.amazon.com/ja_jp/cdk/v2/guide/cli.html#cli-list 現在の CDK アプリに含まれるスタックの一覧を確認するコマンドです。

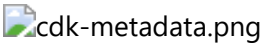
CDK 作成時の Metadata を削除したい場合

CDK で作成される CloudFormation テンプレートファイルには、メタデータが含まれます。バージョンレポートと呼ばれるものです。

```
Resources:
  CDKMetadata:
```

```
    Type: AWS::CDK::Metadata
    Properties:
      Analytics:
v2:deflate64:H4sIAAAAAAAAAEzPUMzQw0TNQdEgsL9ZNTsnWT84vStWrDi5JTM7Wcc7PKy4pKk0u0XFOy
wtKLc4vLUpOBbGBEimZJZn5ebU6efkpqXpZxfplhmZ6hkCDsoozM3WLSvNKMnNT9YIgNAAtXENFZQAAAA=
=
    Metadata:
      aws:cdk:path: DevioStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Conditions:
  CDKMetadataAvailable:
    Fn::Or:
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - af-south-1
:
```

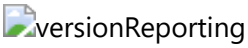
これは、次のような目的で付与される内容です。

 cdk-metadata.png

バージョンレポートを付与したくない場合は、cdk コマンドに以下のオプションを追加して実行します。

```
cdk synth --no-version-reporting --path-metadata false
```

または、cdk.json に `"versionReporting": false,` を追加します。

 versionReporting

GitHub などからダウンロードしてきた CloudFormation テンプレートにこのような Metadata の記載がある場合は、消しても問題ありません。

AWS CDK での開発方法

ディレクトリ構造例

`cdk init --typescript` を実行すると初期ディレクトリが作成されます。 通常は、[lib] ディレクトリにスタックの構成ファイルを配置します。しかし、共通で利用したいものなどが出てきたときに分かりにくくなるので、[stacks] と [utils] などのディレクトリを追加した例を示します。これ以外は、必要になったら [lib] 内にサブディレクトリを作成していきます。 '★' が付いているディレクトリが追加したディレクトリです。

```
プロジェクトルートディレクトリ
    ├ [bin]                    // App定義。複数のスタックの依存関係などを定義
    ├ [lib]
        ├ ★[stacks]            // スタック定義
        ├ ★[resources]         // 各リソースごとの定義。スタックから呼ばれる
        ├ ★[utils]             // 共通使用するものを格納
    ├ ★[parameters]            // 環境依存情報ファイルを格納（※contextを使わない）
```

```
    ├─ ★[src]                    // Lambda や HTML などのソースを格納
    ├─ [test]                   // テスト
    ├─ [node_modules]
    ├─ cdk.context.json         // 環境依存情報(context)
    ├─ cdk.json                 // デフォルトのappなどが入る
    ├─ cdk.out                  // cfnなどの出力先。コンパイルされたjsを動かすと出力される
    ├─ jest.config.js           // テストの設定ファイル
    ├─ package-lock.json
    ├─ package.json
    ├─ tsconfig.json
    ├─ README.md
```

App 定義

`bin` に配置するスタックの依存関係を定義しておくファイルです。

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

const app = new cdk.App();

// env
const defaultEnv = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region: process.env.CDK_DEFAULT_REGION,
};

// よく使うリージョンを定義しておくこともできます
// see: https://docs.aws.amazon.com/ja_jp/general/latest/gr/rande.html#regional-
endpoints
const useast1Env = {
// US East (Virginia)
account: process.env.CDK_DEFAULT_ACCOUNT,
  region: "us-east-1", // リージョンを固定したい場合
};

// 環境識別子の指定 -> 環境識別子はコマンド実行時に '-c project=xxx -c env=xxx' と指定ます
const projectName: string = app.node.tryGetContext('project');
const envName: string = app.node.tryGetContext('env');

//  （オプション）環境識別子のチェック
if (!envname.match(/^(dev|test|stage|prod)$/)) {
  console.warn('Invalid context. envname must be [dev , test, stage, prod].')
  process.exit(1)
}

// スタック
const myStack = new MyStack (app, 'MyStack ', {
  stackName: "ここにスタック名を記述", // 環境識別子をつけたい場合は、`xxxx-${envname}` のように
```

```
できます
  description: "ここにスタックの説明を記述",
  // ここからスタックのパラメータ
  PJName: conf.PJName,
  EnvName: conf.EnvName,
  :
  // ここまでスタックのパラメータ

  // 以下は基本的にどのスタックでも固定で指定する
  env: defaultEnv,
  /* リージョンを固定したい場合は次のようにも指定できます。1回限りの利用の場合はこのような指定も許容します。
  何度も利用する場合は、最初に定義するのを推奨します。
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: "us-east-2",
  },
  */
  terminationProtection: true, // 削除保護の有効化 -> スタック作成と同時に削除保護を有効にできます。ただし、コンソールから解除しないと cdk destroy できなくなります

});

// スタック
const myStack2 = ....
const myStack3 = ....
const myStack4 = ....

// ---------------------------------- Tagging  -----------------------------------
-y
// ここを指定しておくと、全てのスタックにタグ付けしてくれます。
// プロジェクト名や環境識別子は共通タグとして定義します。
cdk.Tags.of(this).add('Project', props.PJName);
cdk.Tags.of(this).add('Environment', props.EnvName);
```

App 定義にリリースミス防止策の例

デプロイするときは、`cdk deploy MyStack -c env=dev --profile xxxxx` として、AWS プロファイル名を指定するのが一般的ですが、これだとプロファイル名を間違えてしまった場合、間違った環境にデプロイされてしまう危険があります。

コンソール上で注意喚起のメッセージがあると、それだけで気付くことがありミス防止にもなります。

その方法は、env で与えられる環境によって、メッセージを追加する方法です。

`prod` とした環境の場合には、次のようにコンソールに表示されます。

 CAUTION

```
// 文字色
const color_red: string = "\u001b[31m";
```

```
const color_green: string = "\u001b[32m";
const color_yellow: string = "\u001b[33m";
const color_white: string = "\u001b[37m";
const color_reset: string = "\u001b[0m";

console.log();
console.log(
  `${color_yellow}##############################################${color_reset}`
);
console.log(`${color_yellow}  プロジェクト名: ${projectName} ${color_reset}`);
console.log(
  `${color_yellow}  リリース環境：${color_reset}
${color_red}${envname}${color_reset}`
);
console.log(
  `${color_yellow}##############################################${color_reset}`
);
console.log();

// 環境識別子のチェック
if (!envname.match(/^(dev|test|stage|prod|jump)$/)) {
  console.warn(
    "Invalid context. envname must be [dev , test, stage, prod, jump]."
  );
  process.exit(1);
}

const isProduction: boolean = envname.match(/^(prod)$/) ? true : false;
if (isProduction) {
  console.log(`${color_red}!!!!!!!!!! CAUTION !!!!!!!!!!${color_reset}`);
  console.log(`${color_red}   本番環境へのリリースです。${color_reset}`);
  console.log(`${color_red}!!!!!!!!!! CAUTION !!!!!!!!!!${color_reset}`);
}
```

スタック定義ファイルの基本構造

`lib/stacks` に配置するスタック定義ファイルです。

```
import { Stack, StackProps , CfnMapping} from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as cdk from 'aws-cdk-lib';
import { // ここにリソース作成に必要なモジュールを列挙します
  aws_ec2 as ec2,
  aws_s3 as s3,
    :
} from 'aws-cdk-lib';

interface IMyStackProps extends StackProps {
  // スタック実行時のパラメータを指定
  readonly PJName: string;
  readonly EnvName: string;
```

```
    :
}

export class MyStack extends Stack {
  // 別のスタックで参照
  public readonly xxx: string;

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


    // タグを付与する -> スタック内の全てにタグ付けしてくれます。
    // 基本的にスタック固有のタグのみを指定し、全体で共通するものは、App 定義のほうでタグを付与します。
    cdk.Tags.of(this).add('hogehoge', 'foobar');
  }
}
```