AWS Cloud Development Kit (CDK)

Developer Guide

AWS Cloud Development Kit (CDK): Developer Guide

Table of Contents

What is the AWS CDK?

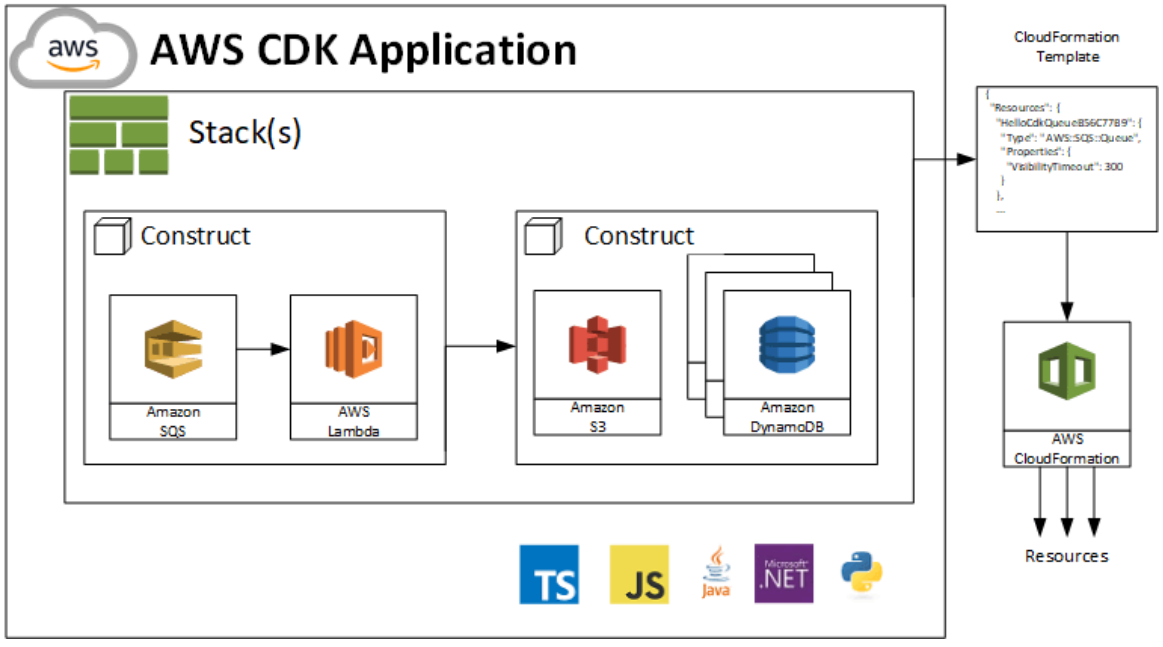
Welcome to the *AWS Cloud Development Kit (CDK) Developer Guide*. This document provides information about the AWS CDK, which is a software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.

AWS CloudFormation enables you to:

- Create and provision AWS infrastructure deployments predictably and repeatedly.
- Leverage AWS products such as Amazon EC2, Amazon Elastic Block Store, Amazon SNS, Elastic Load Balancing, and Auto Scaling.
- Build highly reliable, highly scalable, cost-effective applications in the cloud without worrying about creating and configuring the underlying AWS infrastructure.
- Use a template file to create and delete a collection of resources together as a single unit (a stack).

Use the AWS CDK to define your cloud resources in a familiar programming language. The AWS CDK supports TypeScript, JavaScript, Python, Java, C#/.Net, and (in developer preview) Go.

Developers can use one of the supported programming languages to define reusable cloud components known as Constructs (p. 64). You compose these together into Stacks (p. 85) and Apps (p. 80).



Why use the AWS CDK?

Let's look at the power of the AWS CDK. Here is some code in an AWS CDK project to create an Amazon ECS service with AWS Fargate launch type (this is the code we use in the the section called "ECS" (p. 215)).

TypeScript

```
export class MyEcsConstructStack extends core.Stack {
  constructor(scope: core.App, id: string, props?: core.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 core.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(core.Stack):

    def __init__(self, scope: core.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#

```
using Amazon.CDK;
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;

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,
        });
    }
}
```

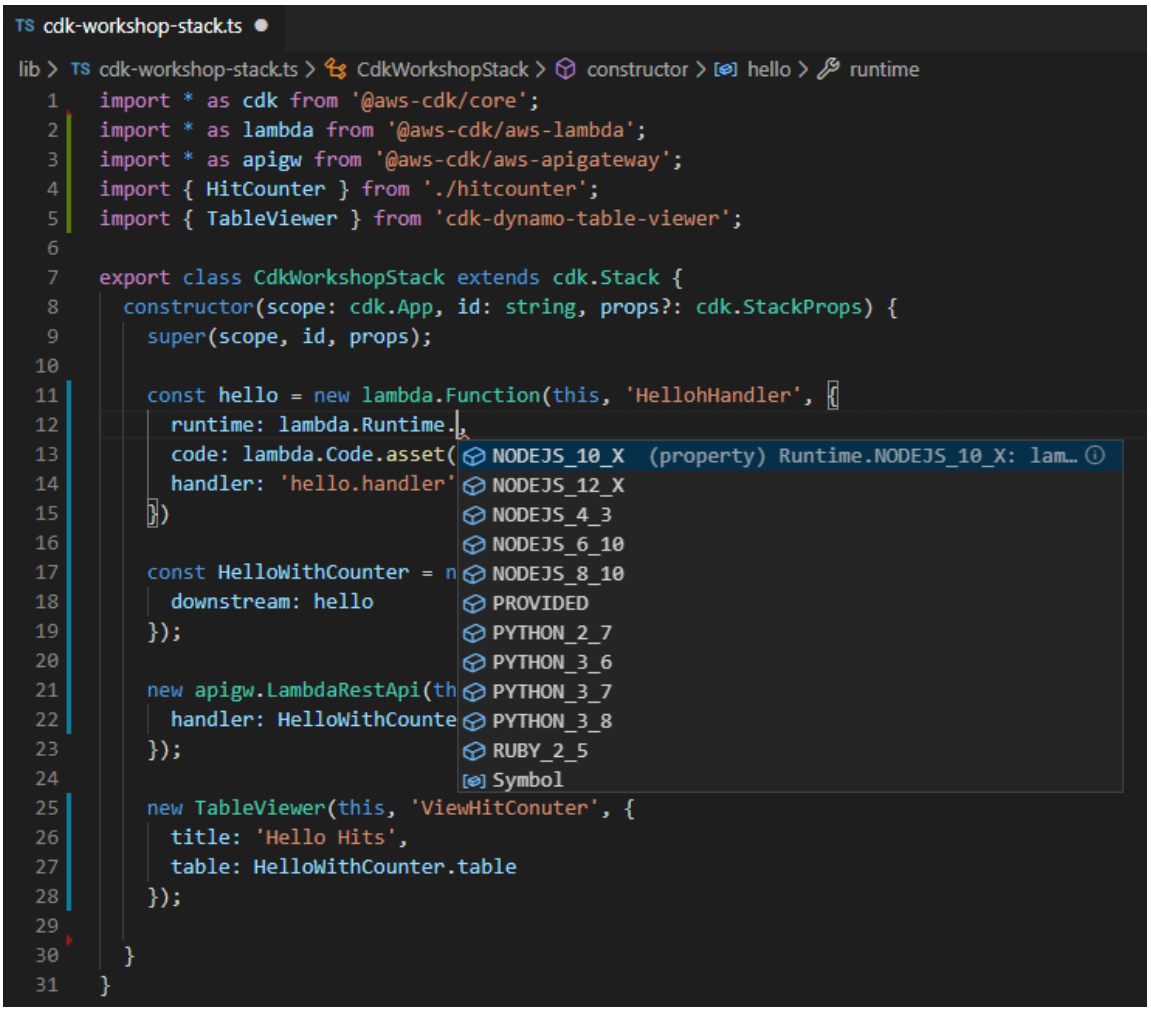
This class produces an AWS CloudFormation template of more than 500 lines; deploying the AWS CDK app produces more than 50 resources of the following types.

- 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

Other advantages of the AWS CDK include:

- Use logic (if statements, for-loops, etc) when defining your infrastructure
- Use object-oriented techniques to create a model of your system
- Define high level abstractions, share them, and publish them to your team, company, or community
- Organize your project into logical modules
- Share and reuse your infrastructure as a library
- Testing your infrastructure code using industry-standard protocols
- Use your existing code review workflow

- Code completion within your IDE



Developing with the AWS CDK

Code snippets and longer examples are available in the AWS CDK's supported programming languages: TypeScript, JavaScript, Python, Java, and C#. See AWS CDK examples (p. 221) for a list of the examples.

The AWS CDK Toolkit (p. 279) is a command line tool for interacting with CDK apps. It enables developers to synthesize artifacts such as AWS CloudFormation templates, deploy stacks to development AWS accounts, and **diff** against a deployed stack to understand the impact of a code change.

The AWS Construct Library (p. 64) includes a module for each AWS service with constructs that offer rich APIs that encapsulate the details of how to create resources for an Amazon or AWS service. The aim of the AWS Construct Library is to reduce the complexity and glue logic required when integrating various AWS services to achieve your goals on AWS.

> **Note**
> There is no charge for using the AWS CDK, but you might incur AWS charges for creating or using AWS chargeable resources, such as running Amazon EC2 instances or using Amazon S3 storage. Use the AWS Pricing Calculator to estimate charges for the use of various AWS resources.

Contributing to the AWS CDK

Because the AWS CDK is open source, the team encourages you to contribute to make it an even better tool. For details, see Contributing.

Additional documentation and resources

In addition to this guide, the following are other resources available to AWS CDK users:

- API Reference
- AWS CDK Workshop
- cdk.dev community hub, including a Slack channel
- AWS CDK Examples
- CDK Patterns
- Awesome CDK
- AWS Solutions Constructs
- AWS Developer Blog CDK category
- Stack Overflow
- GitHub Repository
  - Issues
  - Examples
  - Documentation Source
  - License
  - Releases
    - AWS CDK OpenPGP key (p. 316)
    - JSII OpenPGP key (p. 317)
- AWS CDK Sample for Cloud9
- AWS CloudFormation Concepts
- AWS Glossary

Resources for serverless apps with CDK

These tools can work with the AWS CDK to simplify serverless application development and deployment.

- AWS Serverless Application Model
- AWS Chalice, a Python serverless microframework

About Amazon Web Services

Amazon Web Services (AWS) is a collection of digital infrastructure services that developers can use when developing their applications. The services include computing, storage, database, and application synchronization (messaging and queueing).

AWS uses a pay-as-you-go service model. You are charged only for the services that you — or your applications — use. Also, to make AWS useful as a platform for prototyping and experimentation, AWS

offers a free usage tier, in which services are free below a certain level of usage. For more information about AWS costs and the free usage tier, see Test-Driving AWS in the Free Usage Tier.

To obtain an AWS account, go to aws.amazon.com, and then choose **Create an AWS Account**.

Getting started with the AWS CDK

This topic introduces you to important AWS CDK concepts and describes how to install and configure the AWS CDK. When you're done, you'll be ready to create your first AWS CDK app (p. 13).

Your background

The AWS Cloud Development Kit (CDK) lets you define your cloud infrastructure as code in one of five supported programming languages. It is intended for moderately to highly experienced AWS users.

Ideally, you already have experience with popular AWS services, particularly AWS Identity and Access Management (IAM). You might already have AWS credentials on your workstation for use with an AWS SDK or the AWS CLI and experience working with AWS resources programmatically.

Familiarity with AWS CloudFormation is also useful, as the output of an AWS CDK program is an AWS CloudFormation template.

Finally, you should be proficient in the programming language you intend to use with the AWS CDK.

Key concepts

The AWS CDK is designed around a handful of important concepts. We will introduce a few of these here briefly. Follow the links to learn more, or see the Concepts topics in this guide's Table of Contents.

An AWS CDK app (p. 80) is an application written in TypeScript, JavaScript, Python, Java, or C# that uses the AWS CDK to define AWS infrastructure. An app defines one or more stacks (p. 85). Stacks (equivalent to AWS CloudFormation stacks) contain constructs (p. 64), each of which defines one or more concrete AWS resources, such as Amazon S3 buckets, Lambda functions, Amazon DynamoDB tables, and so on.

> **Note**
> The AWS CDK also supports Go in a developer preview. This Guide does not include instructions or code examples for Go aside from the section called “In Go” (p. 49).

Constructs (as well as stacks and apps) are represented as types in your programming language of choice. You instantiate constructs within a stack to declare them to AWS, and connect them to each other using well-defined interfaces.

The AWS CDK includes the AWS CDK Toolkit (also called the CLI), a command-line tool for working with your AWS CDK apps and stacks. Among other functions, the Toolkit provides the ability to convert one or more AWS CDK stacks to AWS CloudFormation templates and related assets (a process called *synthesis*) and to deploy your stacks to an AWS account.

The AWS CDK includes a library of AWS constructs called the AWS Construct Library. Each AWS service has at least one corresponding module in the library containing the constructs that represent that service's resources.

Constructs come in three fundamental flavors:

- **AWS CloudFormation-only** or L1 (short for "level 1"). These constructs correspond directly to resource types defined by AWS CloudFormation. In fact, these constructs are automatically generated from the AWS CloudFormation specification, so when a new AWS service is launched, the AWS CDK supports it as soon as AWS CloudFormation does.

AWS CloudFormation resources always have names that begin with `Cfn`. For example, in the Amazon S3 module, `CfnBucket` is the L1 module for an Amazon S3 bucket.

- **Curated** or L2. These constructs are carefully developed by the AWS CDK team to address specific use cases and simplify infrastructure development. For the most part, they encapsulate L1 modules, providing sensible defaults and best-practice security policies. For example, in the Amazon S3 module, `Bucket` is the L2 module for an Amazon S3 bucket.

  L2 modules may also define supporting resources needed by the primary resource. Some services have more than one L2 module in the Construct Library for organizational purposes.
- **Patterns** or L3. Patterns declare multiple resources to create entire AWS architectures for particular use cases. All the plumbing is already hooked up, and configuration is boiled down to a few important parameters. In the AWS Construct Library, patterns are in separate modules from L1 and L2 constructs.

The AWS CDK's core module (usually imported into code as `core` or `cdk`) contains constructs used by the AWS CDK itself as well as base classes for constructs, apps, resources, and other AWS CDK objects.

Supported programming languages

The AWS CDK has first-class support for TypeScript, JavaScript, Python, Java, and C#. (Other JVM and .NET CLR languages may also be used, at least in theory, but we are unable to offer support for them at this time.)

To facilitate supporting so many languages, the AWS CDK is developed in one language (TypeScript) and language bindings are generated for the other languages through the use of a tool called JSII.

We have taken pains to make AWS CDK app development in each language follow that language's usual conventions, so writing AWS CDK apps feels natural, not like writing TypeScript in Python (for example). Take a look:

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(self, "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"
                        }});
```

Note
These code snippets are intended for illustration only. They are incomplete and won't run as they are.

The AWS Construct Library is distributed using each language's standard package management tools, including NPM, PyPi, Maven, and NuGet. There's even a version of the AWS CDK API Reference for each language.

To help you use the AWS CDK in your favorite language, this Guide includes topics that explain how to use the AWS CDK in all supported languages.

- the section called “In TypeScript” (p. 26)
- the section called “In JavaScript” (p. 30)
- the section called “In Python” (p. 36)
- the section called “In Java” (p. 41)
- the section called “In C#” (p. 44)

TypeScript was the first language supported by the AWS CDK, and much AWS CDK example code is written in TypeScript. This Guide includes a topic specifically to show how to adapt TypeScript AWS CDK code for use with the other supported languages. See *Translating from TypeScript* (p. 58).

Prerequisites

Here's what you need to install to use the AWS CDK.

All AWS CDK developers, even those working in Python, Java, or C#, need Node.js 10.13.0 or later. All supported languages use the same back end, which runs on Node.js. We recommend a version in active long-term support, which, at this writing, is the latest 14.x release. Your organization may have a different recommendation.

Important
Node.js versions 13.0.0 through 13.6.0 are not compatible with the AWS CDK due to compatibility issues with its dependencies.

You must configure your workstation with your credentials and an AWS region, if you have not already done so. If you have the AWS CLI installed, the easiest way to satisfy this requirement is issue the following command:

```
aws configure
```

Provide your AWS access key ID, secret access key, and default region when prompted.

You may also manually create or edit the `~/.aws/config` and `~/.aws/credentials` (macOS/Linux) or `%USERPROFILE%\.aws\config` and `%USERPROFILE%\.aws\credentials` (Windows) files to contain credentials and a default region, in the following format.

- In `~/.aws/config` or `%USERPROFILE%\.aws\config`

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

- In `~/.aws/credentials` or `%USERPROFILE%\.aws\credentials`

```
[default]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
```

> **Note**
> Although the AWS CDK uses credentials from the same configuration files as other AWS tools and SDKs, including the AWS Command Line Interface, it may behave slightly differently from these tools. In particular, if you use a named profile from the `credentials` file, the `config` must have a profile of the same name specifying the region. The AWS CDK does not fall back to reading the region from the `[default]` section in `config`. Also, do not use a profile named "default" (e.g. `[profile default]`). See Setting credentials for complete details on setting up credentials for the AWS SDK for JavaScript, which the AWS CDK uses under the hood.

Alternatively, you can set the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` to appropriate values.

> **Important**
> We strongly recommend against using your AWS root account for day-to-day tasks. Instead, create a user in IAM and use its credentials with the CDK. Best practices are to change this account's access key regularly and to use a least-privileges role (specifying `--role-arn`) when deploying.

Other prerequisites depend on the language in which you develop AWS CDK applications and are as follows.

TypeScript

- TypeScript 2.7 or later (`npm -g install typescript`)

JavaScript

No additional requirements

Python

- Python 3.6 or later including `pip` and `virtualenv`

Java

- Java Development Kit (JDK) 8 (a.k.a. 1.8) or later
- Apache Maven 3.5 or later

Java IDE recommended (we use Eclipse in some examples in this Developer Guide). IDE must be able to import Maven projects. Check to make sure your project is set to use Java 1.8. Set the JAVA_HOME environment variable to the path where you have installed the JDK.

C#

.NET Core 3.1 or later.

Visual Studio 2019 (any edition) or Visual Studio Code recommended.

Install the AWS CDK

Install the AWS CDK Toolkit globally using the following Node Package Manager command.

```
npm install -g aws-cdk
```

Run the following command to verify correct installation and print the version number of the AWS CDK.

```
cdk --version
```

Bootstrapping

Many AWS CDK stacks that you write will include assets (p. 138): external files that are deployed with the stack, such as AWS Lambda functions or Docker images. The AWS CDK uploads these to an Amazon S3 bucket or other container so they are available to AWS CloudFormation during deployment. Deployment requires that these containers already exist in the account and region you are deploying into. Creating them is called bootstrapping (p. 174). To bootstrap, issue:

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

Tip
If you don't have your AWS account number handy, you can get it from the AWS Management Console. Or, if you have the AWS CLI installed, the following command displays your default account information, including the account number.

```
aws sts get-caller-identity
```

If you have created named profiles in your local AWS configuration, you can use the `--profile` option to display the account information for a specific profile's account, such as the *prod* profile as shown here.

```
aws sts get-caller-identity --profile prod
```

To display the default region, use `aws configure get`.

```
aws configure get region
aws configure get region --profile prod
```

AWS CDK tools

The AWS CDK Toolkit, also known as the Command Line Interface (CLI), is the main tool you use to interact with your AWS CDK app. It executes your code and produces and deploys the AWS

CloudFormation templates it generates. It also has deployment, diff, deletion, and troubleshooting capabilities. For more information, see **cdk --help** or the section called "AWS CDK Toolkit" (p. 279).

The AWS Toolkit for Visual Studio Code is an open-source plug-in for Visual Studio Code that makes it easier to create, debug, and deploy applications on AWS. The toolkit provides an integrated experience for developing AWS CDK applications, including the AWS CDK Explorer feature to list your AWS CDK projects and browse the various components of the CDK application. Install the plug-in and learn more about using the AWS CDK Explorer.

Next steps

Where do you go now that you've dipped your toes in the AWS CDK?

- Come on in; the water's fine! Build your first AWS CDK app (p. 13).
- Try the CDK Workshop for a more in-depth tour involving a more complex project.
- See the API reference to begin exploring the CDK constructs available for your favorite AWS services.
- Dig deeper into concepts like the section called "Environments" (p. 92), the section called "Assets" (p. 138), the section called "Bootstrapping" (p. 174), the section called "Permissions" (p. 151), the section called "Context" (p. 159), the section called "Parameters" (p. 128), and the section called "Escape hatches" (p. 168).
- Explore Examples of using the AWS CDK.

The AWS CDK is an open-source project. Want to contribute?

Your first AWS CDK app

You've read *Getting started* (p. 8) and set up your development environment for writing AWS CDK apps? Great! Now let's see how it feels to work with the AWS CDK by building the simplest possible AWS CDK app.

> **Note**
> The AWS CDK supports Go in a developer preview. This tutorial does not include instructions or code examples for Go.

In this tutorial, you'll learn about the structure of a AWS CDK project, how to use the AWS Construct Library to define AWS resources using code, and how to synthesize, diff, and deploy collections of resources using the AWS CDK Toolkit command-line tool.

The standard AWS CDK development workflow is similar to the workflow you're already familiar with as a developer, just with a few extra steps.

1. Create the app from a template provided by the AWS CDK
2. Add code to the app to create resources within stacks
3. Build the app (optional; the AWS CDK Toolkit will do it for you if you forget)
4. Synthesize one or more stacks in the app to create an AWS CloudFormation template
5. Deploy one or more stacks to your AWS account

The build step catches syntax and type errors. The synthesis step catches logical errors in defining your AWS resources. The deployment may find permission issues. As always, you go back to the code, find the problem, fix it, then build, synthesize and deploy again.

Tip
Don't forget to keep your AWS CDK code under version control!

This tutorial walks you through creating and deploying a simple AWS CDK app, from initializing the project to deploying the resulting AWS CloudFormation template. The app contains one stack, which contains one resource: an Amazon S3 bucket.

We'll also show what happens when you make a change and re-deploy, and how to clean up when you're done.

Create the app

Each AWS CDK app should be in its own directory, with its own local module dependencies. Create a new directory for your app. Starting in your home directory, or another directory if you prefer, issue the following commands.

Important
Be sure to name your project directory `hello-cdk`, *exactly as shown here.* The AWS CDK project template uses the directory name to name things in the generated code, so if you use a different name, the code in this tutorial won't work.

```
mkdir hello-cdk
cd hello-cdk
```

Now initialize the app using the **cdk init** command, specifying the desired template ("app") and programming language. That is:

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

After the app has been created, also enter the following two commands to activate the app's Python virtual environment and install the AWS CDK core dependencies.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

C#

```
cdk init app --language csharp
```

If you are using Visual Studio, open the solution file in the `src` directory.

Tip
If you don't specify a template, the default is "app," which is the one we wanted anyway, so technically you can leave it out and save four keystrokes.

The **cdk init** command creates a number of files and folders inside the `hello-cdk` directory to help you organize the source code for your AWS CDK app. Take a moment to explore. The structure of a basic app is all there; you'll fill in the details in this tutorial.

If you have Git installed, each project you create using **cdk init** is also initialized as a Git repository. We'll ignore that for now, but it's there when you need it.

Build the app

In most programming environments, after making changes to your code, you'd build (compile) it. This isn't strictly necessary with the AWS CDK—the Toolkit does it for you so you can't forget. But you can still build manually whenever you want to catch syntax and type errors. For reference, here's how.

TypeScript

```
npm run build
```

JavaScript

No build step is necessary.

Python

No build step is necessary.

Java

```
mvn compile -q
```

Or press Control-B in Eclipse (other Java IDEs may vary)

C#

```
dotnet build src
```

Or press F6 in Visual Studio

Note
If your project was created with an older version of the AWS CDK Toolkit, it may not automatically build when you run it. If changes you make in your code fail to be reflected in the synthesized template, try a manual build. Make sure you are using the latest available version of the AWS CDK for this tutorial.

List the stacks in the app

Just to verify everything is working correctly, list the stacks in your app.

```
cdk ls
```

If you don't see `HelloCdkStack`, make sure you named your app's directory `hello-cdk`. If you didn't, go back to the section called “Create the app” (p. 14) and try again.

Add an Amazon S3 bucket

At this point, your app doesn't do anything because the stack it contains doesn't define any resources. Let's add an Amazon S3 bucket.

Install the Amazon S3 package from the AWS Construct Library.

TypeScript

```
npm install @aws-cdk/aws-s3
```

JavaScript

```
npm install @aws-cdk/aws-s3
```

Python

```
pip install aws-cdk.aws-s3
```

Java

Add the following to the `<dependencies>` container of `pom.xml`.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

Tip
If you are using a Java IDE, it probably has a simpler way to add this dependency to your project, such as a GUI for editing the POM. We recommend editing `pom.xml` by hand because of the use of the `cdk.version` variable, which helps keep the versions of installed modules consistent.

C#

Run the following command in the `src/HelloCdk` directory.

```
dotnet add package Amazon.CDK.AWS.S3
```

Or **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio, then locate and install the `Amazon.CDK.AWS.S3` package

Next, define an Amazon S3 bucket in the stack using the Bucket construct.

TypeScript

In `lib/hello-cdk-stack.ts`:

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

```
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

In `lib/hello-cdk-stack.js`:

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

Add the following import statement in `hello_cdk_stack.py` in the `hello_cdk` directory below the existing imports.

```
from aws_cdk import aws_s3 as s3
```

Replace the comment with the following code.

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

Java

In `src/main/java/com/myorg/HelloCdkStack.java`:

```
package com.myorg;

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

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#

In `HelloCdkStack.cs`:

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

namespace HelloCdk
{
    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
            });
        }
    }
}
```

`Bucket` is the first construct we've seen, so let's take a closer look. Like all constructs, the `Bucket` class takes three parameters.

- **scope:** Tells the bucket that the stack is its parent: it is defined within the scope of the stack. You can define constructs inside of constructs, creating a hierarchy (tree). Here, and in most cases, the scope is `this` (`self` in Python), meaning the construct that contains the bucket: the stack.
- **Id:** The logical ID of the Bucket within your AWS CDK app. This (plus a hash based on the bucket's location within the stack) uniquely identifies the bucket across deployments so the AWS CDK can update it if you change how it's defined in your app. Here it is "MyFirstBucket." Buckets can also have a name, which is separate from this ID (it's the `bucketName` property).
- **props:** A bundle of values that define properties of the bucket. Here we've defined only one property: `versioned`, which enables versioning for the files in the bucket.

All constructs take these same three arguments, so it's easy to stay oriented as you learn about new ones. And as you might expect, you can subclass any construct to extend it to suit your needs, or just to change its defaults.

Tip
If a construct's props are all optional, you can omit the `props` parameter entirely.

Props are represented differently in the languages supported by the AWS CDK.

- In TypeScript and JavaScript, `props` is a single argument and you pass in an object containing the desired properties.
- In Python, props are passed as keyword arguments.
- In Java, a Builder is provided to pass the props. Two, actually; one for `BucketProps`, and a second for `Bucket` to let you build the construct and its props object in one step. This code uses the latter.
- In C#, you instantiate a `BucketProps` object using an object initializer and pass it as the third parameter.

Synthesize an AWS CloudFormation template

Synthesize an AWS CloudFormation template for the app, as follows.

```
cdk synth
```

If your app contained more than one stack, you'd need to specify which stack(s) to synthesize. But since it only contains one, the CDK Toolkit knows you must mean that one.

> **Tip**
> If you received an error like `--app is required...`, it's probably because you are running the command from a subdirectory. Navigate to the main app directory and try again.

The `cdk synth` command executes your app, which causes the resources defined in it to be translated into an AWS CloudFormation template. The displayed output of `cdk synth` is a YAML-format template; the beginning of our app's output is shown below. The template is also saved in the `cdk.out` directory in JSON format.

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
    Metadata:...
```

Even if you aren't very familiar with AWS CloudFormation, you should be able to find the definition for the bucket and see how the `versioned` property was translated.

> **Note**
> Every generated template contains a `AWS::CDK::Metadata` resource by default. (We haven't shown it here.) The AWS CDK team uses this metadata to gain insight into how the AWS CDK is used, so we can continue to improve it. For details, including how to opt out of version reporting, see Version reporting (p. 281).

The `cdk synth` generates a perfectly valid AWS CloudFormation template. You could take it and deploy it using the AWS CloudFormation console or another tool. But the AWS CDK Toolkit can also do that.

Deploying the stack

To deploy the stack using AWS CloudFormation, issue:

```
cdk deploy
```

As with `cdk synth`, you don't need to specify the name of the stack since there's only one in the app.

It is optional (though good practice) to synthesize before deploying. The AWS CDK synthesizes your stack before each deployment.

If your code has security implications, you'll see a summary of these and need to confirm them before deployment proceeds. This isn't the case in our stack.

`cdk deploy` displays progress information as your stack is deployed. When it's done, the command prompt reappears. You can go to the AWS CloudFormation console and see that it now lists `HelloCdkStack`. You'll also find `MyFirstBucket` in the Amazon S3 console.

You've deployed your first stack using the AWS CDK—congratulations! But that's not all there is to the AWS CDK.

Modifying the app

The AWS CDK can update your deployed resources after you modify your app. Let's change our bucket so it can be automatically deleted when we delete the stack, which involves changing its `RemovalPolicy`. Also, because AWS CloudFormation won't delete Amazon S3 buckets that contain any objects, we'll ask the AWS CDK to delete the objects from our bucket before destroying the bucket, via the `autoDeleteObjects` property..

TypeScript

Update `lib/hello-cdk-stack.ts`.

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

JavaScript

Update `lib/hello-cdk-stack.js`.

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

Python

Update `hello_cdk/hello_cdk_stack.py`.

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

Java

Update `src/main/java/com/myorg/HelloCdkStack.java`, adding the new import and updating the bucket definition in the appropriate places.

```
import software.amazon.awscdk.core.RemovalPolicy;
```

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

C#

Update `HelloCdkStack.cs`.

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

```
{
    Versioned = true,
    RemovalPolicy = RemovalPolicy.DESTROY,
    AutoDeleteObjects = true
});
```

Here, we haven't written any code that, in itself, changes our Amazon S3 bucket. Instead, our code defines the desired state of the bucket. The AWS CDK synthesizes that state to a new AWS CloudFormation template and deploys a changeset that makes only the changes necessary to reach that state.

To see these changes, we'll use the `cdk diff` command .

```
cdk diff
```

The AWS CDK Toolkit queries your AWS account for the last-deployed AWS CloudFormation template for the `HelloCdkStack` and compares it with the template it just synthesized from your app. The output should look like the following.

```
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
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3F:
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
```

```
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF93626:
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56CD69A:
 {"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
```

This diff has four sections.

- **IAM Statement Changes** and **IAM Policy Changes** - These permission changes are there because we set the `AutoDeleteObjects` property on our Amazon S3 bucket. The auto-delete feature uses a custom resource to delete the objects in the bucket before the bucket itself is deleted. The IAM objects grant the custom resource's code access to the bucket.
- **Parameters** - The AWS CDK uses these entries to locate the Lambda function asset for the custom resource.
- **Resources** - The new and changed resources in this stack. We can see the aforementioned IAM objects, the custom resource ,and its associated Lambda function being added. We can also see that the bucket's `DeletionPolicy` and `UpdateReplacePolicy` attributes are being updated. These allow the bucket to be deleted along with the stack, and to be replaced with a new one.

You may be curious about why we specified `RemovalPolicy` in our AWS CDK app but got a `DeletionPolicy` property in the resulting AWS CloudFormation template. The AWS CDK uses a different name for the property because the AWS CDK default is to retain the bucket when the stack is deleted, while AWS CloudFormation's default is to delete it. See the section called “Removal policies” (p. 115) for further details.

It's informative to compare the output of **cdk synth** here with the previous output and see the many additional lines of AWS CloudFormation template that the AWS CDK generated for us based on these relatively small changes.

> **Important**
> Since the `autoDeleteObjects` property is implemented using a AWS CloudFormation custom resource, which is implemented using an AWS Lambda function, our stack contains an asset (p. 138). This fact requires that our AWS account and region be bootstrapped (p. 174) so that there's an Amazon S3 bucket to hold the asset during deployment. If you haven't already bootstrapped, issue:
>
> ```
> cdk bootstrap aws://ACCOUNT-NUMBER/REGION
> ```

Now let's deploy.

```
cdk deploy
```

The AWS CDK warns you about the security policy changes we've already seen in the diff. Enter **y** to approve the changes and deploy the updated stack. The CDK Toolkit updates the bucket configuration as you requested.

```
HHelloCdkStack: 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
```

Destroying the app's resources

Now that you're done with the quick tour, destroy your app's resources to avoid incurring any costs from the bucket you created, as follows.

```
cdk destroy
```

Enter **y** to approve the changes and delete any stack resources.

> **Note**
> If we hadn't changed the bucket's `RemovalPolicy`, the stack deletion would complete successfully, but the bucket would become orphaned (no longer associated with the stack).

Next steps

Where do you go now that you've dipped your toes in the AWS CDK?

- Try the CDK Workshop for a more in-depth tour involving a more complex project.
- See the API reference to begin exploring the CDK constructs available for your favorite AWS services.
- Dig deeper into concepts like the section called “Environments” (p. 92), the section called “Assets” (p. 138), the section called “Permissions” (p. 151), the section called “Context” (p. 159), the section called “Parameters” (p. 128), and the section called “Escape hatches” (p. 168).
- Explore Examples of using the AWS CDK.

The AWS CDK is an open-source project. Want to contribute?

Working with the AWS CDK

The AWS Cloud Development Kit (CDK) lets you define your AWS cloud infrastructure in a general-purpose programming language. Currently, the AWS CDK supports TypeScript, JavaScript, Python, Java, C#, and (in developer preview) Go. It is also possible to use other JVM and .NET languages, though we are unable to provide support for every such language.

> **Note**
> This Guide does not include instructions or code examples for Go aside from the section called "In Go" (p. 49).

We develop the AWS CDK in TypeScript and use JSII to provide a "native" experience in other supported languages. For example, we distribute AWS Construct Library modules using your preferred language's standard repository, and you install them using the language's standard package manager. Methods and properties are even named using your language's recommended naming patterns.

AWS CDK prerequisites

To use the AWS CDK, you need an AWS account and a corresponding access key. If you don't have an AWS account yet, see Create and Activate an AWS Account. To find out how to obtain an access key ID and secret access key for your AWS account, see Understanding and Getting Your Security Credentials. To find out how to configure your workstation so the AWS CDK uses your credentials, see Setting Credentials in Node.js.

> **Tip**
> If you have the AWS CLI installed, the simplest way to set up your workstation with your AWS credentials is to open a command prompt and type:

```
aws configure
```

All AWS CDK applications require Node.js 10.13 or later, even if you work in Python, Java, or C#. You may download a compatible version at nodejs.org. We recommend the active LTS version (at this writing, the latest 14.x release). Node.js versions 13.0.0 through 13.6.0 are not compatible with the AWS CDK due to compatibility issues with its dependencies.

After installing Node.js, install the AWS CDK Toolkit (the `cdk` command):

```
npm install -g aws-cdk
```

> **Note**
> If you get a permission error, and have administrator access on your system, try `sudo npm install -g aws-cdk`.

Test the installation by issuing `cdk --version`.

If you get an error message at this point, try uninstalling (`npm uninstall -g aws-cdk`) and reinstalling. As a last resort, delete the `node-modules` folder from the current project as well as the global `node-modules` folder. To figure out where this folder is, issue `npm config get prefix`.

The specific language you work in also has its own prerequisites, described in the corresponding topic listed here.

Topics

Working with the AWS CDK in TypeScript

TypeScript is a fully-supported client language for the AWS CDK and is considered stable. Working with the AWS CDK in TypeScript uses familiar tools, including Microsoft's TypeScript compiler (`tsc`), Node.js and the Node Package Manager (`npm`). You may also use Yarn if you prefer, though the examples in this Guide use NPM. The modules comprising the AWS Construct Library are distributed via the NPM repository, npmjs.org.

You can use any editor or IDE; many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has excellent support for TypeScript.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

You also need TypeScript itself. If you don't already have it, you can install it using `npm`.

```
npm install -g typescript
```

> **Note**
> If you get a permission error, and have administrator access on your system, try `sudo npm install -g typescript`.

Keep TypeScript up to date with a regular `npm update -g typescript`.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language typescript
```

Creating a project also installs the `core` module and its dependencies.

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

Using local `tsc` and `cdk`

For the most part, this guide assumes you install TypeScript and the CDK Toolkit globally (`npm install -g typescript aws-cdk`), and the provided command examples (such as `cdk synth`) follow this assumption. This approach makes it easy to keep both components up to date, and since both take a strict approach to backward compatibility, there is generally little risk in always using the latest versions.

Some teams prefer to specify all dependencies within each project, including tools like the TypeScript compiler and the CDK Toolkit. This practice lets you pin these components to specific versions and

ensure that all developers on your team (and your CI/CD environment) use exactly those versions. This eliminates a possible source of change, helping to make builds and deployments more consistent and repeatable.

The CDK includes dependencies for both TypeScript and the CDK Toolkit in the TypeScript project template's `package.json`, so if you want to use this approach, you don't need to make any changes to your project. All you need to do is use slightly different commands for building your app and for issunig `cdk` commands.

Operation	Use global tools	Use local tools
Initialize project	`cdk init --language typescript`	`npx aws-cdk init --language typescript`
Build	`tsc`	`npm run build`
Run CDK Toolkit command	`cdk ...`	`npm run cdk ...` or `npx aws-cdk ...`

`npx aws-cdk` runs the version of the CDK Toolkit installed locally in the current project, if one exists, falling back to the global installation, if any. If no global installation exists, `npx` downloads a temporary copy of the CDK Toolkit and runs that. You may specify an arbitrary version of the CDK Toolkit using the `@` syntax: `npx aws-cdk@1.120 --version` prints `1.120.0`.

Tip
Set up an alias so you can use the `cdk` command with a local CDK Toolkit installation.

macOS/Linux

```
alias cdk=npx aws-cdk
```

Windows

```
doskey cdk=npx aws-cdk $*
```

Managing AWS Construct Library modules

Use the Node Package Manager (`npm`) to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. (You may use `yarn` instead of `npm` if you prefer.) `npm` also installs the dependencies for those modules automatically.

The AWS CDK core module is named `@aws-cdk/core`. AWS Construct Library modules are named like `@aws-cdk/`*`SERVICE-NAME`*. The service name has an *a* prefix. If you're unsure of a module's name, search for it on NPM.

Note
The CDK API Reference also shows the package names.

For example, the command below installs the modules for Amazon S3 and AWS Lambda.

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

Some services' Construct Library support is in more than one module. For example, besides the `@aws-cdk/aws-route53` module, there are three additional Amazon Route 53 modules, named `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

Your project's dependencies are maintained in `package.json`. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project's NPM dependencies to the latest permitted version according to the rules you specified in `package.json`:

```
npm update
```

In TypeScript, you import modules into your code under the same name you use to install them using NPM. We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Use ES6-style `import` directives, not `require()`.
- Generally, import individual classes from `@aws-cdk/core`.

```
import { App, Construct } from '@aws-cdk/core';
```

- If you need many classes from the core module, you may use a namespace alias of `cdk` instead of importing the individual classes. Avoid doing both.

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

- Generally, import AWS Construct Libraries using short namespace aliases.

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

Important
All AWS Construct Library modules used in your project must be the same version.

AWS CDK idioms in TypeScript

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the AWS resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In TypeScript, the shape of `props` is defined using an interface that tells you the required and optional arguments and their types. Such an interface is defined for each kind of `props` argument, usually specific to a single construct or method. For example, the Bucket construct (in the `@aws-cdk/aws-s3 module`) specifies a `props` argument conforming to the BucketProps interface.

If a property is itself an object, for example the websiteRedirect property of `BucketProps`, that object will have its own interface to which its shape must conform, in this case RedirectTarget.

If you are subclassing an AWS Construct Library class (or overriding a method that takes a props-like argument), you can inherit from the existing interface to create a new one that specifies any new props your code requires. When calling the parent class or base method, generally you can pass the entire props argument you received, since any attributes provided in the object but not specified in the interface will be ignored.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. Passing the value you receive up the inheritance chain can then cause unexpected

behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to `undefined`. For example:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

Alternatively, name your properties so that it is clear that they belong to your construct. This way, it is unlikely they will collide with properties in future AWS CDK releases. If there are many of them, use a single appropriately-named object to hold them.

Missing values

Missing values in an object (such as props) have the value `undefined` in TypeScript. Recent versions of the language include operators that simplify working with these values, making it easier to specify defaults and "short-circuit" chaining when an undefined value is reached. For more information about these features, see the TypeScript 3.7 Release Notes, specifically the first two features, Optional Chaining and Nullish Coalescing.

Building, synthesizing, and deploying

Generally, you should be in the project's root directory when building and running your application.

Node.js cannot run TypeScript directly; instead, your application is converted to JavaScript using the TypeScript compiler, `tsc`. The resulting JavaScript code is then executed.

The AWS CDK automatically does this whenever it needs to run your app. However, it can be useful to compile manually to check for errors and to run tests. To compile your TypeScript app manually, issue `npm run build`. You may also issue `npm run watch` to enter watch mode, in which the TypeScript compiler automatically rebuilds your app whenever you save changes to a source file.

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 279).

Working with the AWS CDK in JavaScript

JavaScript is a fully-supported client language for the AWS CDK and is considered stable. Working with the AWS CDK in JavaScript uses familiar tools, including Node.js and the Node Package Manager (`npm`). You may also use Yarn if you prefer, though the examples in this Guide use NPM. The modules comprising the AWS Construct Library are distributed via the NPM repository, npmjs.org.

You can use any editor or IDE; many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has good support for JavaScript.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

JavaScript AWS CDK applications require no additional prerequisites beyond these.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language javascript
```

Creating a project also installs the `core` module and its dependencies.

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

Using local `cdk`

For the most part, this guide assumes you install the CDK Toolkit globally (`npm install -g aws-cdk`), and the provided command examples (such as `cdk synth`) follow this assumption. This approach makes it easy to keep the CDK Toolkit up to date, and since the CDK takes a strict approach to backward compatibility, there is generally little risk in always using the latest version.

Some teams prefer to specify all dependencies within each project, including tools like the CDK Toolkit. This practice lets you pin such components to specific versions and ensure that all developers on your team (and your CI/CD environment) use exactly those versions. This eliminates a possible source of change, helping to make builds and deployments more consistent nand repeatable.

The CDK includes a dependency for the CDK Toolkit in the JavaScript project template's `package.json`, so if you want to use this approach, you don't need to make any changes to your project. All you need to do is use slightly different commands for building your app and for issunig `cdk` commands.

Operation	Use global CDK Toolkit	Use local CDK Toolkit
Initialize project	`cdk init --language javascript`	`npx aws-cdk init --language javascript`
Run CDK Toolkit command	`cdk ...`	`npm run cdk ...` or `npx aws-cdk ...`

`npx aws-cdk` runs the version of the CDK Toolkit installed locally in the current project, if one exists, falling back to the global installation, if any. If no global installation exists, `npx` downloads a temporary copy of the CDK Toolkit and runs that. You may specify an arbitrary version of the CDK Toolkit using the `@` syntax: `npx aws-cdk@1.120 --version` prints `1.120.0`.

Tip
Set up an alias so you can use the `cdk` command with a local CDK Toolkit installation.

macOS/Linux

```
alias cdk=npx aws-cdk
```

Windows

```
doskey cdk=npx aws-cdk
```

Managing AWS Construct Library modules

Use the Node Package Manager (`npm`) to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. (You may use `yarn` instead of `npm` if you prefer.) `npm` also installs the dependencies for those modules automatically.

The AWS CDK core module is named `@aws-cdk/core`. AWS Construct Library modules are named like `@aws-cdk/`*`SERVICE-NAME`*. The service name has an *aws-* prefix. If you're unsure of a module's name, search for it on NPM.

Note
The CDK API Reference also shows the package names.

For example, the command below installs the modules for Amazon S3 and AWS Lambda.

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

Some services' Construct Library support is in more than one module. For example, besides the `@aws-cdk/aws-route53` module, there are three additional Amazon Route 53 modules, named `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

Your project's dependencies are maintained in `package.json`. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project's NPM dependencies to the latest permitted version according to the rules you specified in `package.json`:

```
npm update
```

In JavaScript, you import modules into your code under the same name you use to install them using NPM. We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Use `require()`, not ES6-style `import` directives. Most of the versions of Node.js that the AWS CDK runs on do not support ES6 imports, so using the older syntax is more widely compatible. (If you really want to use ES6 imports, use esm to ensure your project is compatible with all supported versions of Node.js.)
- Generally, import individual classes from `@aws-cdk/core`.

```
const { App, Construct } = require('@aws-cdk/core');
```

- If you need many classes from the core module, you may use a namespace alias of `cdk` instead of importing the individual classes. Avoid doing both.

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

- Generally, import AWS Construct Libraries using short namespace aliases.

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

Important
All AWS Construct Library modules used in your project must be the same version.

AWS CDK idioms in JavaScript

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the AWS resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

Using an IDE or editor that has good JavaScript autocomplete will help avoid misspelling property names. If a construct is expecting an `encryptionKeys` property, and you spell it `encryptionkeys`, when instantiating the construct, you haven't passed the value you intended. This can cause an error at synthesis time if the property is required, or cause the property to be silently ignored if it is optional. In the latter case, you may get a default behavior you intended to override. Take special care here.

When subclassing an AWS Construct Library class (or overriding a method that takes a props-like argument), you may want to accept additional properties for your own use. These values will be ignored by the parent class or overridden method, because they are never accessed in that code, so you can generally pass on all the props you received.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. Passing the value you receive up the inheritance chain can then cause unexpected behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to `undefined`. For example:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

Alternatively, name your properties so that it is clear that they belong to your construct. This way, it is unlikely they will collide with properties in future AWS CDK releases. If there are many of them, use a single appropriately-named object to hold them.

Missing values

Missing values in an object (such as `props`) have the value `undefined` in JavaScript. The usual techniques apply for dealing with these. For example, a common idiom for accessing a property of a value that may be undefined is as follows:

```
// 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;
```

However, if `a` could have some other "falsy" value besides `undefined`, it is better to make the test more explicit. Here, we'll take advantage of the fact that `null` and `undefined` are equal to test for them both at once:

```
let c = a == null ? a : a.b;
```

Tip
Node.js 14.0 and later support new operators that can simplify the handling of undefined values. For more information, see the optional chaining and nullish coalescing proposals.

Synthesizing and deploying

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 279).

Using TypeScript examples with JavaScript

TypeScript is the language we use to develop the AWS CDK, and it was the first language supported for developing applications, so many available AWS CDK code examples are written in TypeScript. These code examples can be a good resource for JavaScript developers; you just need to remove the TypeScript-specific parts of the code.

TypeScript snippets often use the newer ECMAScript `import` and `export` keywords to import objects from other modules and to declare the objects to be made available outside the current module. Node.js has just begun supporting these keywords in its latest releases. Depending on the version of Node.js you're using, you might rewrite imports and exports to use the older syntax.

Imports can be replaced with calls to the `require()` function.

TypeScript

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

```
import { Bucket, BucketPolicy } from '@aws-cdk/aws-s3';
```

JavaScript

```
const cdk = require('@aws-cdk/core');
const { Bucket, BucketPolicy } = require('@aws-cdk/aws-s3');
```

Exports can be assigned to the `module.exports` object.

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
An alternative to using the old-style imports and exports is to use the `esm` module.

Once you've got the imports and exports sorted, you can dig into the actual code. You may run into these commonly-used TypeScript features:

- Type annotations
- Interface definitions
- Type conversions/casts
- Access modifiers

Type annotations may be provided for variables, class members, function parameters, and function return types. For variables, parameters, and members, types are specified by following the identifier with a colon and the type. Function return values follow the function signature and consist of a colon and the type.

To convert type-annotated code to JavaScript, remove the colon and the type. Class members must have some value in JavaScript; set them to `undefined` if they only have a type annotation in TypeScript.

TypeScript

```
var encrypted: boolean = true;
```

```
class myStack extends core.Stack {
    bucket: s3.Bucket;
    // ...
}

function makeEnv(account: string, region: string) : object {
    // ...
}
```

JavaScript

```
var encrypted = true;

class myStack extends core.Stack {
    bucket = undefined;
    // ...
}

function makeEnv(account, region) {
    // ...
}
```

In TypeScript, interfaces are used to give bundles of required and optional properties, and their types, a name. You can then use the interface name as a type annotation. TypeScript will make sure that the object you use as, for example, an argument to a function has the required properties of the right types.

```
interface myFuncProps {
    code: lambda.Code,
    handler?: string
}
```

JavaScript does not have an interface feature, so once you've removed the type annotations, delete the interface declarations entirely.

When a function or method returns a general-purpose type (such as `object`), but you want to treat that value as a more specific child type to access properties or methods that are not part of the more general type's interface, TypeScript lets you *cast* the value using `as` followed by a type or interface name. JavaScript doesn't support (or need) this, so simply remove `as` and the following identifier. A less-common cast syntax is to use a type name in brackets, `<LikeThis>`; these casts, too, must be removed.

Finally, TypeScript supports the access modifiers `public`, `protected`, and `private` for members of classes. All class members in JavaScript are public. Simply remove these modifiers wherever you see them.

Knowing how to identify and remove these TypeScript features goes a long way toward adapting short TypeScript snippets to JavaScript. But it may be impractical to convert longer TypeScript examples in this fashion, since they are more likely to use other TypeScript features. For these situations, we recommend Babel with the TypeScript plug-in. Babel won't complain if code uses an undefined variable, for example, as `tsc` would. If it is syntactically valid, then with few exceptions, Babel can translate it to JavaScript. This makes Babel particularly valuable for converting snippets that may not be runnable on their own.

Migrating to TypeScript

Many JavaScript developers move to TypeScript as their projects get larger and more complex. TypeScript is a superset of JavaScript—all JavaScript code is valid TypeScript code, so no changes to your code are required—and it is also a supported AWS CDK language. Type annotations and other TypeScript features are optional and can be added to your AWS CDK app as you find value in them. TypeScript also

gives you early access to new JavaScript features, such as optional chaining and nullish coalescing, before they're finalized—and without requiring that you upgrade Node.js.

TypeScript's "shape-based" interfaces, which define bundles of required and optional properties (and their types) within an object, allow common mistakes to be caught while you're writing the code, and make it easier for your IDE to provide robust autocomplete and other real-time coding advice.

Coding in TypeScript does involve an additional step: compiling your app with the TypeScript compiler, `tsc`. For typical AWS CDK apps, compilation requires a few seconds at most.

The easiest way to migrate an existing JavaScript AWS CDK app to TypeScript is to create a new TypeScript project using `cdk init app --language typescript`, then copy your source files (and any other necessary files, such as assets like AWS Lambda function source code) to the new project. Rename your JavaScript files to end in `.ts` and begin developing in TypeScript.

Working with the AWS CDK in Python

Python is a fully-supported client language for the AWS CDK and is considered stable. Working with the AWS CDK in Python uses familiar tools, including the standard Python implementation (CPython), virtual environments with `virtualenv`, and the Python package installer `pip`. The modules comprising the AWS Construct Library are distributed via pypi.org. The Python version of the AWS CDK even uses Python-style identifiers (for example, `snake_case` method names).

You can use any editor or IDE. Many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has good support for Python via an official extension. The IDLE editor included with Python will suffice to get started. The Python modules for the AWS CDK do have type hints, which are useful for a linting tool or an IDE that supports type validation.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

Python AWS CDK applications require Python 3.6 or later. If you don't already have it installed, download a compatible version for your platform at python.org. If you run Linux, your system may have come with a compatible version, or you may install it using your distro's package manager (`yum`, `apt`, etc.). Mac users may be interested in Homebrew, a Linux-style package manager for macOS.

The Python package installer, `pip`, and virtual environment manager, `virtualenv`, are also required. Windows installations of compatible Python versions include these tools. On Linux, `pip` and `virtualenv` may be provided as separate packages in your package manager. Alternatively, you may install them with the following commands:

```
python -m ensurepip --upgrade
python -m pip install --upgrade pip
python -m pip install --upgrade virtualenv
```

If you encounter a permission error, run the above commands with the `--user` flag so that the modules are installed in your user directory, or use `sudo` to obtain the permissions to install the modules system-wide.

> **Note**
> It is common for Linux distros to use the executable name `python3` for Python 3.x, and have `python` refer to a Python 2.x installation. Some distros have an optional package you can install that makes the `python` command refer to Python 3. Failing that, you can adjust the command used to run your application by editing `cdk.json` in the project's main directory.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language python
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

After initializing the project, activate the project's virtual environment. This allows the project's dependencies to be installed locally in the project folder, instead of globally.

```
source .venv/bin/activate
```

> **Note**
> You may recognize this as the Mac/Linux command to activate a virtual environment. The Python templates include a batch file, `source.bat`, that allows the same command to be used on Windows. The traditional Windows command, `.venv\Scripts\activate.bat`, works, too. If you initialized your AWS CDK project using CDK Toolkit v1.70.0 or earlier, your virtual environment is in the `.env` directory instead of `.venv`.

After activating your virtual environment, install the app's standard dependencies:

```
python -m pip install -r requirements.txt
```

> **Important**
> Activate the project's virtual environment whenever you start working on it. Otherwise, you won't have access to the modules installed there, and modules you install will go in the Python global module directory (or will result in a permission error).

Managing AWS Construct Library modules

Use the Python package installer, **pip**, to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. **pip** also installs the dependencies for those modules automatically. If your system does not recognize **pip** as a standalone command, invoke **pip** as a Python module, like this:

```
python -m pip PIP-COMMAND
```

The AWS CDK core module is named `aws-cdk.core`. AWS Construct Library modules are named like `aws-cdk.`*`SERVICE-NAME`*. The service name includes an *aws* prefix. If you're unsure of a module's name, search for it at PyPI. For example, the command below installs the modules for Amazon S3 and AWS Lambda.

```
python -m pip install aws-cdk.aws-s3 aws-cdk.aws-lambda
```

Some services' Construct Library support is in more than one module. For example, besides the `aws-cdk.aws-route53` module, there are three additional Amazon Route 53 modules, named `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

The names used for importing AWS Construct Library modules into your Python code are similar to their package names. Simply replace the hyphens with underscores.

```
import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lambda_
```

We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Generally, import individual classes from `aws_cdk.core`.

```
from aws_cdk.core import App, Construct
```

- If you need many classes from the core module, you may use a namespace alias of `cdk` instead of importing individual classes. Avoid doing both.

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

- Generally, import AWS Construct Libraries using short namespace aliases.

```
import aws_cdk.aws_s3 as s3
```

After installing a module, update your project's `requirements.txt` file, which lists your project's dependencies. It is best to do this manually rather than using `pip freeze`. `pip freeze` captures the current versions of all modules installed in your Python virtual environment, which can be useful when bundling up a project to be run elsewhere.

Usually, though, your `requirements.txt` should list only top-level dependencies (modules that your app depends on directly) and not the dependencies of those modules. This strategy makes updating your dependencies simpler. Here is what your `requirements.txt` file might look like if you have installed the Amazon S3 and AWS Lambda modules as shown earlier.

```
aws-cdk.aws-s3==X.YY.ZZ
aws-cdk.aws-lambda==X.YY.ZZ
```

You can edit `requirements.txt` to allow upgrades; simply replace the `==` preceding a version number with `~=` to allow upgrades to a higher compatible version, or remove the version requirement entirely to specify the latest available version of the module.

With `requirements.txt` edited appropriately to allow upgrades, issue this command to upgrade your project's installed modules at any time:

```
pip install --upgrade -r requirements.txt
```

Important
All AWS Construct Library modules used in your project must be the same version.

AWS CDK idioms in Python

Language conflicts

In Python, `lambda` is a language keyword, so you cannot use it as a name for the AWS Lambda construct library module or Lambda functions. The Python convention for such conflicts is to use a trailing underscore, as in `lambda_`, in the variable name.

By convention, the second argument to AWS CDK constructs is named `id`. When writing your own stacks and constructs, calling a parameter `id` "shadows" the Python built-in function `id()`, which returns an object's unique identifier. This function isn't used very often, but if you should happen to need it in your construct, rename the argument, for example `id_`, or else call the built-in function as `__builtins__.id()`.

Arguments and properties

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

scope and *id* should always be passed as positional arguments, not keyword arguments, because their names change if the construct accepts a property named *scope* or *id*.

In Python, props are expressed as keyword arguments. If an argument contains nested data structures, these are expressed using a class which takes its own keyword arguments at instantiation. The same pattern is applied to other method calls that take a structured argument.

For example, in a Amazon S3 bucket's `add_lifecycle_rule` method, the `transitions` property is a list of `Transition` instances.

```
bucket.add_lifecycle_rule(
  transitions=[
    Transition(
      storage_class=StorageClass.GLACIER,
      transition_after=Duration.days(10)
    )
  ]
)
```

When extending a class or overriding a method, you may want to accept additional arguments for your own purposes that are not understood by the parent class. In this case you should accept the arguments you don't care about using the `**kwargs` idiom, and use keyword-only arguments to accept the arguments you're interested in. When calling the parent's constructor or the overridden method, pass only the arguments it is expecting (often just `**kwargs`). Passing arguments that the parent class or method doesn't expect results in an error.

```
class MyConstruct(Construct):
    def __init__(self, id, *, MyProperty=42, **kwargs):
        super().__init__(self, id, **kwargs)
        # ...
```

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. This won't cause any technical issues for users of your construct or method (since your property isn't passed "up the chain," the parent class or overridden method will simply use a default value) but it may cause confusion. You can avoid this potential problem by naming your properties so they clearly belong to your construct. If there are many new properties, bundle them into an appropriately-named class and pass it as a single keyword argument.

Missing values

The AWS CDK uses `None` to represent missing or undefined values. When working with `**kwargs`, use the dictionary's `get()` method to provide a default value if a property is not provided. Avoid using `kwargs[...]`, as this raises `KeyError` for missing values.

```
encrypted = kwargs.get("encrypted")         # None if no property "encrypted" exists
```

```
encrypted = kwargs.get("encrypted", False)  # specify default of False if property is
 missing
```

Some AWS CDK methods (such as `tryGetContext()` to get a runtime context value) may return `None`, which you will need to check explicitly.

Using interfaces

Python doesn't have an interface feature as some other languages do, though it does have abstract base classes, which are similar. (If you're not familiar with interfaces, Wikipedia has a good introduction.) TypeScript, the language in which the AWS CDK is implemented, does provide interfaces, and constructs and other AWS CDK objects often require an object that adheres to a particular interface, rather than inheriting from a particular class. So the AWS CDK provides its own interface feature as part of the JSII layer.

To indicate that a class implements a particular interface, you can use the `@jsii.implements` decorator:

```
from aws_cdk.core import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
    def visit(self, node: IConstruct) -> None:
        print("Visited", node.node.path)
```

Type pitfalls

Python uses dynamic typing, where variables may refer to a value of any type. Parameters and return values may be annotated with types, but these are "hints" and are not enforced. This means that in Python, it is easy to pass the incorrect type of value to a AWS CDK construct. Instead of getting a type error during build, as you would from a statically-typed language, you may instead get a runtime error when the JSII layer (which translates between Python and the AWS CDK's TypeScript core) is unable to deal with the unexpected type.

In our experience, the type errors Python programmers make tend to fall into these categories.

- Passing a single value where a construct expects a container (Python list or dictionary) or vice versa.
- Passing a value of a type associated with a Level 1 (`CfnXxxxxx`) construct to a higher-level construct, or vice versa.

The AWS CDK Python modules do include type annotations, so you can use tools that support them to help with types. If you are not using an IDE that supports these, such as PyCharm, you might want to call the MyPy type validator as a step in your build process. There are also runtime type checkers that can improve error messages for type-related errors.

Synthesizing and deploying

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 279).

Working with the AWS CDK in Java

Java is a fully-supported client platform for the AWS CDK and is considered stable. You can develop AWS CDK applications in Java using familiar tools, including the JDK (Oracle's, or an OpenJDK distribution such as Amazon Corretto) and Apache Maven. The modules comprising the AWS Construct Library are distributed via the Maven Central Repository.

You can use any text editor, or a Java IDE that can read Maven projects, to work on your AWS CDK apps. We provide Eclipse hints in this Guide, but IntelliJ IDEA, NetBeans, and other IDEs can import Maven projects and will work fine for developing AWS CDK applications in Java.

It is possible to write AWS CDK applications in JVM-hosted languages other than Java (for example, Kotlin, Groovy, Clojure, or Scala), but we are unable to provide support for these languages.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

Java AWS CDK applications require Java 8 (v1.8) or later. We recommend Amazon Corretto, but you can use any OpenJDK distribution or Oracle's JDK. You will also need Apache Maven 3.5 or later. You can also use tools such as Gradle, but the application skeletons generated by the AWS CDK Toolkit are Maven projects.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language java
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

The resulting project includes a reference to the `software.amazon.awscdk.core` Maven package. It and its dependencies are automatically installed by Maven.

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

Managing AWS Construct Library modules

Use Maven to install AWS Construct Library packages, which are in the group `software.amazon.awscdk` and named with a short version (no AWS or Amazon prefix) of their service's name. For example, the Maven artifact ID for Amazon S3 is `s3`. Search the Maven Central Repository to find the names of all AWS CDK and AWS Construct Module libraries.

> **Note**
> The Java edition of the CDK API Reference also shows the package names.

Some services' AWS Construct Library support is in more than one module. For example, Amazon Route 53 has the three modules in addition to the main `software.amazon.awscdk.route53` module, named `route53-patterns`, `route53resolver`, and `route53-targets`.

The AWS CDK's core module, which you'll need in most AWS CDK apps, is imported in Java code as `software.amazon.awscdk.core`. Modules for the various services in the AWS Construct Library live under `software.amazon.awscdk.services` and are named similarly to their Maven package name. For example, the Amazon S3 module's namespace is `software.amazon.awscdk.services.s3`.

We recommend writing a separate Java `import` statement for each AWS Construct Library class you use in each of your Java source files, and avoiding wildcard imports. You can always use a type's fully-qualified name (including its namespace) without an `import` statement.

> **Important**
> All AWS Construct Library modules used in your project must be the same version.

Specify the modules that your application depends on by editing `pom.xml` and adding a new `<dependency>` element in the `<dependencies>` container. For example, the following `<dependency>` element specifies the Amazon S3 construct library module:

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

> **Tip**
> If you use a Java IDE, it probably has features for managing Maven dependencies. We recommend editing `pom.xml` directly, however, unless you are absolutely sure the IDE's functionality matches what you'd do by hand.

The default `pom.xml` defines the variable `cdk.version` to be the version of the AWS CDK that created the project. You can easily update the version required by updating the value of this variable, which keeps all AWS Construct Library module versions in sync.

```
<cdk.version>1.XX.Y</cdk.version>
```

This value can be any valid Maven version specifier. For example, `[1.XX.Y,2.0)` indicates that any version between the current version 1.XX.Y (inclusive) and 2.0 (exclusive), may be installed. However, to avoid mismatched versions, we recommend using a fixed version like 1.XX and updating it when moving a new AWS CDK release.

AWS CDK idioms in Java

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In Java, props are expressed using the Builder pattern. Each construct type has a corresponding props type; for example, the `Bucket` construct (which represents an Amazon S3 bucket) takes as its props an instance of `BucketProps`.

The `BucketProps` class (like every AWS Construct Library props class) has an inner class called `Builder`. The `BucketProps.Builder` type offers methods to set the various properties of a `BucketProps` instance. Each method returns the `Builder` instance, so the method calls can be chained to set multiple properties. At the end of the chain, you call `build()` to actually produce the `BucketProps` object.

```
Bucket bucket = new Bucket(this, "MyBucket", new BucketProps.Builder()
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build());
```

Constructs, and other classes that take a props-like object as their final argument, offer a shortcut. The class has a `Builder` of its own that instantiates it and its props object in one step. This way, you don't need to explicitly instantiate (for example) both `BucketProps` and a `Bucket`—and you don't need an import for the props type.

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build();
```

When deriving your own construct from an existing construct, you may want to accept additional properties. We recommend that you follow these builder patterns. However, this isn't as simple as subclassing a construct class. You must provide the moving parts of the two new `Builder` classes yourself. You may prefer to simply have your construct accept one or more additional arguments. You should provide additional constructors when an argument is optional.

Generic structures

In some places, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's `BuildSpec.fromObject()` method.) In Java, these objects are represented as `java.util.Map<String, Object>`. In cases where the values are all strings, you can use `Map<String, String>`. It is convenient to use double braces to define `HashMaps`.

```
new HashMap<String, String>() {{
    put("base-directory", "dist");
    put("files", "LambdaStack.template.json");
}};
```

> **Note**
> The double-brace notation (which technically declares an anonymous inner class) is sometimes considered an anti-pattern. However, its disadvantages are not very relevant to using it in CDK apps. It is a reasonable substitute for what would be object or dictionary literals in other languages.

JavaScript arrays are represented as `List<Object>` or `List<String>` in Java. The method `java.util.Arrays.asList` is convenient for defining short `ArrayLists`.

```
String[] cmds = Arrays.asList("cd lambda", "npm install", "npm install typescript")
```

Missing values

In Java, missing values in AWS CDK objects such as props are represented by `null`. You must explicitly test any value that could be `null` to make sure it contains a value before doing anything with it. Java does not have "syntactic sugar" to help handle null values as some other languages do. You may find Apache ObjectUtil's defaultIfNull and firstNonNull useful in some situations. Alternatively, write your own static helper methods to make it easier to handle potentially null values and make your code more readable.

Building, synthesizing, and deploying

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and to run tests. You can do this in your IDE (for example, press Control-B in Eclipse) or by issuing `mvn compile` at a command prompt while in your project's root directory.

Run any tests you've written by running `mvn test` at a command prompt.

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 279).

Working with the AWS CDK in C#

.NET is a fully-supported client platform for the AWS CDK and is considered stable. C# is the main .NET language for which we provide examples and support. You can choose to write AWS CDK applications

in other .NET languages, such as Visual Basic or F#, but AWS offers limited support for using these languages with the CDK.

You can develop AWS CDK applications in C# using familiar tools including Visual Studio, Visual Studio Code, the `dotnet` command, and the NuGet package manager. The modules comprising the AWS Construct Library are distributed via nuget.org.

We suggest using Visual Studio 2019 (any edition) on Windows to develop AWS CDK apps in C#.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

C# AWS CDK applications require .NET Core v3.1 or later, available here.

The .NET Standard toolchain includes `dotnet`, a command-line tool for building and running .NET applications and managing NuGet packages. Even if you work mainly in Visual Studio, this command can be useful for batch operations and for installing AWS Construct Library packages.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language csharp
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

The resulting project includes a reference to the `Amazon.CDK` NuGet package. It and its dependencies are installed automatically by NuGet.

Managing AWS Construct Library modules

The .NET ecosystem uses the NuGet package manager. AWS Construct Library modules are named like `Amazon.CDK.AWS.`*`SERVICE-NAME`* where the service name is a short name without an AWS or Amazon prefix. For example, the NuGet package name for the Amazon S3 module is `Amazon.CDK.AWS.S3`. If you can't find a package you want, search Nuget.org.

> **Note**
> The .NET edition of the CDK API Reference also shows the package names.

Some services' AWS Construct Library support is in more than one module. For example, Amazon Route 53 has the three modules in addition to the main `Amazon.CDK.AWS.Route53` module, named `Route53.Patterns`, `Route53.Resolver`, and `Route53.Targets`.

The AWS CDK's core module, which you'll need in most AWS CDK apps, is imported in C# code as `Amazon.CDK`. Modules for the various services in the AWS Construct Library live under `Amazon.CDK.AWS` and are named the same as their NuGet package name. For example, the Amazon S3 module's namespace is `Amazon.CDK.AWS.S3`.

We recommend writing a single C# `using` directive for each AWS Construct Library module you use in each of your C# source files. You may find it convenient to use an alias for a namespace or type to help resolve name conflicts. You can always use a type's fully-qualfiied name (including its namespace) without a `using` statement.

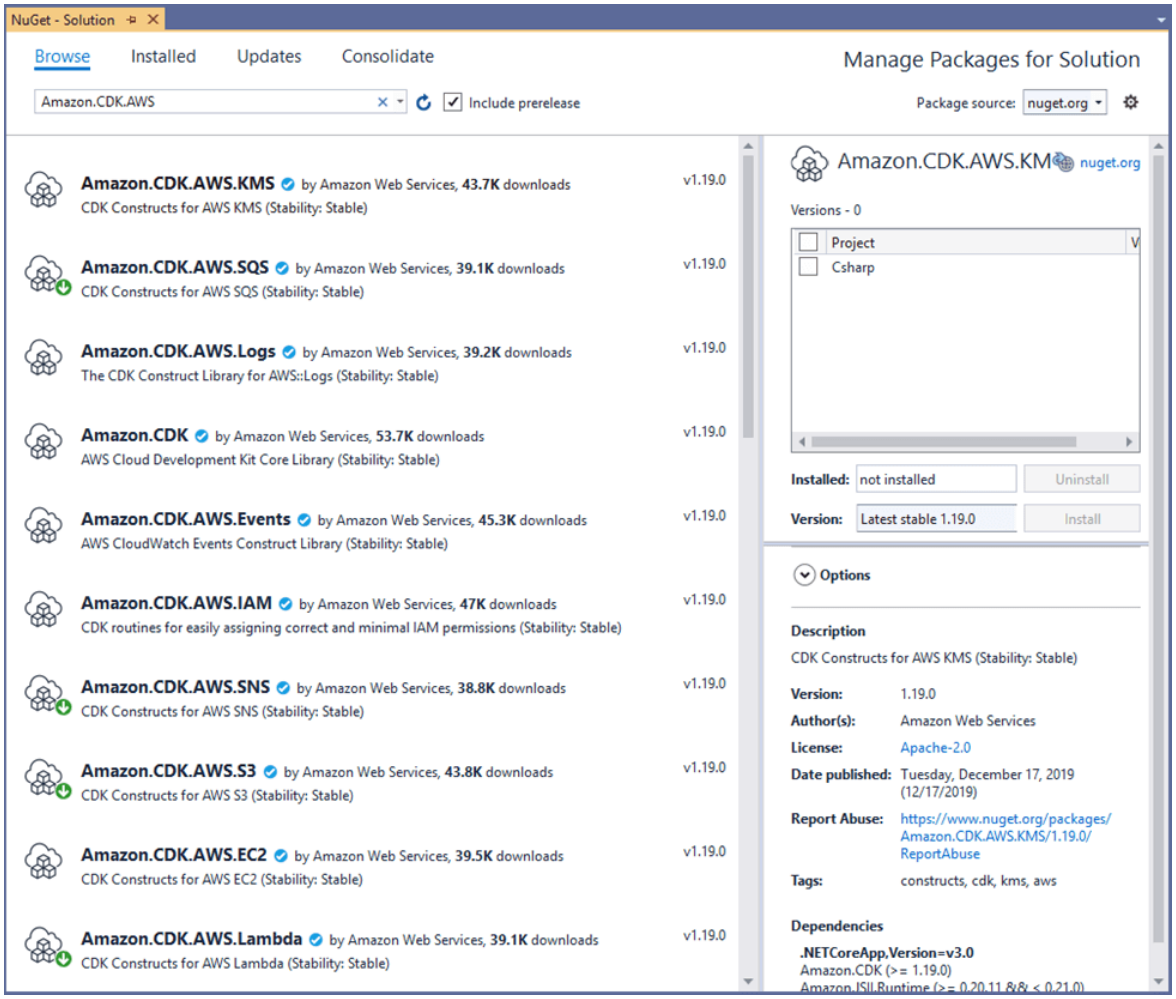
Important
All AWS Construct Library modules used in your project must be the same version.

NuGet has four standard, mostly-equivalent interfaces; you can use the one that suits your needs and working style. You can also use compatible tools, such as Paket or MyGet.

The Visual Studio NuGet GUI

Visual Studio's NuGet tools are accessible from **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**. Use the **Browse** tab to find the AWS Construct Library packages you want to install. You can choose the desired version, including pre-release versions (mark the **Include prerelease** checkbox) and add them to any of the open projects.

Note
All AWS Construct Library modules deemed "experimental" (see the section called "Versioning" (p. 199)) are flagged as pre-release in NuGet.



Look on the **Updates** page to install new versions of your packages.

The NuGet console

The NuGet console is a PowerShell-based interface to NuGet that works in the context of a Visual Studio project. You can open it in Visual Studio by choosing **Tools** > **NuGet Package Manager** > **Package**

Manager Console. For more information about using this tool, see Install and Manage Packages with the Package Manager Console in Visual Studio.

The `dotnet` command

The `dotnet` command is the primary command-line tool for working with Visual Studio C# projects. You can invoke it from any Windows command prompt. Among its many capabilities, `dotnet` can add NuGet dependencies to a Visual Studio project.

Assuming you're in the same directory as the Visual Studio project (`.csproj`) file, issue a command like the following to install a package.

```
dotnet add package Amazon.CDK.AWS.S3
```

You may issue the command from another directory by including the path to the project file, or to the directory that contains it, after the `add` keyword. The following example assumes that you are in your AWS CDK project's main directory.

```
dotnet add src/PROJECT-DIR package Amazon.CDK.AWS.S3
```

To install a specific version of a package, include the `-v` flag and the desired version. AWS Construct Library modules that are deemed "experimental" (see the section called “Versioning” (p. 199)) are flagged as pre-release in NuGet, and must be installed using an explicit version number.

```
dotnet add package Amazon.CDK.AWS.S3 -v VERSION-NUMBER
```

To update a package, issue the same `dotnet add` command you used to install it. If you do not specify a version number, the latest version is installed. For experimental modules, again, you must specify an explicit version number.

For more information about managing packages using the `dotnet` command, see Install and Manage Packages Using the dotnet CLI.

The `nuget` command

The `nuget` command line tool can install and update NuGet packages. However, it requires your Visual Studio project to be set up differently from the way `cdk init` sets up projects. (Technical details: `nuget` works with `Packages.config` projects, while `cdk init` creates a newer-style `PackageReference` project.)

We do not recommend the use of the `nuget` tool with AWS CDK projects created by `cdk init`. If you are using another type of project, and want to use `nuget`, see the NuGet CLI Reference.

AWS CDK idioms in C#

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In C#, props are expressed using a props type. In idiomatic C# fashion, we can use an object initializer to set the various properties. Here we're creating an Amazon S3 bucket using the `Bucket` construct; its corresponding props type is `BucketProps`.

```
var bucket = new Bucket(this, "MyBucket", new BucketProps {
    Versioned = true
});
```

Tip
Add the package `Amazon.JSII.Analyzers` to your project to get required-values checking in your props definitions inside Visual Studio.

When extending a class or overriding a method, you may want to accept additional props for your own purposes that are not understood by the parent class. To do this, subclass the appropriate props type and add the new attributes.

```
// 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 MyBucket class
var bucket = new MyBucket(this, "MyBucket", new MimeBucketProps {
    Versioned = true,
    MimeType = "image/jpeg"
});
```

When calling the parent class's initializer or overridden method, you can generally pass the props you received. The new type is compatible with its parent, and extra props you added are ignored.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. This won't cause any technical issues using your construct or method (since your property isn't passed "up the chain," the parent class or overridden method will simply use a default value) but it may cause confusion for your construct's users. You can avoid this potential problem by naming your properties so they clearly belong to your construct. If there are many new properties, bundle them into an appropriately-named class and pass them as a single property.

Generic structures

In some places, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's `BuildSpec.fromObject()` method.) In C#, these objects are represented as `System.Collections.Generic.Dictionary<String, Object>`. In cases where the values are all strings, you can use `Dictionary<String, String>`. JavaScript arrays are represented as `object[]` or `string[]` in C#.

Missing values

In C#, missing values in AWS CDK objects such as props are represented by `null`. The null-conditional member access operator `?.` and the null coalescing operator `??` are convenient for working with these values.

```
// 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";
```

Building, synthesizing, and deploying

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and run tests. You can do this by pressing F6 in Visual Studio or by issuing `dotnet build src` from the command line, where `src` is the directory in your project directory that contains the Visual Studio Solution (`.sln`) file.

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called "AWS CDK Toolkit" (p. 279).

Working with the AWS CDK in Go

The Go language binding for the AWS CDK is now available as a Developer Preview. It is not suitable for production use and may undergo significant changes before being designated stable. To follow development, see the Project Board on GitHub. Please report any issues you encounter.

Unlike the other languages the CDK supports, Go is not a traditional object-oriented programming language. Go uses composition where other languages often leverage inheritance. We have tried to employ idiomatic Go approaches as much as possible, but there are places where the CDK charts its own path.

This topic explains the ins and outs of working with the AWS CDK in Go. See the announcement blog post for a walkthrough of a simple Go project for the AWS CDK.

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 25).

The Go bindings for the AWS CDK use the standard Go toolchain, v1.16 or later. You can use the editor of your choice.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory.

```
mkdir my-project
cd my-project
cdk init app --language go
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.

The resulting project includes a reference to the AWS CDK Go module, `github.com/aws/aws-cdk-go/awscdk`, in `go.mod`. The CDK and its dependencies are automatically installed when you build your app.

Managing AWS Construct Library modules

In most AWS CDK documentation and examples, the word "module" is used primarily to refer to AWS Construct Library modules, one or more per AWS service, which differs from idiomatic Go usage of the term. The CDK Construct Library is provided in one Go module with the individual Construct Library modules, which support the various AWS services, provided as Go packages within that module.

Some services' AWS Construct Library support is in more than one Construct Library module (Go package). For example, Amazon Route 53 has three Construct Library modules in addition to the main `awsroute53` package, named `awsroute53patterns`, `awsroute53resolver`, and `awsroute53targets`.

The AWS CDK's core package, which you'll need in most AWS CDK apps, is imported in Go code as `github.com/aws/aws-cdk-go/awscdk`. Packages for the various services in the AWS Construct Library live under `github.com/aws/aws-cdk-go/awscdk`. For example, the Amazon S3 module's namespace is `github.com/aws/aws-cdk-go/awscdk/awss3`.

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

Once you have imported the Construct Library modules (Go packages) for the services you want to use in your app, you access constructs in that module using, for example, `awss3.Bucket`.

AWS CDK idioms in Go

Field and method names

Field and method names use camel casing (`likeThis`) in TypeScript, the CDK's language of origin. In Go, these follow Go conventions, so are Pascal-cased (`LikeThis`).

Missing values and pointer conversion

In Go, missing values in AWS CDK objects such as property bundles are represented by `nil`. Go doesn't have nullable types; the only type that can contain `nil` is a pointer. To allow values to be optional, then, all CDK properties, arguments, and return values are pointers, even for primitive types. This applies

to required values as well as optional ones, so if a required value later becomes optional, no breaking change in type is needed.

When passing literal values or expressions, you can use the following helper functions to create pointers to the values.

- `jsii.String`
- `jsii.Number`
- `jsii.Bool`
- `jsii.Time`

For consistency, we recommend that you use pointers similarly when defining your own constructs, even though it may seem more convenient to, for example, receive your construct's `id` as a string rather than a pointer to a string.

When dealing with optional AWS CDK values, including primitive values as well as complex types, you should explicitly test pointers to make sure they are not `nil` before doing anything with it. Go does not have "syntactic sugar" to help handle empty or missing values as some other languages do. However, required values in property bundles and similar structures are guaranteed to exist (construction fails otherwise), so these values need not be `nil`-checked.

Constructs and Props

Constructs, which represent one or more AWS resources and their associated attributes, are represented in Go as interfaces. For example, `awss3.Bucket` is an interface. Every construct has a factory function, such as `awss3.NewBucket`, to return a struct that implements the corresponding interface.

All factory functions take three arguments: the `scope` in which the construct is being defined (its parent in the construct tree), an `id`, and `props`, a bundle of key/value pairs that the construct uses to configure the resources it creates. The "bundle of attributes" pattern is also used elsewhere in the AWS CDK.

In Go, props are represented by a specific struct type for each construct. For example, an `awss3.Bucket` takes a props argument of type `awss3.BucketProps`. Use a struct literal to write props arguments.

```
var bucket = awss3.NewBucket(stack, jsii.String("MyBucket"), &awss3.BucketProps{
    Versioned: jsii.Bool(true),
})
```

Generic structures

In some places, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's `BuildSpec.fromObject()` method.) In Go, these objects are represented as slices and an empty interface, respectively.

The CDK provides variadic helper functions such as `jsii.Strings` for building slices containing primitive types.

```
jsii.Strings("One", "Two", "Three")
```

Developing custom constructs

In Go, it is usually more straightforward to write a new construct than to extend an existing one. First, define a new struct type, anonymously embedding one or more existing types if extension-like semantics are desired. Write methods for any new functionality you're adding and the fields necessary to hold the

data they need. Define a props interface if your construct needs one. Finally, write a factory function `NewMyConstruct()` to return an instance of your construct.

If you are simply changing some default values on an existing construct or adding a simple behavior at instantiation, you don't need all that plumbing. Instead, write a factory function that calls the factory function of the construct you're "extending." In other CDK languages, for example, you might create a `TypedBucket` construct that enforces the type of objects in an Amazon S3 bucket by overriding the `s3.Bucket` type and, in your new type's initializer, adding a bucket policy that allows only specified filename extensions to be added to the bucket. In Go, it is easier to simply write a `NewTypedBucket` that returns an `s3.Bucket` (instantiated using `s3.NewBucket`) to which you have added an appropriate bucket policy. No new construct type is necessary because the functionality is already available in the standard bucket construct; the new "construct" just provides a simpler way to configure it.

Building, synthesizing, and deploying

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and to run tests. You can do this by issuing `go build` at a command prompt while in your project's root directory.

Run any tests you've written by running `go test` at a command prompt.

The stacks (p. 85) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.

You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

```
cdk synth                 # app defines single stack
cdk deploy Happy Grumpy   # app defines two or more stacks; two are deployed
```

You may also use the wildcards * (any number of characters) and ? (any single character) to identify stacks by pattern. When using wildcards, enclose the pattern in quotes. Otherwise, the shell may try to expand it to the names of files in the current directory before they are passed to the AWS CDK Toolkit.

```
cdk synth "Stack?"    # Stack1, StackA, etc.
cdk deploy "*Stack"   # PipeStack, LambdaStack, etc.
```

Tip
You don't need to explicitly synthesize stacks before deploying them; `cdk deploy` performs this step for you to make sure your latest code gets deployed.

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 279).

Migrating to AWS CDK v2

Version 2 of the AWS Cloud Development Kit (CDK) is designed to make writing infrastructure as code in your preferred programming language even easier.

As a developer preview, CDK v2 is not intended for production use and may be subject to breaking API changes before it is finalized. We are eager to hear your feedback on how CDK v2 works for you, and your suggestions for making it better, via GitHub Issues.

Changes in AWS CDK v2

The main changes in CDK v2 are:

- AWS CDK v2 consolidates the AWS Construct Library into a single package; developers no longer need to install one or more individual packages for each AWS service. This change also eliminates the requirement to keep all CDK library versions in sync.
- Experimental L2/L3 constructs are no longer included in the AWS Construct Library; they will instead be distributed separately with an appropriate semantic version number. Constructs will move into the main AWS Construct Library only after being designated stable, permitting the Construct Library to adhere to strict semantic versioning going forward. We are still finalizing the details of how experimental constructs will be distributed for CDK v2, so initially, CDK v2 does not provide experimental constructs.
- The core `Construct` class has been extracted from the AWS CDK into a separate library, along with related types, to support efforts to apply the CDK programming model to other domains. If you are writing your own constructs or using related APIs, you must declare the `construct` module as a dependency and make minor changes to your imports. If you are using advanced features, such as hooking into the CDK app lifecycle, more changes are needed. See the RFC for full details.
- Deprecated properties, methods, and types in AWS CDK v1.x and its Construct Library have been removed from the CDK v2 API. In most supported languages, these APIs produce warnings under v1.x, so you may have already migrated to the replacement APIs. A complete list of deprecated APIs in CDK v1.x is available on GitHub.
- Behavior that was gated by feature flags in AWS CDK v1.x is enabled by default in CDK v2, and the old feature flags are no longer needed or, in most cases, supported.
- CDK v2 requires that the environments you deploy into be boostrapped using the modern bootstrap stack; the legacy stack is no longer supported. CDK v2 furthermore requires a new version of the modern stack. Simply re-bootstrap the affected environments to upgrade them. It is not necessary to set any feature flags or environment variables to specify the modern bootstrap stack.

  Important
  The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list, which by default will extend permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack (p. 178) with policies and trusted accounts you are comfortable with.

Aside from this topic, the AWS CDK Developer Guide describes CDK v1.x. Most of the information in the Guide still applies in CDK v2, or can be adapted with only minor changes. A v2 Developer Guide will be available at General Availability (GA) of CDK v2. A version of the AWS CDK API Reference is available for CDK v2.

Prerequisites

Most requirements for AWS CDK v2 are the same as for AWS CDK v1.x. See the section called "Prerequisites" (p. 10). Additional requirements are listed here.

- For TypeScript developers, TypeScript 3.8 or later is required.
- A new version of the CDK Toolkit is required for use with CDK v2. To install it, issue `npm install -g aws-cdk@next`.

Migrating to AWS CDK v2

To migrate your app to AWS CDK v2, first update the feature flags in `cdk.json`. Then update your app's dependencies and imports as necessary for the programming language it is written in.

Updating `cdk.json`

Remove all feature flags from `cdk.json`. You can add one or more of the three flags listed below, set to `false`, if your app relies on these specific AWS CDK v1.x behaviors. Use the `cdk diff` command to inspect the changes to your synthesized template to see if any of these are needed.

`@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId`

If your application uses multiple Amazon API Gateway API keys and associates them to usage plans

`@aws-cdk/aws-rds:lowercaseDbIdentifier`

If your application uses Amazon RDS database instance or database clusters, and explicitly specifies the identifier for these

`@aws-cdk/core:stackRelativeExports`

If your application uses multiple stacks and you refer to resources from one stack in another, this determines whether absolute or relative path is used to construct AWS CloudFormation exports

The syntax for reverting these flags in `cdk.json` is shown here.

```
{
  "context": {
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/core:stackRelativeExports": false,
  }
}
```

Updating dependencies and imports

Update your app's dependencies in the appropriate configuration file, then install the new dependencies. Finally, update the imports in your code.

TypeScript

For AWS CDK applications, update `package.json` as follows. Note that `@aws-cdk/assert` must be updated to the same version as `aws-cdk-lib`.

```
{
```

```
  "dependencies": {
    "aws-cdk-lib": "^2.0.0-rc.1",
    "@aws-cdk/assert": "^2.0.0-rc.1",
    "constructs": "^10.0.0"
  }
}
```

For construct libraries, establish the lowest version of `aws-cdk-lib` you require for your application (2.0.0-rc.1 here) and update `package.json` as follows. Note that `aws-cdk-lib` appears both as a peer dependency and a dev dependency.

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0-rc.1",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.0.0-rc.1",
    "@aws-cdk/assert": "^2.0.0-rc.1",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

Install the new dependencies by running `npm install` or `yarn install`.

Change your app's imports to import `Construct` from the new `constructs` module, core types such as `App` and `Stack` from the top level of `aws-cdk-lib`, and AWS Construct Library modules from namespaces under `aws-cdk-lib`.

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';
import { aws_s3 as s3 } from 'aws-cdk-lib';
```

JavaScript

Update `package.json` as follows. Note that `@aws-cdk/assert` must be updated to the same version as `aws-cdk-lib`.

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0-rc.1",
    "@aws-cdk/assert": "^2.0.0-rc.1",
    "constructs": "^10.0.0"
  }
}
```

Install the new dependencies by running `npm install` or `yarn install`.

Change your app's imports to import `Construct` from the new `constructs` module, core types such as `App` and `Stack` from the top level of `aws-cdk-lib`, and AWS Construct Library modules from namespaces under `aws-cdk-lib`.

```
const { Construct } = require('constructs');
const { App, Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib').aws_s3;
```

Python

Update the `install_requires` definition in `setup.py` to look like this.

```
install_requires=[
     "aws-cdk-lib>=2.0.0rc1",
     "constructs>=10.0.0",
     # ...
],
```

Install the new dependencies with `pip install -r requirements.txt`.

Tip
Uninstall any other versions of AWS CDK modules already installed in your app's virtual environment using `pip uninstall`.

Change your app's imports to import `Construct` from the new `constructs` module, core types such as `App` and `Stack` from the top level of `aws_cdk`, and AWS Construct Library modules from namespaces under `aws_cdk`.

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

# ...

class MyConstruct(Construct):
    # ...

class MyStack(Stack):
    # ...

s3.Bucket(...)
```

Alternatively, you can import `constructs` and `aws_cdk`, then refer to classes through their namespace.

```
import constructs
import aws_cdk as cdk
from aws_cdk import aws_s3 as s3

# ...

class MyConstruct(constructs.Construct):
    # ...

class MyStack(cdk.Stack):
    # ...

s3.Bucket(...)
```

Java

In `pom.xml`, remove all `software.amazon.awscdk` dependencies and replace them with these dependencies on `software.constructs` and `software.amazon.awscdk`.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0-rc.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
```

```
</dependency>
```

Install the new dependencies by running `mvn package`.

Change your code to import `Construct` from the new `software.constructs` library, core classes like `Stack` and `App` from `software.amazon.awscdk`, and service constructs from `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;
```

C#

The most straightforward way to upgrade the dependencies of a C# CDK application is to edit the `.csproj` file manually. Remove all `Amazon.CDK.*` package references and replace them with references to the `Amazon.CDK.Lib` and `Constructs` packages.

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0-rc.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

Install the new dependencies by running `dotnet restore`.

Change the imports in your source files as follows.

```
using Constructs;         // for Construct class
using Amazon.CDK;         // for core classes like App and Stack
using Amazon.CDK.AWS.S3;  // for service constructs like Bucket
```

Troubleshooting

Unexpected infrastructure changes

Before deploying your app, use `cdk diff` to check for unexpected changes to its resources. Such changes are typically not caused by upgrading to AWS CDK v2, but are the result of deprecated behavior that was previously hidden by feature flags. This is a symptom of upgrading from a version of CDK older than about 1.85.x; you'd have the same problem upgrading to the latest v1.x release. You can usually resolve this by upgrading your app to the latest v1.x release, revising your code as necessary, deploying, and then upgrading to v2.

If your upgraded app ends up undeployable after the two-stage upgrade, please report the issue.

Typescript `'from' expected` or `';' expected` error in imports

Upgrade to TypeScript 3.8 or later.

Translating TypeScript AWS CDK code to other languages

TypeScript was the first language supported for developing AWS CDK applications, and for that reason, there is a substantial amount of example CDK code written in TypeScript. If you are developing in another language, it may be useful to compare how AWS CDK code is implemented in TypeScript and your language of choice, so you can, with a little effort, make use of these examples.

For more details on working with the AWS CDK in its supported programming languages, see:

- the section called “In TypeScript” (p. 26)
- the section called “In JavaScript” (p. 30)
- the section called “In Python” (p. 36)
- the section called “In Java” (p. 41)
- the section called “In C#” (p. 44)

Importing a module

TypeScript/JavaScript

TypeScript supports importing either an entire module, or individual objects from a module.

```
// Import entire module as s3 into current namespace
import * as s3 from '@aws-cdk/aws-s3';

// Import an entire module using Node.js require() (import * as s3 generally preferred)
const s3 = require('@aws-cdk/aws-s3');

// TypeScript version of require() (again, import * as s3 generally preferred)
import s3 = require('@aws-cdk/aws-s3');

// Now use s3 to access the S3 types
const bucket = s3.Bucket(...);

// Selective import of s3.Bucket into current namespace
import { Bucket } from '@aws-cdk/aws-s3';

// Selective import of Bucket and EventType into current namespace
import { Bucket, EventType } from '@aws-cdk/aws-s3';

// Now use Bucket to instantiate an S3 bucket
const bucket = Bucket(...);
```

Python

Like TypeScript, Python supports namespaced module imports and selective imports. Module names in Python look like **aws_cdk.***xxx*, where *xxx* represents an AWS service name, such as **s3** for Amazon S3 (we'll use Amazon S3 for our examples).

```
# 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

# Selective import of Bucket and EventType into current namespace
from aws_cdk.s3 import Bucket, EventType

# Bucket can now be used to instantiate a bucket
bucket = Bucket(...)
```

Java

Java's imports work differently from TypeScript's. Each import statement imports either a single class name from a given package, or all classes defined in that package (using `*`). Classes may be accessed using either the class name by itself if it has been imported, or the *qualified* class name including its package.

Packages are named like `software.amazon.awscdk.services.xxx` for AWS Construct Library packages (the core module is `software.amazon.awscdk.core`). The Maven group ID for AWS CDK packages is `software.amazon.awscdk`.

```
// 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 = new Bucket(...);

// 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 =
     new software.amazon.awscdk.services.s3.Bucket(...);
```

C#

In C#, you import types with the `using` directive. There are two styles, which give you access either all the types in the specified namespace using their plain names, or let you refer to the namespace itself using an alias.

Packages are named like `Amazon.CDK.AWS.xxx` for AWS Construct Library packages (the core module is `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(...)
```

Instantiating a construct

AWS CDK construct classes have the same name in all supported languages. Most languages use the `new` keyword to instantiate a class (Python is the only one that doesn't). Also, in most languages, the keyword `this` refers to the current instance. Python, again, is the exception (it uses `self` by convention). You should pass a reference to the current instance as the `scope` parameter to every construct you create.

The third argument to a AWS CDK construct is `props`, an object containing attributes needed to build the construct. This argument may be optional, but when it is required, the supported languages handle it in idiomatic ways. The names of the attributes are also adapted to the language's standard naming patterns.

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 doesn't use a `new` keyword when instantiating a class. The properties argument is represented using keyword arguments, and the arguments are named using `snake_case`.

If a props value is itself a bundle of attributes, it is represented by a class named after the property, which accepts keyword arguments for the sub-properties.

In Python, the current instance is passed to methods as the first argument, which is named `self` by convention.

```
# 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

In Java, the props argument is represented by a class named `XxxxProps` (for example, `BucketProps` for the `Bucket` construct's props). You build the props argument using a builder pattern.

Each `XxxxProps` class has a builder, and there is also a convenient builder for each construct that builds the props and the construct in one step, as shown here.

Props are named the same as in TypeScript, using `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#

In C#, props are specified using an object initializer to a class named `XxxxProps` (for example, `BucketProps` for the `Bucket` construct's props).

Props are named similarly to TypeScript, except using `PascalCase`.

It is convenient to use the `var` keyword when instantiating a construct, so you don't need to type the class name twice. However, your local code style guide may vary.

```
// 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"
                      }});
```

Accessing members

It is common to refer to attributes or properties of constructs and other AWS CDK classes and use these values as, for examples, inputs to build other constructs. The naming differences described above for methods apply. Furthermore, in Java, it is not possible to access members directly; instead, a getter method is provided.

TypeScript/JavaScript

Names are `camelCase`.

```
bucket.bucketArn
```

Python

Names are `snake_case`.

```
bucket.bucket_arn
```

Java

A getter method is provided for each property; these names are `camelCase`.

```
bucket.getBucketArn()
```

C#

Names are `PascalCase`.

```
bucket.BucketArn
```

Enum constants

Enum constants are scoped to a class, and have uppercase names with underscores in all languages (sometimes referred to as `SCREAMING_SNAKE_CASE`). Since class names also use the same casing in all supported languages, qualified enum names are also the same.

```
s3.BucketEncryption.KMS_MANAGED
```

Object interfaces

The AWS CDK uses TypeScript object interfaces to indicate that a class implements an expected set of methods and properties. You can recognize an object interface because its name starts with `I`. A concrete class indicates the interface(s) it implements using the `implements` keyword.

TypeScript/JavaScript

Note
JavaScript doesn't have an interface feature. You can ignore the `implements` keyword and the class names following it.

```
import { IAspect, IConstruct } from '@aws-cdk/core';

class MyAspect implements IAspect {
  public visit(node: IConstruct) {
    console.log('Visited', node.node.path);
  }
}
```

Python

Python doesn't have an interface feature. However, for the AWS CDK you can indicate interface implementation by decorating your class with `@jsii.implements(interface)`.

```
from aws_cdk.core 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.core.IAspect;
import software.amazon.awscdk.core.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}");
    }
}
```

Concepts

This topic describes some of the concepts (the why and how) behind the AWS CDK. It also discusses the AWS Construct Library.

AWS CDK apps are composed of building blocks known as Constructs (p. 64), which are composed together to form stacks and apps.

Constructs

Constructs are the basic building blocks of AWS CDK apps. A construct represents a "cloud component" and encapsulates everything AWS CloudFormation needs to create the component.

A construct can represent a single resource, such as an Amazon Simple Storage Service (Amazon S3) bucket, or it can represent a higher-level component consisting of multiple AWS resources. Examples of such components include a worker queue with its associated compute capacity, a cron job with monitoring resources and a dashboard, or even an entire app spanning multiple AWS accounts and regions.

AWS Construct library

The AWS CDK includes the AWS Construct Library, which contains constructs representing AWS resources.

This library includes constructs that represent all the resources available on AWS. For example, the `s3.Bucket` class represents an Amazon S3 bucket, and the `dynamodb.Table` class represents an Amazon DynamoDB table.

There are three different levels of constructs in this library, beginning with low-level constructs, which we call *CFN Resources* (or L1, short for "level 1") or Cfn (short for CloudFormation) resources. These constructs directly represent all resources available in AWS CloudFormation. CFN Resources are periodically generated from the AWS CloudFormation Resource Specification. They are named **Cfn***Xyz*, where *Xyz* is name of the resource. For example, CfnBucket represents the AWS::S3::Bucket AWS CloudFormation resource. When you use Cfn resources, you must explicitly configure all resource properties, which requires a complete understanding of the details of the underlying AWS CloudFormation resource model.

The next level of constructs, L2, also represent AWS resources, but with a higher-level, intent-based API. They provide similar functionality, but provide the defaults, boilerplate, and glue logic you'd be writing yourself with a CFN Resource construct. AWS constructs offer convenient defaults and reduce the need to know all the details about the AWS resources they represent, while providing convenience methods that make it simpler to work with the resource. For example, the s3.Bucket class represents an Amazon S3 bucket with additional properties and methods, such as bucket.addLifeCycleRule(), which adds a lifecycle rule to the bucket.

Finally, the AWS Construct Library includes even higher-level constructs, which we call *patterns*. These constructs are designed to help you complete common tasks in AWS, often involving multiple kinds of resources. For example, the aws-ecs-patterns.ApplicationLoadBalancedFargateService construct represents an architecture that includes an AWS Fargate container cluster employing an Application Load Balancer (ALB). The aws-apigateway.LambdaRestApi construct represents an Amazon API Gateway API that's backed by an AWS Lambda function.

For more information about how to navigate the library and discover constructs that can help you build your apps, see the API Reference.

Composition

Composition is the key pattern for defining higher-level abstractions through constructs. A high-level construct can be composed from any number of lower-level constructs, and in turn, those could be composed from even lower-level constructs, which eventually are composed from AWS resources. From a bottom-up perspective, you use constructs to organize the individual AWS resources you want to deploy using whatever abstractions are convenient for your purpose, with as many layers as you need.

Composition lets you define reusable components and share them like any other code. For example, a team can define a construct that implements the company's best practice for a DynamoDB table with backup, global replication, auto-scaling, and monitoring, and share it with other teams in their organization, or publicly. Teams can now use this construct as they would any other library package in their preferred programming language to define their tables and comply with their team's best practices. When the library is updated, developers will get access to the new version's bug fixes and improvements through the workflows they already have for their other types of code.

Initialization

Constructs are implemented in classes that extend the `Construct` base class. You define a construct by instantiating the class. All constructs take three parameters when they are initialized:

- **Scope** – The construct within which this construct is defined. You should usually pass `this` for the scope, because it represents the current scope in which you are defining the construct.
- **id** – An identifier (p. 117) that must be unique within this scope. The identifier serves as a namespace for everything that's defined within the current construct and is used to allocate unique identities such as resource names (p. 102) and AWS CloudFormation logical IDs.
- **Props** – A set of properties or keyword arguments, depending upon the language, that define the construct's initial configuration. In most cases, constructs provide sensible defaults, and if all props elements are optional, you can leave out the **props** parameter completely.

Identifiers need only be unique within a scope. This lets you instantiate and reuse constructs without concern for the constructs and identifiers they might contain, and enables composing constructs into higher level abstractions. In addition, scopes make it possible to refer to groups of constructs all at once, for example for tagging or for specifying where the constructs will be deployed.

Apps and stacks

We call your CDK application an *app*, which is represented by the AWS CDK class App. The following example defines an app with a single stack that contains a single Amazon S3 bucket with versioning enabled:

TypeScript

```
import { App, Stack, StackProps } from '@aws-cdk/core';
import * as s3 from '@aws-cdk/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/core');
const s3 = require('@aws-cdk/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.core import App, Stack
from aws_cdk import aws_s3 as s3

class HelloCdkStack(core.Stack):

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

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

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

Java

```
import software.amazon.awscdk.core.*;
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;
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 });
        }
    }
}
```

As you can see, you need a scope within which to define your bucket. Since resources eventually need to be deployed as part of a AWS CloudFormation stack into an *AWS* *environment (p. 92)*, which covers a specific AWS account and AWS region. AWS constructs, such as `s3.Bucket`, must be defined within the scope of a Stack.

Stacks in AWS CDK apps extend the **Stack** base class, as shown in the previous example. This is a common pattern when creating a stack within your AWS CDK app: extend the **Stack** class, define a constructor that accepts **scope**, **id**, and **props**, and invoke the base class constructor via `super` with the received **scope**, **id**, and **props**, as shown in the following example.

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(core.Stack):

    def __init__(self, scope: core.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)
    {
        //...
    }
}
```

Using L1 constructs

Once you have defined a stack, you can populate it with resources by instantiating constructs. First, we'll do it with an L1 construct.

L1 constructs are exactly the resources defined by AWS CloudFormation—no more, no less. You must provide the resource's required configuration yourself. Here, for example, is how to create an Amazon S3 bucket using the `CfnBucket` class. (You'll see a similar definition using the `Bucket` class in the next section.)

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"
});
```

In Python, Java, and C#, L1 construct properties that aren't simple Booleans, strings, numbers, or containers are represented by types defined as inner classes of the L1 construct. For example, the optional property `corsConfiguration` of a `CfnBucket` requires a wrapper of type `CfnBucket.CorsConfigurationProperty`. Here we are defining `corsConfiguration` on a `CfnBucket` instance.

TypeScript

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

JavaScript

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

Python

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

Java

```
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#

```
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" },
            }
        }
    }
});
```

Important
You can't use L2 property types with L1 constructs, or vice versa. When working with L1 constructs, always use the types defined inside the L1 construct you're using. Do not use types from other L1 constructs (some may have the same name, but they are not the same type). Some of our language-specific API references currently have errors in the paths to L1 property types, or don't document these classes at all. We hope to fix this soon. In the meantime, just remember that such types are always inner classes of the L1 construct they are used with.

Using L2 constructs

The following example defines an Amazon S3 bucket by creating an instance of the `Bucket` class, an L2 construct.

TypeScript

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

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

JavaScript

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

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

Python

```
from aws_cdk import 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
});
```

The AWS Construct Library includes constructs that represent many AWS resources.

Note
`MyFirstBucket` is not the name of the bucket that AWS CloudFormation creates. It is a logical identifier given to the new construct. See Physical Names for details.

Configuration

Most constructs accept `props` as their third argument (or in Python, keyword arguments), a name/value collection that defines the construct's configuration. The following example defines a bucket with AWS Key Management Service (AWS KMS) encryption and static website hosting enabled. Since it does not explicitly specify an encryption key, the `Bucket` construct defines a new `kms.Key` and associates it with the bucket.

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"
});
```

AWS constructs are designed around the concept of "sensible defaults." Most constructs have a minimal required configuration, enabling you to quickly get started while also providing full control over the configuration when you need it.

Interacting with constructs

Constructs are classes that extend the base Construct class. After you instantiate a construct, the construct object exposes a set of methods and properties that enable you to interact with the construct and pass it around as a reference to other parts of the system. The AWS CDK framework doesn't put any restrictions on the APIs of constructs; authors can define any API they wish. However, the AWS constructs that are included with the AWS Construct Library, such as `s3.Bucket`, follow guidelines and common patterns in order to provide a consistent experience across all AWS resources.

For example, almost all AWS constructs have a set of grant (p. 152) methods that you can use to grant AWS Identity and Access Management (IAM) permissions on that construct to a principal. The following example grants the IAM group `data-science` permission to read from the Amazon S3 bucket `raw-data`.

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);
```

Another common pattern is for AWS constructs to set one of the resource's attributes, such as its Amazon Resource Name (ARN), name, or URL from data supplied elsewhere. For example, the following code defines an AWS Lambda function and associates it with an Amazon Simple Queue Service (Amazon SQS) queue through the queue's URL in an environment variable.

TypeScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_10_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_10_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_10_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(new HashMap<String, String>() {{
                        put("QUEUE_URL", jobsQueue.getQueueUrl());
```

```
                }}).build();
```

C#

```
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
{
    Runtime = Runtime.NODEJS_10_X,
    Handler = "index.handler",
    Code = Code.FromAsset(@".\create-job-lambda-code"),
    Environment = new Dictionary<string, string>
    {
        ["QUEUE_URL"] = jobsQueue.QueueUrl
    }
});
```

For information about the most common API patterns in the AWS Construct Library, see the section called “Resources” (p. 98).

Writing your own constructs

In addition to using existing constructs like `s3.Bucket`, you can also write your own constructs, and then anyone can use them in their apps. All constructs are equal in the AWS CDK. An AWS CDK construct such as `s3.Bucket` or `sns.Topic` behaves the same as a construct imported from a third-party library that someone published via NPM or Maven or PyPI—or to your company's internal package repository.

To declare a new construct, create a class that extends the Construct base class, then follow the pattern for initializer arguments.

For example, you could declare a construct that represents an Amazon S3 bucket which sends an Amazon Simple Notification Service (Amazon SNS) notification every time someone uploads a file into it:

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(core.Construct):

    def __init__(self, scope: core.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
        });
    }
}
```

The `NotifyingBucket` constructor has a typical construct signature: `scope`, `id`, and `props`. The last argument, `props`, is optional (gets the default value `{}`) because all props are optional. (The base `Construct` class does not take a `props`argument.) You could define an instance of this construct in your app without `props`, for example:

TypeScript

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

JavaScript

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

Python

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

Java

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

C#

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

Or you could use `props` (in Java, an additional parameter) to specify the path prefix to filter on, for example:

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"
});
```

Typically, you would also want to expose some properties or methods on your constructs. For example, it's not very useful to have a topic hidden behind your construct, because it wouldn't be possible for users of your construct to subscribe to it. Adding a `topic` property allows consumers to access the inner topic, as shown in the following example:

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(core.Construct):

    def __init__(self, scope: core.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 Bucket {

    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
        });
    }
}
```

Now, consumers can subscribe to the topic, for example:

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));
```

The construct tree

As we've already seen, in AWS CDK apps, you define constructs "inside" other constructs using the `scope` argument passed to every construct. In this way, an AWS CDK app defines a hierarchy of constructs known as the *construct tree*.

The root of this tree is your app—that is, an instance of the `App` class. Within the app, you instantiate one or more stacks. Within stacks, you instantiate either AWS CloudFormation resources or higher-level constructs, which may themselves instantiate resources or other constructs, and so on down the tree.

Constructs are *always* explicitly defined within the scope of another construct, so there is never any doubt about the relationships between constructs. Almost always, you should pass `this` (in Python, `self`) as the scope, indicating that the new construct is a child of the current construct. The intended pattern is that you derive your construct from `core.Construct`, then instantiate the constructs it uses in its constructor.

Passing the scope explicitly allows each construct to add itself to the tree, with this behavior entirely contained within the `Construct` base class. It works the same way in every language supported by the AWS CDK and does not require introspection or other "magic."

> **Important**
> Technically, it's possible to pass some scope other than `this` when instantiating a construct, which allows you to add constructs anywhere in the tree, even in another stack entirely. For example, you could write a mixin-style function that adds constructs to a scope passed in as an argument. The practical difficulty here is that you can't easily ensure that the IDs you choose for your constructs are unique within someone else's scope. The practice also makes your code more difficult to understand, maintain, and reuse. It is virtually always better to find a way to express your intent without resorting to abusing the `scope` argument.

The AWS CDK uses the IDs of all constructs in the path from the tree's root to each child construct to generate the unique IDs required by AWS CloudFormation. This approach means that construct IDs need be unique only within their scope, rather than within the entire stack as in native AWS CloudFormation. It does, however, mean that if you move a construct to a different scope, its generated stack-unique ID will change, and AWS CloudFormation will no longer consider it the same resource.

The construct tree is separate from the constructs you define in your AWS CDK code, but it is accessible through any construct's `node` attribute, which is a reference to the node that represents that construct in the tree. Each node is a `ConstructNode` instance, the attributes of which provide access to the tree's root and to the node's parent scopes and children.

- `node.children` – The direct children of the construct.
- `node.id` – The identifier of the construct within its scope.

- `node.path` – The full path of the construct including the IDs of all of its parents.
- `node.root` – The root of the construct tree (the app).
- `node.scope` – The scope (parent) of the construct, or undefined if the node is the root.
- `node.scopes` – All parents of the construct, up to the root.
- `node.uniqueId` – The unique alphanumeric identifier for this construct within the tree (by default, generated from `node.path` and a hash).

The construct tree defines an implicit order in which constructs are synthesized to resources in the final AWS CloudFormation template. Where one resource must be created before another, AWS CloudFormation or the AWS Construct Library will generally infer the dependency and make sure the resources are created in the right order. You can also add an explicit dependency between two nodes using `node.addDependency()`; see Dependencies in the AWS CDK API Reference.

The AWS CDK provides a simple way to visit every node in the construct tree and perform an operation on each one. See the section called "Aspects" (p. 165).

Apps

As described in the section called "Constructs" (p. 64), to provision infrastructure resources, all constructs that represent AWS resources must be defined, directly or indirectly, within the scope of a Stack construct.

The following example declares a stack class named `MyFirstStack` that includes a single Amazon S3 bucket. However, this only declares a stack. You still need to define (also known as to instantiate) it in some scope to deploy it.

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");
    }
}
```

The app construct

To define the previous stack within the scope of an application, use the App construct. The following example app instantiates a `MyFirstStack` and produces the AWS CloudFormation template that the stack defined.

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();
```

The `App` construct doesn't require any initialization arguments, because it's the only construct that can be used as a root for the construct tree. You can now use the `App` instance as a scope for defining a single instance of your stack.

You can also define constructs within an App-derived class as follows.

TypeScript

```
class MyApp extends App {
  constructor() {
    new MyFirstStack(this, 'hello-cdk');
  }
}

new MyApp().synth();
```

JavaScript

```
class MyApp extends App {
  constructor() {
    new MyFirstStack(this, 'hello-cdk');
  }
}

new MyApp().synth();
```

Python

```
class MyApp(App):
     def __init__(self):
         MyFirstStack(self, "hello-cdk")

MyApp().synth()
```

Java

```
// MyApp.java
package com.myorg;

import software.amazon.awscdk.core.App;

public class MyApp extends App{
    public MyApp() {
        new MyFirstStack(this, "hello-cdk");
    }
}

// Main.java
package com.myorg;

public class Main {
    public static void main(String[] args) {
        new MyApp().synth();
    }
```

```
}
```

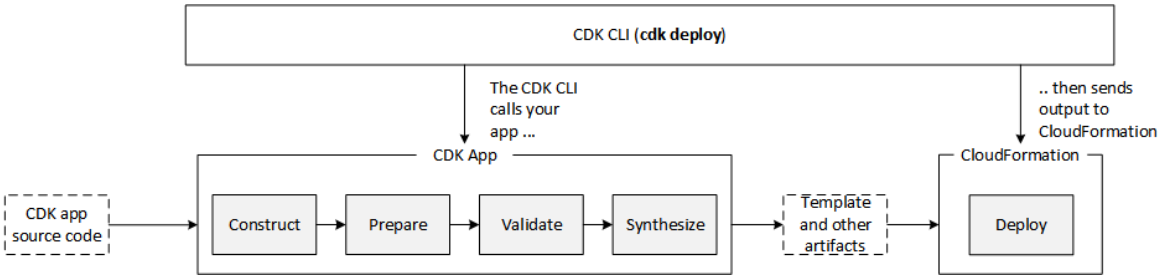
C#

```
public class MyApp : App
{
    public MyApp(AppProps props = null) : base(props)
    {
        new MyFirstStack(this, "hello-cdk");

    }
}

class Program
{
    static void Main(string[] args)
    {
        new MyApp().Synth();
    }
}
```

These two methods are equivalent.

App lifecycle

The following diagram shows the phases that the AWS CDK goes through when you call the **cdk deploy**. This command deploys the resources that your app defines.



An AWS CDK app goes through the following phases in its lifecycle.

1. Construction (or Initialization)

   Your code instantiates all of the defined constructs and then links them together. In this stage, all of the constructs (app, stacks, and their child constructs) are instantiated and the constructor chain is executed. Most of your app code is executed in this stage.

2. Preparation

   All constructs that have implemented the `prepare` method participate in a final round of modifications, to set up their final state. The preparation phase happens automatically. As a user, you don't see any feedback from this phase. It's rare to need to use the "prepare" hook, and generally not recommended. You should be very careful when mutating the construct tree during this phase, because the order of operations could impact behavior.

3. Validation

   All constructs that have implemented the `validate` method can validate themselves to ensure that they're in a state that will correctly deploy. You will get notified of any validation failures that happen during this phase. Generally, we recommend that you perform validation as soon as possible (usually as soon as you get some input) and throw exceptions as early as possible. Performing

validation early improves diagnosability as stack traces will be more accurate, and ensures that your code can continue to execute safely.

4. Synthesis

   This is the final stage of the execution of your AWS CDK app. It's triggered by a call to `app.synth()`, and it traverses the construct tree and invokes the `synthesize` method on all constructs. Constructs that implement `synthesize` can participate in synthesis and emit deployment artifacts to the resulting cloud assembly. These artifacts include AWS CloudFormation templates, AWS Lambda application bundles, file and Docker image assets, and other deployment artifacts. the section called “Cloud assemblies” (p. 84) describes the output of this phase. In most cases, you won't need to implement the `synthesize` method

5. Deployment

   In this phase, the AWS CDK Toolkit takes the deployment artifacts cloud assembly produced by the synthesis phase and deploys it to an AWS environment. It uploads assets to Amazon S3 and Amazon ECR, or wherever they need to go, and then starts an AWS CloudFormation deployment to deploy the application and create the resources.

By the time the AWS CloudFormation deployment phase (step 5) starts, your AWS CDK app has already finished and exited. This has the following implications:

- The AWS CDK app can't respond to events that happen during deployment, such as a resource being created or the whole deployment finishing. To run code during the deployment phase, you have to inject it into the AWS CloudFormation template as a custom resource (p. 174). For more information about adding a custom resource to your app, see the AWS CloudFormation module, or the custom-resource example.
- The AWS CDK app might have to work with values that can't be known at the time it runs. For example, if the AWS CDK app defines an Amazon S3 bucket with an automatically generated name, and you retrieve the `bucket.bucketName` (Python: `bucket_name`) attribute, that value is not the name of the deployed bucket. Instead, you get a `Token` value. To determine whether a particular value is available, call `cdk.isToken(value)` (Python: `is_token`). See the section called “Tokens” (p. 122) for details.

Cloud assemblies

The call to `app.synth()` is what tells the AWS CDK to synthesize a cloud assembly from an app. Typically you don't interact directly with cloud assemblies. They are files that include everything needed to deploy your app to a cloud environment. For example, it includes an AWS CloudFormation template for each stack in your app, and a copy of any file assets or Docker images that you reference in your app.

See the cloud assembly specification for details on how cloud assemblies are formatted.

To interact with the cloud assembly that your AWS CDK app creates, you typically use the AWS CDK Toolkit, a command-line tool. But any tool that can read the cloud assembly format can be used to deploy your app.

The CDK Toolkit needs to know how to execute your AWS CDK app. If you created the project from a template using the `cdk init` command, your app's `cdk.json` file includes an `app` key that specifies the necessary command for the language the app is written in. If your language requires compilation, the command line performs this step before running the app, so you can't forget to do it.

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"
}
```

If you did not create your project using the CDK Toolkit, or wish to override the command line given in `cdk.json`, you can use the **--app** option when issuing the `cdk` command.

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

The *`executable`* part of the command indicates the command that should be run to execute your CDK application. Use quotation marks as shown, since such commands contain spaces. The *`cdk-command`* is a subcommand like **synth** or **deploy** that tells the CDK Toolkit what you want to do with your app. Follow this with any additional options needed for that subcommand.

The CLI can also interact directly with an already-synthesized cloud assembly. To do that, just pass the directory in which the cloud assembly is stored in **--app**. The following example lists the stacks defined in the cloud assembly stored under `./my-cloud-assembly`.

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

Stacks

The unit of deployment in the AWS CDK is called a *stack*. All AWS resources defined within the scope of a stack, either directly or indirectly, are provisioned as a single unit.

Because AWS CDK stacks are implemented through AWS CloudFormation stacks, they have the same limitations as in AWS CloudFormation.

You can define any number of stacks in your AWS CDK app. Any instance of the `Stack` construct represents a stack, and can be either defined directly within the scope of the app, like the `MyFirstStack` example shown previously, or indirectly by any construct within the tree.

For example, the following code defines an AWS CDK app with two stacks.

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();
```

To list all the stacks in an AWS CDK app, run the **cdk ls** command, which for the previous AWS CDK app would have the following output.

```
stack1
stack2
```

When you run the **cdk synth** command for an app with multiple stacks, the cloud assembly includes a separate template for each stack instance. Even if the two stacks are instances of the same class, the AWS CDK emits them as two individual templates.

You can synthesize each template by specifying the stack name in the **cdk synth** command. The following example synthesizes the template for **stack1**.

```
cdk synth stack1
```

This approach is conceptually different from how AWS CloudFormation templates are normally used, where a template can be deployed multiple times and parameterized through AWS CloudFormation parameters. Although AWS CloudFormation parameters can be defined in the AWS CDK, they are generally discouraged because AWS CloudFormation parameters are resolved only during deployment. This means that you cannot determine their value in your code. For example, to conditionally include a resource in your app based on the value of a parameter, you must set up an AWS CloudFormation condition and tag the resource with this condition. Because the AWS CDK takes an approach where concrete templates are resolved at synthesis time, you can use an **if** statement to check the value to determine whether a resource should be defined or some behavior should be applied.

> **Note**
> The AWS CDK provides as much resolution as possible during synthesis time to enable idiomatic and natural usage of your programming language.

Like any other construct, stacks can be composed together into groups. The following code shows an example of a service that consists of three stacks: a control plane, a data plane, and monitoring stacks. The service construct is defined twice: once for the beta environment and once for the production environment.

TypeScript

```
import { App, Construct, Stack } from "@aws-cdk/core";

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 , Construct , Stack } = require("@aws-cdk/core");

// 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.core import App, Construct, Stack

# 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.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.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;

// 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();
    }
}
```

This AWS CDK app eventually consists of six stacks, three for each environment:

```
$ cdk ls

betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

The physical names of the AWS CloudFormation stacks are automatically determined by the AWS CDK based on the stack's construct path in the tree. By default, a stack's name is derived from the construct ID of the `Stack` object, but you can specify an explicit name using the `stackName` prop (in Python, `stack_name`), as follows.

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"
});
```

Stack API

The Stack object provides a rich API, including the following:

- `Stack.of(construct)` – A static method that returns the **Stack** in which a construct is defined. This is useful if you need to interact with a stack from within a reusable construct. The call fails if a stack cannot be found in scope.
- `stack.stackName` (Python: `stack_name`) – Returns the physical name of the stack. As mentioned previously, all AWS CDK stacks have a physical name that the AWS CDK can resolve during synthesis.
- `stack.region` and `stack.account` – Return the AWS Region and account, respectively, into which this stack will be deployed. These properties return either the account or Region explicitly specified when the stack was defined, or a string-encoded token that resolves to the AWS CloudFormation pseudo-parameters for account and Region to indicate that this stack is environment agnostic. See the section called “Environments” (p. 92) for information about how environments are determined for stacks.
- `stack.addDependency(stack)` (Python: `stack.add_dependency(stack)` – Can be used to explicitly define dependency order between two stacks. This order is respected by the **cdk deploy** command when deploying multiple stacks at once.
- `stack.tags` – Returns a TagManager that you can use to add or remove stack-level tags. This tag manager tags all resources within the stack, and also tags the stack itself when it's created through AWS CloudFormation.
- `stack.partition`, `stack.urlSuffix` (Python: `url_suffix`), `stack.stackId` (Python: `stack_id`), and `stack.notificationArn` (Python: `notification_arn`) – Return tokens that resolve to the respective AWS CloudFormation pseudo-parameters, such as `{ "Ref": "AWS::Partition" }`. These tokens are associated with the specific stack object so that the AWS CDK framework can identify cross-stack references.
- `stack.availabilityZones` (Python: `availability_zones`) – Returns the set of Availability Zones available in the environment in which this stack is deployed. For environment-agnostic stacks, this always returns an array with two Availability Zones, but for environment-specific stacks, the AWS CDK queries the environment and returns the exact set of Availability Zones available in the region you specified.
- `stack.parseArn(arn)` and `stack.formatArn(comps)` (Python: `parse_arn`, `format_arn`) – Can be used to work with Amazon Resource Names (ARNs).
- `stack.toJsonString(obj)` (Python: `to_json_string`) – Can be used to format an arbitrary object as a JSON string that can be embedded in an AWS CloudFormation template. The object can include tokens, attributes, and references, which are only resolved during deployment.
- `stack.templateOptions` (Python: `template_options`) – Enables you to specify AWS CloudFormation template options, such as Transform, Description, and Metadata, for your stack.

Nested stacks

The NestedStack construct offers a way around the AWS CloudFormation 500-resource limit for stacks. A nested stack counts as only one resource in the stack that contains it, but can itself contain up to 500 resources, including additional nested stacks.

The scope of a nested stack must be a `Stack` or `NestedStack` construct. The nested stack needn't be declared lexically inside its parent stack; it is necessary only to pass the parent stack as the first parameter (`scope`) when instantiating the nested stack. Aside from this restriction, defining constructs in a nested stack works exactly the same as in an ordinary stack.

At synthesis time, the nested stack is synthesized to its own AWS CloudFormation template, which is uploaded to the AWS CDK staging bucket at deployment. Nested stacks are bound to their parent stack and are not treated as independent deployment artifacts; they are not listed by `cdk list` nor can they be deployed by `cdk deploy`.

References between parent stacks and nested stacks are automatically translated to stack parameters and outputs in the generated AWS CloudFormation templates, as with any cross-stack reference (p. 101).

Warning
Changes in security posture are not displayed before deployment for nested stacks. This information is displayed only for top-level stacks.

Environments

Each `Stack` instance in your AWS CDK app is explicitly or implicitly associated with an environment (`env`). An environment is the target AWS account and region into which the stack is intended to be deployed.

Note
For all but the simplest deployments, you will need to bootstrap (p. 174) each environment you will deploy into. Deployment requires certain AWS resources to be available, and these resources are provisioned by bootstrapping.

If you don't specify an environment when you instantiate a stack, the stack is said to be *environment-agnostic*. AWS CloudFormation templates synthesized from such a stack will try to use deploy-time resolution on environment-related attributes such as `stack.account`, `stack.region`, and `stack.availabilityZones` (Python: `availability_zones`).

Tip
If you're using the standard AWS CDK development template, your stacks are instantiated in the same file where you instantiate the `App` object.

TypeScript

The file named after your project (for example, `hello-cdk.ts`) in your project's `bin` folder.

JavaScript

The file named after your project (for example, `hello-cdk.js`) in your project's `bin` folder.

Python

The file `app.py` in your project's main directory.

Java

The file named *`ProjectName`*`App.java`, for example `HelloCdkApp.java`, nested deep under the `src/main` directory.

C#

The file named `Program.cs` under `src\`*`ProjectName`*, for example `src\HelloCdk\Program.cs`.

In an environment-agnostic stack, any constructs that use availability zones will see two of them, allowing the stack to be deployed to any region.

When using **cdk deploy** to deploy environment-agnostic stacks, the AWS CDK CLI uses the specified AWS CLI profile (or the default profile, if none is specified) to determine where to deploy. The AWS CDK CLI follows a protocol similar to the AWS CLI to determine which AWS credentials to use when performing operations in your AWS account. See the section called "AWS CDK Toolkit" (p. 279) for details.

For production stacks, we recommend that you explicitly specify the environment for each stack in your app using the `env` property. The following example specifies different environments for its two different stacks.

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 = core.Environment(account="8373873873", region="eu-west-1")
env_USA = core.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 });
```

When you hard-code the target account and region as above, the stack will always be deployed to that specific account and region. To make the stack deployable to a different target, but to determine the target at synthesis time, your stack can use two environment variables provided by the AWS CDK CLI: `CDK_DEFAULT_ACCOUNT` and `CDK_DEFAULT_REGION`. These variables are set based on the AWS profile specified using the **--profile** option, or the default AWS profile if you don't specify one.

The following code fragment shows how to access the account and region passed from the AWS CDK CLI in your stack.

TypeScript

Access environment variables via Node's `process` object.

Note
You need the DefinitelyTyped module to use `process` in TypeScript. `cdk init` installs this module for you, but if you are working with a project created before it was added, or didn't set up your project using `cdk init`, install it manually.

```
npm install @types/node
```

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

JavaScript

Access environment variables via Node's `process` object.

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

Python

Use the `os` module's `environ` dictionary to access environment variables.

```
import os
MyDevStack(app, "dev", env=core.Environment(
    account=os.environ["CDK_DEFAULT_ACCOUNT"],
    region=os.environ["CDK_DEFAULT_REGION"]))
```

Java

Use `System.getenv()` to get the value of an environment variable.

```
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#

Use `System.Environment.GetEnvironmentVariable()` to get the value of an environment variable.

```
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() });
```

The AWS CDK distinguishes between not specifying the `env` property at all and specifying it using `CDK_DEFAULT_ACCOUNT` and `CDK_DEFAULT_REGION`. The former implies that the stack should synthesize an environment-agnostic template. Constructs that are defined in such a stack cannot use any information about their environment. For example, you can't write code like `if (stack.region === 'us-east-1')` or use framework facilities like Vpc.fromLookup (Python: `from_lookup`), which need to query your AWS account. These features do not work at all without an explicit environment specified; to use them, you must specify `env`.

When you pass in your environment using `CDK_DEFAULT_ACCOUNT` and `CDK_DEFAULT_REGION`, the stack will be deployed in the account and Region determined by the AWS CDK CLI at the time of synthesis. This allows environment-dependent code to work, but it also means that the synthesized template could be different based on the machine, user, or session under which it is synthesized. This behavior is often acceptable or even desirable during development, but it would probably be an anti-pattern for production use.

You can set `env` however you like, using any valid expression. For example, you might write your stack to support two additional environment variables to let you override the account and region at synthesis time. We'll call these `CDK_DEPLOY_ACCOUNT` and `CDK_DEPLOY_REGION` here, but you could name them anything you like, as they are not set by the AWS CDK. In the following stack's environment, we use our alternative environment variables if they're set, falling back to the default environment provided by the AWS CDK if they are not.

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=core.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() });
```

With your stack's environment declared this way, you can now write a short script or batch file like the following to set the variables from command line arguments, then call `cdk deploy`. Any arguments beyond the first two are passed through to `cdk deploy` and can be used to specify command-line options or stacks.

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
```

Save the script as `cdk-deploy-to.sh`, then execute `chmod +x cdk-deploy-to.sh` to make it executable.

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
}
```

The Windows version of the script uses PowerShell to provide the same functionality as the macOS/Linux version. It also contains instructions to allow it to be run as a batch file so it can be easily invoked from a command line. It should be saved as `cdk-deploy-to.bat`. The file `cdk-deploy-to.ps1` will be created when the batch file is invoked.

Then you can write additional scripts that call the "deploy-to" script to deploy to specific environments (even multiple environments per script):

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 %*
```

When deploying to multiple environments, consider whether you want to continue deploying to other environments after a deployment fails. The following example avoids deploying to the second production environment if the first doesn't succeed.

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 %*
```

Developers could still use the normal `cdk deploy` command to deploy to their own AWS environments for development.

Resources

As described in the section called “Constructs” (p. 64), the AWS CDK provides a rich class library of constructs, called *AWS constructs*, that represent all AWS resources. This section describes some common patterns and best practices for how to use these constructs.

Defining AWS resources in your CDK app is exactly like defining any other construct. You create an instance of the construct class, pass in the scope as the first argument, the logical ID of the construct, and a set of configuration properties (props). For example, here's how to create an Amazon SQS queue with KMS encryption using the sqs.Queue construct from the AWS Construct Library.

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
});
```

Some configuration props are optional, and in many cases have default values. In some cases, all props are optional, and the last argument can be omitted entirely.

Resource attributes

Most resources in the AWS Construct Library expose attributes, which are resolved at deployment time by AWS CloudFormation. Attributes are exposed in the form of properties on the resource classes with the type name as a prefix. The following example shows how to get the URL of an Amazon SQS queue using the `queueUrl` (Python: `queue_url`) property.

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

```
from 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
```

See the section called “Tokens” (p. 122) for information about how the AWS CDK encodes deploy-time attributes as strings.

Referencing resources

Many AWS CDK classes require properties that are AWS CDK resource objects (resources). To satisfy these requirements, you can refer to a resource in one of two ways:

- By passing the resource directly
- By passing the resource's unique identifier, which is typically an ARN, but it could also be an ID or a name

For example, an Amazon ECS service requires a reference to the cluster on which it runs; an Amazon CloudFront distribution requires a reference to the bucket containing source code.

If a construct property represents another AWS construct, its type is that of the interface type of that construct. For example, the Amazon ECS service takes a property `cluster` of type `ecs.ICluster`; the CloudFront distribution takes a property `sourceBucket` (Python: `source_bucket`) of type `s3.IBucket`.

Because every resource implements its corresponding interface, you can directly pass any resource object you're defining in the same AWS CDK app. The following example defines an Amazon ECS cluster and then uses it to define an Amazon ECS service.

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 });
```

Accessing resources in a different stack

You can access resources in a different stack, as long as they are in the same account and AWS Region. The following example defines the stack `stack1`, which defines an Amazon S3 bucket. Then it defines a second stack, `stack2`, which takes the bucket from `stack1` as a constructor property.

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.getBucket());
```

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 an argument "bucket"
var stack2 = new StackThatExpectsABucket(app, "Stack2", new StackProps { Env = prod,
    bucket = stack1.Bucket});
```

If the AWS CDK determines that the resource is in the same account and Region, but in a different stack, it automatically synthesizes AWS CloudFormation exports in the producing stack and an Fn::ImportValue in the consuming stack to transfer that information from one stack to the other.

Referencing a resource from one stack in a different stack creates a dependency between the two stacks,. Once this dependency is established, removing the use of the shared resource from the consuming stack can cause an unexpected deployment failure if the AWS CDK Toolkit deploys the producing stack before the consuming stack. This happens if there is another dependency between the two stacks, but it can also happen that the producing stack is chosen by the AWS CDK Toolkit to be deployed first. The AWS CloudFormation export is removed from the producing stack because it is no longer needed, but the exported resource is still being used in the consuming stack because its update has not yet been deployed, so deploying the producer stack fails.

To break this deadlock, remove the use of the shared resource from the consuming stack (which will remove the automatic export from the producing stack), then manually add the same export to the producing stack using exactly the same logical ID as the automatically-generated export. Remove the use of the shared resource in the consuming stack and deploy both stacks. Then remove the manual export (and the shared resource if it is no longer nededed), and deploy both stacks again. The stack's `exportValue()` method is a convenient way to create the manual export for this purpose (see the example in the linked method reference).

Physical names

The logical names of resources in AWS CloudFormation are different from the names of resources that are shown in the AWS Management Console after AWS CloudFormation has deployed the resources. The AWS CDK calls these final names *physical names*.

For example, AWS CloudFormation might create the Amazon S3 bucket with the logical ID **Stack2MyBucket4DD88B4F** from the previous example with the physical name **stack2mybucket4dd88b4f-iuv1rbv9z3to**.

You can specify a physical name when creating constructs that represent resources by using the property <resourceType>Name. The following example creates an Amazon S3 bucket with the physical name **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" });
```

Assigning physical names to resources has some disadvantages in AWS CloudFormation. Most importantly, any changes to deployed resources that require a resource replacement, such as changes to a resource's properties that are immutable after creation, will fail if a resource has a physical name assigned. If you end up in a state like that, the only solution is to delete the AWS CloudFormation stack, then deploy the AWS CDK app again. See the AWS CloudFormation documentation for details.

In some cases, such as when creating an AWS CDK app with cross-environment references, physical names are required for the AWS CDK to function correctly. In those cases, if you don't want to bother with coming up with a physical name yourself, you can let the AWS CDK name it for you by using the special value `PhysicalName.GENERATE_IF_NEEDED`, as follows.

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 });
```

Passing unique identifiers

Whenever possible, you should pass resources by reference, as described in the previous section. However, there are cases where you have no other choice but to refer to a resource by one of its attributes. For example, when you are using the low-level AWS CloudFormation resources, or need to expose resources to the runtime components of an AWS CDK application, such as when referring to Lambda functions through environment variables.

These identifiers are available as attributes on the resources, such as the following.

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

The Java AWS CDK binding uses getter methods for attributes.

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

C#

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

The following example shows how to pass a generated bucket name to an AWS Lambda function.

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(new HashMap<String, String>() {{
                put("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
    }
});
```

Importing existing external resources

Sometimes you already have a resource in your AWS account and want to use it in your AWS CDK app, for example, a resource that was defined through the console, an AWS SDK, directly with AWS CloudFormation, or in a different AWS CDK application. You can turn the resource's ARN (or another

identifying attribute, or group of attributes) into an AWS CDK object in the current stack by calling a static factory method on the resource's class.

The following example shows how to define a bucket based on an existing bucket with the ARN **arn:aws:s3:::my-bucket-name**, and a Amazon Virtual Private Cloud based on an existing VPC having a specific ID.

TypeScript

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

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

// Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde',
});
```

JavaScript

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

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

// Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde'
});
```

Python

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

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

# Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")
```

Java

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

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

// Construct a resource by giving attribute(s) (complex resources)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
        .vpcId("vpc-1234567890abcdef").build());
```

C#

```
// Construct a resource (bucket) just by its name (must be same account)
```

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

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

// Construct a resource by giving  attribute(s) (complex resources)
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
{
    VpcId = "vpc-1234567890abcdef"
});
```

Because the `ec2.Vpc` construct is complex, composed of many AWS resources, such as the VPC itself, subnets, security groups, and routing tables), it can be difficult to import those resources using attributes. To address this, the VPC construct contains a `fromLookup` method (Python: `from_lookup`) that uses a context method (p. 160) to resolve all the required attributes at synthesis time, and cache the values for future use in `cdk.context.json`.

You must provide attributes sufficient to uniquely identify a VPC in your AWS account. For example, there can only ever be one default VPC, so specifying that you want to import the VPC marked as the default is sufficient.

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 });
```

You can use the `tags` property to query by tag. Tags may be added to the VPC at the time of its creation using AWS CloudFormation or the AWS CDK, and they may be edited at any time after creation using the AWS Management Console, the AWS CLI, or an AWS SDK. In addition to any tags you have added yourself, the AWS CDK automatically adds the following tags to all VPCs it creates.

- *Name* – The name of the VPC.
- *aws-cdk:subnet-name* – The name of the subnet.
- *aws-cdk:subnet-type* – The type of the subnet: Public, Private, or 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(new HashMap<String, String> {{ put("aws-cdk:subnet-type", "Public"); }})
        .build());
```

C#

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

`Vpc.fromLookup()` works only in stacks that are defined with an explicit **account** and **region** in their `env` property. If the AWS CDK attempts to look up an Amazon VPC from an environment-agnostic stack (p. 90), the CLI does not know which environment to query to find the VPC.

Results of `Vpc.fromLookup()` are cached in the project's `cdk.context.json` file. Commit this file to version control if you will be deploying the stack in an environment that does not have access to the AWS account that defines the VPC, such as CDK Pipelines (p. 255).

Although you can use an imported resource anywhere, you cannot modify the imported resource. For example, calling `addToResourcePolicy` (Python: `add_to_resource_policy`) on an imported `s3.Bucket` does nothing.

Permission grants

AWS constructs make least-privilege permissions easy to achieve by offering simple, intent-based APIs to express permission requirements. Many AWS constructs offer grant methods that enable you to easily grant an entity, such as an IAM role or a user, permission to work with the resource without having to manually craft one or more IAM permission statements.

The following example creates the permissions to allow a Lambda function's execution role to read and write objects to a particular Amazon S3 bucket. If the Amazon S3 bucket is encrypted using an AWS KMS key, this method also grants the Lambda function's execution role permissions to decrypt using this key.

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)
{
    // ...
}
```

The grant methods return an `iam.Grant` object. Use the `success` attribute of the `Grant` object to determine whether the grant was effectively applied (for example, it may not have been applied on imported resources (p. 100)). You can also use the `assertSuccess` (Python: `assert_success`) method of the `Grant` object to enforce that the grant was successfully applied.

If a specific grant method isn't available for the particular use case, you can use a generic grant method to define a new grant with a specified list of actions.

The following example shows how to grant a Lambda function access to the Amazon DynamoDB `CreateBackup` action.

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");
```

Many resources, such as Lambda functions, require a role to be assumed when executing code. A configuration property enables you to specify an `iam.IRole`. If no role is specified, the function automatically creates a role specifically for this use. You can then use grant methods on the resources to add statements to the role.

The grant methods are built using lower-level APIs for handling with IAM policies. Policies are modeled as PolicyDocument objects. Add statements directly to roles (or a construct's attached role) using the `addToRolePolicy` method (Python: `add_to_role_policy`), or to a resource's policy (such as a `Bucket` policy) using the `addToResourcePolicy` (Python: `add_to_resource_policy`) method.

Metrics and alarms

Many resources emit CloudWatch metrics that can be used to set up monitoring dashboards and alarms. AWS constructs have metric methods that allow easy access to the metrics without having to look up the correct name to use.

The following example shows how to define an alarm when the `ApproximateNumberOfMessagesNotVisible` of an Amazon SQS queue exceeds 100.

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,
    // ..
});
```

If there is no method for a particular metric, you can use the general metric method to specify the metric name manually.

Metrics can also be added to CloudWatch dashboards. See CloudWatch.

Network traffic

In many cases, you must enable permissions on a network for an application to work, such as when the compute infrastructure needs to access the persistence layer. Resources that establish or listen for connections expose methods that enable traffic flows, including setting security group rules or network ACLs.

IConnectable resources have a `connections` property that is the gateway to network traffic rules configuration.

You enable data to flow on a given network path by using `allow` methods. The following example enables HTTPS connections to the web and incoming connections from the Amazon EC2 Auto Scaling group `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());
```

Certain resources have default ports associated with them, for example, the listener of a load balancer on the public port, and the ports on which the database engine accepts connections for instances of an Amazon RDS database. In such cases, you can enforce tight network control without having to manually specify the port by using the `allowDefaultPortFrom` and `allowToDefaultPort` methods (Python: `allow_default_port_from`, `allow_to_default_port`).

The following example shows how to enable connections from any IPV4 address, and a connection from an Auto Scaling group to access a database.

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");
```

Event handling

Some resources can act as event sources. Use the `addEventNotification` method (Python: `add_event_notification`) to register an event target to a particular event type emitted by the resource. In addition to this, `addXxxNotification` methods offer a simple way to register a handler for common event types.

The following example shows how to trigger a Lambda function when an object is added to an Amazon S3 bucket.

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));
```

Removal policies

Resources that maintain persistent data, such as databases and Amazon S3 buckets and even Amazon ECR registries, have a *removal policy* that indicates whether to delete persistent objects when the AWS CDK stack that contains them is destroyed. The values specifying the removal policy are available through the `RemovalPolicy` enumeration in the AWS CDK `core` module.

Note
Resources besides those that store data persistently may also have a `removalPolicy` that is used for a different purpose. For example, a Lambda function version uses a `removalPolicy` attribute to determine whether a given version is retained when a new version is deployed. These have different meanings and defaults compared to the removal policy on an Amazon S3 bucket or DynamoDB table.

Value	meaning
`RemovalPolicy.RETAIN`	Keep the contents of the resource when destroying the stack (default). The resource is orphaned from the stack and must be deleted manually. If you attempt to re-deploy the stack while the resource still exists, you will receive an error message due to a name conflict.
`RemovalPolicy.DESTROY`	The resource will be destroyed along with the stack.

AWS CloudFormation does not remove Amazon S3 buckets that contain files even if their removal policy is set to `DESTROY`. Attempting to do so is a AWS CloudFormation error. To have the AWS CDK delete all files from the bucket before destroying it, set the bucket's `autoDeleteObjects` property to `true`.

Following is an example of creating an Amazon S3 bucket with `RemovalPolicy` of `DESTROY` and `autoDeleteOjbects` set to `true`. .

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
    });
}
```

You can also apply a removal policy directly to the underlying AWS CloudFormation resource via the `applyRemovalPolicy()` method. This method is available on some stateful resources that do not have a `removalPolicy` property in their L2 resource's props, including AWS CloudFormation stacks, Amazon Cognito user pools, Amazon DocumentDB database instances, Amazon EC2 volumes, Amazon OpenSearch Service domains, Amazon FSx file systems, and Amazon SQS queues.

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
The AWS CDK's `RemovalPolicy` translates to AWS CloudFormation's `DeletionPolicy`, but the default in AWS CDK is to retain the data, which is the opposite of the AWS CloudFormation default.

Identifiers

The AWS CDK deals with many types of identifiers and names. To use the AWS CDK effectively and avoid errors, you need to understand the types of identifiers.

Identifiers must be unique within the scope in which they are created; they do not need to be globally unique in your AWS CDK application.

If you attempt to create an identifier with the same value within the same scope, the AWS CDK throws an exception.

Construct IDs

The most common identifier, `id`, is the identifier passed as the second argument when instantiating a construct object. This identifier, like all identifiers, need only be unique within the scope in which it is created, which is the first argument when instantiating a construct object.

> **Note**
> The `id` of a stack is also the identifier you use to refer to it in the the section called "AWS CDK Toolkit" (p. 279).

Let's look at an example where we have two constructs with the identifier `MyBucket` in our app. However, since they are defined in different scopes, the first in the scope of the stack with the identifier `Stack1`, and the second in the scope of a stack with the identifier `Stack2`, that doesn't cause any sort of conflict, and they can co-exist in the same app without any issues.

TypeScript

```
import { App, Construct, Stack, StackProps } from '@aws-cdk/core';
import * as s3 from '@aws-cdk/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/core');
const s3 = require('@aws-cdk/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.core import App, Construct, Stack, StackProps
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.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.s3.Bucket;

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

    public MyStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.core.App;

public class Main {
    public static void main(String[] args) {
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

C#

```
using core = Amazon.CDK;
using s3 = Amazon.CDK.AWS.S3;

public class MyStack : core.Stack
{
    public MyStack(core.App scope, string id, core.IStackProps props) : base(scope, id,
 props)
    {
        new s3.Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new core.App();
```

```
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

Paths

The constructs in an AWS CDK application form a hierarchy rooted in the `App` class. We refer to the collection of IDs from a given construct, its parent construct, its grandparent, and so on to the root of the construct tree, as a *path*.

The AWS CDK typically displays paths in your templates as a string, with the IDs from the levels separated by slashes, starting at the node just below the root `App` instance, which is usually a stack. For example, the paths of the two Amazon S3 bucket resources in the previous code example are `Stack1/MyBucket` and `Stack2/MyBucket`.

You can access the path of any construct programmatically, as shown in the following example, which gets the path of `myConstruct` (or `my_construct`, as Python developers would write it). Since IDs must be unique within the scope they are created, their paths are always unique within a AWS CDK application.

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;
```

Unique IDs

Since AWS CloudFormation requires that all logical IDs in a template are unique, the AWS CDK must be able to generate a unique identifier for each construct in an application. Resources have paths that are globally unique (the names of all scopes from the stack to a specific resource) so the AWS CDK generates the necessary unique identifiers by concatenating the elements of the path and adding an 8-digit hash. (The hash is necessary to distinguish distinct paths, such as `A/B/C` and `A/BC`, that would result in the same AWS CloudFormation identifier, since AWS CloudFormation identifiers are alphanumeric and cannot contain slashes or other separator characters.) The AWS CDK calls this string the *unique ID* of the construct.

In general, your AWS CDK app should not need to know about unique IDs. You can, however, access the unique ID of any construct programmatically, as shown in the following example.

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);
```

The *address* is another kind of unique identifier that uniquely distinguishes CDK resources. Derived from the SHA-1 hash of the path, it is not human-readable, but its constant, relatively short length (always 42 hexadecimal characters) makes it useful in situations where the "traditional" unique ID might be too long. Some constructs may use the address in the synthesized AWS CloudFormation template instead of the unique ID. Again, your app generally should not need to know about its constructs' addresses, but you can retrieve a construct's address as follows.

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;
```

Logical IDs

Unique IDs serve as the *logical identifiers*, which are sometimes called *logical names*, of resources in the generated AWS CloudFormation templates for those constructs that represent AWS resources.

For example, the Amazon S3 bucket in the previous example that is created within `Stack2` results in an `AWS::S3::Bucket` resource with the logical ID `Stack2MyBucket4DD88B4F` in the resulting AWS CloudFormation template.

Think of construct IDs as part of your construct's public contract. If you change the ID of a construct in your construct tree, AWS CloudFormation will replace the deployed resource instances of that construct, potentially causing service interruption or data loss.

Logical ID stability

Avoid changing the logical ID of a resource between deployments. Since AWS CloudFormation identifies resources by their logical ID, if you change the logical ID of a resource, AWS CloudFormation deletes the existing resource, and then creates a new resource with the new logical ID.

Tokens

Tokens represent values that can only be resolved at a later time in the lifecycle of an app (see the section called “App lifecycle” (p. 83)). For example, the name of an Amazon S3 bucket that you define in your AWS CDK app is only allocated when the AWS CloudFormation template is synthesized. If you print the `bucket.bucketName` attribute, which is a string, you see it contains something like the following.

```
${TOKEN[Bucket.Name.1234]}
```

This is how the AWS CDK encodes a token whose value is not yet known at construction time, but will become available later. The AWS CDK calls these placeholders *tokens*. In this case, it's a token encoded as a string.

You can pass this string around as if it was the name of the bucket, such as in the following example, where the bucket name is specified as an environment variable to an AWS Lambda function.

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(new HashMap<String, String>() {{
            put("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
    }
});
```

When the AWS CloudFormation template is finally synthesized, the token is rendered as the AWS CloudFormation intrinsic `{ "Ref": "MyBucket" }`. At deployment time, AWS CloudFormation replaces this intrinsic with the actual name of the bucket that was created.

Tokens and token encodings

Tokens are objects that implement the IResolvable interface, which contains a single `resolve` method. The AWS CDK calls this method during synthesis to produce the final value for the AWS CloudFormation template. Tokens participate in the synthesis process to produce arbitrary values of any type.

> **Note**
> You'll hardly ever work directly with the `IResolvable` interface. You will most likely only see string-encoded versions of tokens.

Other functions typically only accept arguments of basic types, such as `string` or `number`. To use tokens in these cases, you can encode them into one of three types using static methods on the core.Token class.

- `Token.asString` to generate a string encoding (or call `.toString()` on the token object)
- `Token.asList` to generate a list encoding
- `Token.asNumber` to generate a numeric encoding

These take an arbitrary value, which can be an `IResolvable`, and encode them into a primitive value of the indicated type.

> **Important**
> Because any one of the previous types can potentially be an encoded token, be careful when you parse or try to read their contents. For example, if you attempt to parse a string to extract a value from it, and the string is an encoded token, your parsing will fail. Similarly, if you attempt to query the length of an array, or perform math operations with a number, you must first verify that they are not encoded tokens.

To check whether a value has an unresolved token in it, call the `Token.isUnresolved` (Python: `is_unresolved`) method.

The following example validates that a string value, which could be a token, is no more than 10 characters long.

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");
```

If **name** is a token, validation isn't performed, and an error could still occur in a later stage in the lifecycle, such as during deployment.

> **Note**
> You can use token encodings to escape the type system. For example, you could string-encode a token that produces a number value at synthesis time. If you use these functions, it's your responsibility to ensure that your template resolves to a usable state after synthesis.

String-encoded tokens

String-encoded tokens look like the following.

```
${TOKEN[Bucket.Name.1234]}
```

They can be passed around like regular strings, and can be concatenated, as shown in the following example.

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";
```

You can also use string interpolation, if your language supports it, as shown in the following example.

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";
```

Avoid manipulating the string in other ways. For example, taking a substring of a string is likely to break the string token.

List-encoded tokens

List-encoded tokens look like the following

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

The only safe thing to do with these lists is pass them directly to other constructs. Tokens in string list form cannot be concatenated, nor can an element be taken from the token. The only safe way to manipulate them is by using AWS CloudFormation intrinsic functions like Fn.select.

Number-encoded tokens

Number-encoded tokens are a set of tiny negative floating-point numbers that look like the following.

```
-1.8881545897087626e+289
```

As with list tokens, you cannot modify the number value, as doing so is likely to break the number token. The only allowed operation is to pass the value around to another construct.

Lazy values

In addition to representing deploy-time values, such as AWS CloudFormation parameters (p. 128), Tokens are also commonly used to represent synthesis-time lazy values. These are values for which the final value will be determined before synthesis has completed, just not at the point where the value is constructed. Use tokens to pass a literal string or number value to another construct, while the actual value at synthesis time may depend on some calculation that has yet to occur.

You can construct tokens representing synth-time lazy values using static methods on the `Lazy` class, such as Lazy.stringValue (Python: `Lazy.string_value`) and Lazy.numberValue (Python: `Lazy.number_value`. These methods accept an object whose `produce` property is a function that accepts a context argument and returns the final value when called.

The following example creates an Auto Scaling group whose capacity is determined after its creation.

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;
```

Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method stack.toJsonString, as shown in the following example.

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(new HashMap<String, String>() {{
        put("value", bucket.getBucketName());
}});
```

C#

```
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{
    ["value"] = bucket.BucketName
});
```

Parameters

AWS CloudFormation templates can contain parameters—custom values that are supplied at deployment time and incorporated into the template. Since the AWS CDK synthesizes AWS CloudFormation templates, it too offers support for deployment-time parameters.

Using the AWS CDK, you can both define parameters, which can then be used in the properties of constructs you create, and you can also deploy stacks containing parameters.

When deploying the AWS CloudFormation template using the AWS CDK Toolkit, you provide the parameter values on the command line. If you deploy the template through the AWS CloudFormation console, you are prompted for the parameter values.

In general, we recommend against using AWS CloudFormation parameters with the AWS CDK. Unlike context values (p. 159) or environment variables, the usual way to pass values into your AWS CDK apps without hard-coding them, parameter values are not available at synthesis time, and thus cannot be easily used in other parts of your AWS CDK app, particularly for control flow.

> **Note**
> To do control flow with parameters, you can use `CfnCondition` constructs, although this is awkward compared to native `if` statements.

Using parameters requires you to be mindful of how the code you're writing behaves at deployment time, as well as at synthesis time. This makes it harder to understand and reason about your AWS CDK application, in many cases for little benefit.

It is better, again in general, to have your CDK app accept any necessary information from the user and use it directly to declare constructs in your CDK app. An ideal AWS CDK-generated AWS CloudFormation template is concrete, with no values remaining to be specified at deployment time.

There are, however, use cases to which AWS CloudFormation parameters are uniquely suited. If you have separate teams defining and deploying infrastructure, for example, you can use parameters to make the generated templates more widely useful. Additionally, the AWS CDK's support for AWS CloudFormation parameters lets you use the AWS CDK with AWS services that use AWS CloudFormation templates (such as AWS Service Catalog), which use parameters to configure the template being deployed.

Defining parameters

Use the `CfnParameter` class to define a parameter. You'll want to specify at least a type and a description for most parameters, though both are technically optional. The description appears when the user is prompted to enter the parameter's value in the AWS CloudFormation console. For more information on the available types, see Types.

> **Note**
> You can define parameters in any scope, but we recommend defining parameters at the stack level so that their logical ID does not change when you refactor your code.

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"
});
```

Using parameters

A `CfnParameter` instance exposes its value to your AWS CDK app via a token (p. 122). Like all tokens, the parameter's token is resolved at synthesis time, but it resolves to a reference to the parameter defined in the AWS CloudFormation template, which will be resolved at deploy time, rather than to a concrete value.

You can retrieve the token as an instance of the `Token` class, or in string, string list, or numeric encoding, depending on the type of value required by the class or method you want to use the parameter with.

TypeScript

Property	kind of value
`value`	`Token` class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

JavaScript

Property	kind of value
`value`	`Token` class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

Python

Property	kind of value
`value`	`Token` class instance
`value_as_list`	The token represented as a string list
`value_as_number`	The token represented as a number
`value_as_string`	The token represented as a string

Java

Property	kind of value
`getValue()`	`Token` class instance
`getValueAsList()`	The token represented as a string list

Property	kind of value
`getValueAsNumber()`	The token represented as a number
`getValueAsString()`	The token represented as a string

C#

Property	kind of value
`Value`	`Token` class instance
`ValueAsList`	The token represented as a string list
`ValueAsNumber`	The token represented as a number
`ValueAsString`	The token represented as a string

For example, to use a parameter in a Bucket definition:

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
};
```

Deploying with parameters

A generated template containing parameters can be deployed in the usual way through the AWS CloudFormation console; you are prompted for the values of each parameter.

The AWS CDK Toolkit (`cdk` command-line tool) also supports specifying parameters at deployment. You may provide these on the command line following the `--parameters` flag. You might deploy a stack that uses the `uploadBucketName` parameter like this.

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

To define multiple parameters, use multiple `--parameters` flags.

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

If you are deploying multiple stacks, you can specify a different value of each parameter for each stack by prefixing the name of the parameter with the stack name and a colon.

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

By default, the AWS CDK retains values of parameters from previous deployments and uses them in subsequent deployments if they are not specified explicitly. Use the `--no-previous-parameters flag` to require all parameters to be specified.

Tagging

Tags are informational key-value elements that you can add to constructs in your AWS CDK app. A tag applied to a given construct also applies to all of its taggable children. Tags are included in the AWS CloudFormation template synthesized from your app and are applied to the AWS resources it deploys. You can use tags to identify and categorize resources to simplify management, in cost allocation, and for access control, as well as for any other purposes you devise.

> **Tip**
> For more information about how you can use tags with your AWS resources, see the white paper Tagging Best Practices.

The `Tags` class includes the static method `of()`, through which you can add tags to, or remove tags from, the specified construct.

- `Tags.of(SCOPE).add()` applies a new tag to the given construct and all of its children.
- `Tags.of(SCOPE).remove()` removes a tag from the given construct and any of its children, including tags a child construct may have applied to itself.

> **Note**
> Tagging is implemented using the section called “Aspects” (p. 165). Aspects are a way to apply an operation (such as tagging) to all constructs in a given scope.

The following example applies the tag **key** with the value **value** to a construct.

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");
```

The following example deletes the tag **key** from a construct.

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");
```

Tag priorities

The AWS CDK applies and removes tags recursively. If there are conflicts, the tagging operation with the highest priority wins. (Priorities are set using the optional `priority` property.) If the priorities of two operations are the same, the tagging operation closest to the bottom of the construct tree wins. By default, applying a tag has a priority of 100 (except for tags added directly to an AWS CloudFormation resource, which has a priority of 50) and removing a tag has a priority of 200.

The following applies a tag with a priority of 300 to a construct.

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 });
```

Optional properties

Tags support `properties` that fine-tune how tags are applied to, or removed from, resources. All properties are optional.

`applyToLaunchedInstances` (Python: `apply_to_launched_instances`)

Available for add() only. By default, tags are applied to instances launched in an Auto Scaling group. Set this property to **false** to ignore instances launched in an Auto Scaling group.

`includeResourceTypes/excludeResourceTypes` (Python: `include_resource_types/exclude_resource_types`)

Use these to manipulate tags only on a subset of resources, based on AWS CloudFormation resource types. By default, the operation is applied to all resources in the construct subtree, but this can be changed by including or excluding certain resource types. Exclude takes precedence over include, if both are specified.

`priority`

Use this to set the priority of this operation with respect to other `Tags.add()` and `Tags.remove()` operations. Higher values take precedence over lower values. The default is 100 for add operations (50 for tags applied directly to AWS CloudFormation resources) and 200 for remove operations.

The following example applies the tag **tagname** with the value **value** and priority **100** to resources of type **AWS::Xxx::Yyy** in the construct, but not to instances launched in an Amazon EC2 Auto Scaling group or to resources of type **AWS::Xxx::Zzz**. (These are placeholders for two arbitrary but different AWS CloudFormation resource types.)

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("key", "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
});
```

The following example removes the tag **tagname** with priority **200** from resources of type **AWS::Xxx::Yyy** in the construct, but not from resources of type **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
});
```

Example

The following example adds the tag key **StackType** with value **TheBest** to any resource created within the Stack named `MarketingSystem`. Then it removes it again from all resources except Amazon EC2 VPC subnets. The result is that only the subnets have the tag applied.

TypeScript

```
import { App, Stack, Tags } from '@aws-cdk/core';

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/core');

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.core 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.core.App;
import software.amazon.awscdk.core.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"]
});
```

The following code achieves the same result. Consider which approach (inclusion or exclusion) makes your intent clearer.

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"]
});
```

Assets

Assets are local files, directories, or Docker images that can be bundled into AWS CDK libraries and apps; for example, a directory that contains the handler code for an AWS Lambda function. Assets can represent any artifact that the app needs to operate.

You typically reference assets through APIs that are exposed by specific AWS constructs. For example, when you define a lambda.Function construct, the code property lets you pass an asset (directory). `Function` uses assets to bundle the contents of the directory and use it for the function's code. Similarly, ecs.ContainerImage.fromAsset uses a Docker image built from a local directory when defining an Amazon ECS task definition.

Assets in detail

When you refer to an asset in your app, the cloud assembly (p. 84) synthesized from your application includes metadata information with instructions for the AWS CDK CLI on where to find the asset on the local disk, and what type of bundling to perform based on the type of asset, such as a directory to compress (zip) or a Docker image to build.

The AWS CDK generates a source hash for assets, which can be used at construction time to determine whether the contents of an asset have changed.

By default, the AWS CDK creates a copy of the asset in the cloud assembly directory, which defaults to `cdk.out`, under the source hash. This is so that the cloud assembly is self-contained and moved over to a different host for deployment. See the section called “Cloud assemblies” (p. 84) for details.

The AWS CDK also synthesizes AWS CloudFormation parameters that the AWS CDK CLI specifies during deployment. The AWS CDK uses those parameters to refer to the deploy-time values of the asset.

When the AWS CDK deploys an app that references assets (either directly by the app code or through a library), the AWS CDK CLI first prepares and publishes them to Amazon S3 or Amazon ECR, and only then deploys the stack. The AWS CDK specifies the locations of the published assets as AWS CloudFormation parameters to the relevant stacks, and uses that information to enable referencing these locations within an AWS CDK app.

This section describes the low-level APIs available in the framework.

Asset types

The AWS CDK supports the following types of assets:

Amazon S3 Assets

These are local files and directories that the AWS CDK uploads to Amazon S3.

Docker Image

These are Docker images that the AWS CDK uploads to Amazon ECR.

These asset types are explained in the following sections.

Amazon S3 assets

You can define local files and directories as assets, and the AWS CDK packages and uploads them to Amazon S3 through the aws-s3-assets module.

The following example defines a local directory asset and a file asset.

TypeScript

```
import { Asset } from '@aws-cdk/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/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")
});
```

In most cases, you don't need to directly use the APIs in the `aws-s3-assets` module. Modules that support assets, such as `aws-lambda`, have convenience methods that enable you to use assets. For Lambda functions, the asset property enables you to specify a directory or a .zip file in the local file system.

Lambda function example

A common use case is to create AWS Lambda functions with the handler code, which is the entry point for the function, as an Amazon S3 asset.

The following example uses an Amazon S3 asset to define a Python handler in the local directory `handler` and creates a Lambda function with the local directory asset as the `code` property. Below is the Python code for the handler.

```
def lambda_handler(event, context):
  message = 'Hello World!'
  return {
    'message': message
  }
```

The code for the main AWS CDK app should look like the following.

TypeScript

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

export class HelloAssetStack extends cdk.Stack {
  constructor(scope: cdk.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/core');
const lambda = require('@aws-cdk/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.core import Stack, 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.core.Stack;
import software.amazon.awscdk.core.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"
        });
    }
}
```

The `Function` method uses assets to bundle the contents of the directory and use it for the function's code.

Deploy-time attributes example

Amazon S3 asset types also expose deploy-time attributes (p. 99) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command **cdk synth** displays asset properties as AWS CloudFormation parameters.

The following example uses deploy-time attributes to pass the location of an image asset into a Lambda function as environment variables. (The kind of file doesn't matter; the PNG image used here is just an example.)

TypeScript

```
import { Asset } from '@aws-cdk/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_URL': imageAsset.s3Url
  }
});
```

JavaScript

```
const { Asset } = require('@aws-cdk/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_URL': imageAsset.s3Url
  }
});
```

Python

```
import os.path

from aws_cdk import 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_URL=image_asset.s3_url))
```

Java

```
import java.io.File;

import software.amazon.awscdk.core.Stack;
```

```
import software.amazon.awscdk.core.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(new HashMap<String, String>() {{
                    put("S3_BUCKET_NAME", imageAsset.getS3BucketName());
                    put("S3_OBJECT_KEY", imageAsset.getS3ObjectKey());
                    put("S3_URL", imageAsset.getS3Url());
                }}).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_URL"] = imageAsset.S3Url
    }
});
```

Permissions

If you use Amazon S3 assets directly through the aws-s3-assets module, IAM roles, users, or groups, and need to read assets in runtime, grant those assets IAM permissions through the asset.grantRead method.

The following example grants an IAM group read permissions on a file asset.

TypeScript

```
import { Asset } from '@aws-cdk/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/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
from aws_cdk import 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.grantRead(group)
```

Java

```
import java.io.File;

import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.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);
```

Docker image assets

The AWS CDK supports bundling local Docker images as assets through the aws-ecr-assets module.

The following example defines a docker image that is built locally and pushed to Amazon ECR. Images are built from a local Docker context directory (with a Dockerfile) and uploaded to Amazon ECR by the AWS CDK CLI or your app's CI/CD pipeline, and can be naturally referenced in your AWS CDK app.

TypeScript

```
import { DockerImageAsset } from '@aws-cdk/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```
const { DockerImageAsset } = require('@aws-cdk/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(Path.Combine(Directory.GetCurrentDirectory(), "my-image"))
});
```

The `my-image` directory must include a Dockerfile. The AWS CDK CLI builds a Docker image from `my-image`, pushes it to an Amazon ECR repository, and specifies the name of the repository as an AWS CloudFormation parameter to your stack. Docker image asset types expose deploy-time attributes (p. 99) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command **cdk synth** displays asset properties as AWS CloudFormation parameters.

Amazon ECS task definition example

A common use case is to create an Amazon ECS TaskDefinition to run docker containers. The following example specifies the location of a Docker image asset that the AWS CDK builds locally and pushes to Amazon ECR.

TypeScript

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

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromAsset(path.join(__dirname, "..", "demo-image"))
});
```

JavaScript

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

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromAsset(path.join(__dirname, "..", "demo-image"))
});
```

Python

```
import aws_cdk.aws_ecs as ecs

import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_asset(
```

```
        os.path.join(dirname, "..", "demo-image")))
```

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;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder()
            .image(ContainerImage.fromAsset(new File(startDir,
                        "demo-image").toString())).build());
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromAsset(Path.Combine(Directory.GetCurrentDirectory(),
 "demo-image");
});
```

Deploy-time attributes example

The following example shows how to use the deploy-time attributes `repository` and `imageUri` to create an Amazon ECS task definition with the AWS Fargate launch type. Note that the Amazon ECR repo lookup requires the image's tag, not its URI, so we snip it from the end of the asset's URI.

TypeScript

```
import * as ecs from '@aws-cdk/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from '@aws-cdk/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/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('@aws-cdk/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())
});
```

Build arguments example

You can provide customized build arguments for the Docker build step through the `buildArgs` (Python: `build_args`) property option when the AWS CDK CLI builds the image during deployment.

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(new HashMap<String, String>() {{
                put("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"
    }
});
```

Permissions

If you use a module that supports Docker image assets, such as aws-ecs, the AWS CDK manages permissions for you when you use assets directly or through ContainerImage.fromEcrRepository (Python: `from_ecr_repository`). If you use Docker image assets directly, you need to ensure that the consuming principal has permissions to pull the image.

In most cases, you should use asset.repository.grantPull method (Python: `grant_pull`. This modifies the IAM policy of the principal to enable it to pull images from this repository. If the principal that is pulling the image is not in the same account or is an AWS service, such as AWS CodeBuild, that does not assume a role in your account, you must grant pull permissions on the resource policy and not on the principal's policy. Use the asset.repository.addToResourcePolicy method (Python: `add_to_resource_policy`) to grant the appropriate principal permissions.

AWS CloudFormation resource metadata

> **Note**
> This section is relevant only for construct authors. In certain situations, tools need to know that a certain CFN resource is using a local asset. For example, you can use the AWS SAM CLI to invoke Lambda functions locally for debugging purposes. See the section called “SAM CLI” (p. 296) for details.

To enable such use cases, external tools consult a set of metadata entries on AWS CloudFormation resources:

- `aws:asset:path` – Points to the local path of the asset.
- `aws:asset:property` – The name of the resource property where the asset is used.

Using these two metadata entries, tools can identify that assets are used by a certain resource, and enable advanced local experiences.

To add these metadata entries to a resource, use the `asset.addResourceMetadata` (Python: `add_resource_metadata`) method.

Permissions

The AWS Construct Library uses a few common, widely-implemented idioms to manage access and permissions. The IAM module provides you with the tools you need to use these idioms.

Principals

An IAM principal is an entity that can be authenticated in order to access AWS resources, such as a user, a service, or an application. The AWS Construct Library supports many types of principals, including:

1. IAM resources such as `Role`, `User`, and `Group`
2. Service principals (`new iam.ServicePrincipal('service.amazonaws.com')`)
3. Federated principals (`new iam.FederatedPrincipal('cognito-identity.amazonaws.com')`)
4. Account principals (`new iam.AccountPrincipal('0123456789012')`)
5. Canonical user principals (`new iam.CanonicalUserPrincipal('79a59d[...]7ef2be')`)
6. AWS organizations principals (`new iam.OrganizationPrincipal('org-id')`)
7. Arbitrary ARN principals (`new iam.ArnPrincipal(res.arn)`)
8. An `iam.CompositePrincipal(principal1, principal2, ...)` to trust multiple principals

Grants

Every construct that represents a resource that can be accessed, such as an Amazon S3 bucket or Amazon DynamoDB table, has methods that grant access to another entity. All such methods have names starting with **grant**. For example, Amazon S3 buckets have the methods `grantRead` and `grantReadWrite` (Python: `grant_read`, `grant_read_write`) to enable read and read/write access, respectively, from an entity to the bucket without having to know exactly which Amazon S3 IAM permissions are required to perform these operations.

The first argument of a **grant** method is always of type IGrantable. This interface represents entities that can be granted permissions—that is, resources with roles, such as the IAM objects `Role`, `User`, and `Group`.

Other entities can also be granted permissions. For example, later in this topic, we show how to grant a CodeBuild project access to an Amazon S3 bucket. Generally, the associated role is obtained via a `role` property on the entity being granted access. Other entities that can be granted permissions are Amazon EC2 instances and CodeBuild projects.

Resources that use execution roles, such as `lambda.Function`, also implement `IGrantable`, so you can grant them access directly instead of granting access to their role. For example, if `bucket` is an Amazon S3 bucket, and `function` is a Lambda function, the code below grants the function read access to the bucket.

TypeScript

```
bucket.grantRead(function);
```

JavaScript

```
bucket.grantRead(function);
```

Python

```
bucket.grant_read(function)
```

Java

```
bucket.grantRead(function);
```

C#

```
bucket.GrantRead(function);
```

Sometimes permissions must be applied while your stack is being deployed. One such case is when you grant a AWS CloudFormation custom resource access to some other resource. The custom resource will be invoked during deployment, so it must have the specified permissions at deployment time. Another case is when a service verifies that the role you pass to it has the right policies applied (a number of AWS services do this to make sure you didn't forget to set the policies). In those cases, the deployment may fail if the permissions are applied too late.

To force the grant's permissions to be applied before another resource is created, you can add a dependency on the grant itself, as shown here. Though the return value of grant methods is commonly discarded, every grant method in fact returns an `iam.Grant` object.

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);
```

Roles

The IAM package contains a `Role` construct that represents IAM roles. The following code creates a new role, trusting the Amazon EC2 service.

TypeScript

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

```
const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

JavaScript

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

You can add permissions to a role by calling the role's `addToPolicy` method (Python: `add_to_policy`), passing in a `PolicyStatement` that defines the rule to be added. The statement is added to the role's default policy; if it has none, one is created.

The following example adds a `Deny` policy statement to the role for the actions `ec2:SomeAction` and `s3:AnotherAction` on the resources `bucket` and `otherRole` (Python: `other_role`), under the condition that the authorized service is 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(new HashMap<String, Object>() {{
            put("StringEquals", new HashMap<String, String>() {{
                put("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"
        }
    }
}));
```

In our example above, we've created a new `PolicyStatement` inline with the `addToPolicy` (Python: `add_to_policy`) call. You can also pass in an existing policy statement or one you've modified. The PolicyStatement object has numerous methods for adding principals, resources, conditions, and actions.

If you're using a construct that requires a role to function correctly, you can either pass in an existing role when instantiating the construct object, or let the construct create a new role for you, trusting the appropriate service principal. The following example uses such a construct: a CodeBuild project.

TypeScript

```
import * as codebuild from '@aws-cdk/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/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
});
```

Once the object is created, the role (whether the role passed in or the default one created by the construct) is available as the property `role`. This property is not available on imported resources, however, so such constructs have an `addToRolePolicy` (Python: `add_to_role_policy`) method that does nothing if the construct is an imported resource, and calls the `addToPolicy` (Python: `add_to_policy`) method of the `role` property otherwise, saving you the trouble of handling the undefined case explicitly. The following example demonstrates:

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
}));
```

Resource policies

A few resources in AWS, such as Amazon S3 buckets and IAM roles, also have a resource policy. These constructs have an `addToResourcePolicy` method (Python: `add_to_resource_policy`), which takes a `PolicyStatement` as its argument. Every policy statement added to a resource policy must specify at least one principal.

In the following example, the Amazon S3 bucket `bucket` grants a role with the `s3:SomeAction` permission to itself.

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 }
}));
```

Runtime context

Context values are key-value pairs that can be associated with a stack or construct. The AWS CDK uses context to cache information from your AWS account, such as the Availability Zones in your account or the Amazon Machine Image (AMI) IDs used to start your instances. the section called “Feature flags” (p. 164) are also context values. You can create your own context values for use by your apps or constructs.

Context keys are strings, and values may be any type supported by JSON: numbers, strings, arrays, or objects.

> **Important**
> Context values are managed by the AWS CDK and its constructs, including constructs you may write. You should not attempt to add context values manually. It is useful to review `cdk.context.json` to see what values are being cached; by convention, the keys start with the name of the CDK package that set them. You sholud follow this convention when setting your own values.

Construct context

Context values can be provided to your AWS CDK app in six different ways:

- Automatically from the current AWS account.
- Through the **--context** option to the **cdk** command. (These values are always strings.)
- In the project's `cdk.context.json` file.
- In the `context` key of the project's `cdk.json` file.
- In the `context` key of your `~/.cdk.json` file.
- In your AWS CDK app using the `construct.node.setContext()` method.

The project file `cdk.context.json` is where the AWS CDK caches context values retrieved from your AWS account. This practice avoids unexpected changes to your deployments when, for example, a new Amazon Linux AMI is released, changing your Auto Scaling group. The AWS CDK does not write context data to any of the other files listed.

We recommend that your project's context files be placed under version control along with the rest of your application, as the information in them is part of your app's state and is critical to being able to synthesize and deploy consistently. It is also critical to successful automated deployment of stacks that rely on context values (for example, using CDK Pipelines (p. 255)).

Context values are scoped to the construct that created them; they are visible to child constructs, but not to siblings. Context values set by the AWS CDK Toolkit (the **cdk** command), whether automatically, from a file, or from the **--context** option, are implicitly set on the `App` construct, and so are visible to every construct in the app.

You can get a context value using the `construct.node.tryGetContext` method. If the requested entry is not found on the current construct or any of its parents, the result is `undefined` (or your language's equivalent, such as `None` in Python).

Context methods

The AWS CDK supports several context methods that enable AWS CDK apps to get contextual information. For example, you can get a list of Availability Zones that are available in a given AWS account and region, using the stack.availabilityZones method.

The following are the context methods:

HostedZone.fromLookup

Gets the hosted zones in your account.

stack.availabilityZones

Gets the supported Availability Zones.

StringParameter.valueFromLookup

Gets a value from the current Region's Amazon EC2 Systems Manager Parameter Store.

Vpc.fromLookup

Gets the existing Amazon Virtual Private Clouds in your accounts.

LookupMachineImage

Looks up a machine image for use with a NAT instance in an Amazon Virtual Private Cloud.

If a required context value isn't available, the AWS CDK app notifies the AWS CDK CLI that the context information is missing. The CLI then queries the current AWS account for the information, stores the resulting context information in the `cdk.context.json` file, and executes the AWS CDK app again with the context values.

Viewing and managing context

Use the **cdk context** command to view and manage the information in your `cdk.context.json` file. To see this information, use the **cdk context** command without any options. The output should be something like the following.

```
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.
```

To remove a context value, run **cdk context --reset**, specifying the value's corresponding key or number. The following example removes the value that corresponds to the second key in the preceding example, which is the list of availability zones in the Ireland region.

```
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.
```

Therefore, if you want to update to the latest version of the Amazon Linux AMI, you can use the preceding example to do a controlled update of the context value and reset it, and then synthesize and deploy your app again.

```
cdk synth
```

To clear all of the stored context values for your app, run **cdk context --clear**, as follows.

```
cdk context --clear
```

Only context values stored in `cdk.context.json` can be reset or cleared. The AWS CDK does not touch other context values. To protect a context value from being reset using these commands, then, you might copy the value to `cdk.json`.

AWS CDK Toolkit `--context` flag

Use the `--context` (`-c` for short) option to pass runtime context values to your CDK app during synthesis or deployment.

```
cdk synth --context key=value MyStack
```

To specify multiple context values, repeat the **--context** option any number of times, providing one key-value pair each time.

```
cdk synth --context key1=value1 --context key2=value2 MyStack
```

When deploying multiple stacks, the specified context values are normally passed to all of them. If you wish, you may specify different values for each stack by prefixing the stack name to the context value.

```
cdk synth --context Stack1:key=value --context Stack2:key=value Stack1 Stack2
```

Example

Below is an example of importing an existing Amazon VPC using AWS CDK context.

TypeScript

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

export class ExistsVpcStack extends cdk.Stack {

  constructor(scope: cdk.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/core');
const ec2 = require('@aws-cdk/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.core as cdk
import aws_cdk.aws_ec2 as ec2

class ExistsVpcStack(cdk.Stack):

    def __init__(scope: cdk.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.core.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;
```

```
public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(App context, String id) {
        this(context, id, null);
    }

    public ExistsVpcStack(App 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;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(App 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()
        });
    }
}
```

You can use **cdk diff** to see the effects of passing in a context value on the command line:

```
cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

```
Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
 {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
```

The resulting context values can be viewed as shown here.

```
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"
    ]
  }
}
```

Feature flags

The AWS CDK uses *feature flags* to enable potentially breaking behaviors in a release. Flags are stored as the section called “Context” (p. 159) values in `cdk.json` (or `~/.cdk.json`) as shown here.

```
{
  "app": "npx ts-node bin/tscdk.ts",
  "context": {
    "@aws-cdk/core:enableStackNameDuplicates": true
  }
}
```

The names of all feature flags begin with the NPM name of the package affected by the particular flag. In the example above, this is `@aws-cdk/core`, the AWS CDK framework itself, since the flag affects stack naming rules, a core AWS CDK function. AWS Construct Library modules can also use feature flags.

Feature flags are disabled by default, so existing projects that do not specify the flag will continue to work as expected with later AWS CDK releases. New projects created using **cdk init** include flags enabling all features available in the release that created the project. Edit `cdk.json` to disable any flags for which you prefer the old behavior, or to add flags to enable new behaviors after upgrading the AWS CDK.

See the `CHANGELOG` in a given release for a description of any new feature flags added in that release. The AWS CDK source file `features.ts` provides a complete list of all current feature flags.

As feature flags are stored in `cdk.json`, they are not removed by the **cdk context --reset** or **cdk context --clear** commands.

Aspects

Aspects are a way to apply an operation to all constructs in a given scope. The aspect could modify the constructs, such as by adding tags, or it could verify something about the state of the constructs, such as ensuring that all buckets are encrypted.

To apply an aspect to a construct and all constructs in the same scope, call `Aspects.of(SCOPE).add()` with a new aspect, as shown in the following example.

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(...));
```

The AWS CDK currently uses aspects only to tag resources (p. 132), but the framework is extensible and can also be used for other purposes. For example, you can use it to validate or change the AWS CloudFormation resources that are defined for you by higher-level constructs.

Aspects in detail

Aspects employ the visitor pattern. An aspect is a class that implements the following interface.

TypeScript

```
interface IAspect {
   visit(node: IConstruct): void;}
```

JavaScript

JavaScript doesn't have interfaces as a language feature, so an aspect is simply an instance of a class having a `visit` method that accepts the node to be operated on.

Python

Python doesn't have interfaces as a language feature, so an aspect is simply an instance of a class having a `visit` method that accepts the node to be operated on.

Java

```
public interface IAspect {
```

```
    public void visit(Construct node);
}
```

C#

```
public interface IAspect
{
    void Visit(IConstruct node);
}
```

When you call `Aspects.of(SCOPE).add(...)`, the construct adds the aspect to an internal list of aspects. You can obtain the list with `Aspects.of(SCOPE)`.

During the prepare phase (p. 83), the AWS CDK calls the `visit` method of the object for the construct and each of its children in top-down order.

The `visit` method is free to change anything in the construct. In strongly-typed languages, cast the received construct to a more specific type before accessing construct-specific properties or methods.

Example

The following example validates that all buckets created in the stack have versioning enabled. The aspect adds an error annotation to the constructs that fail the validation, which results in the **synth** operation failing and prevents deploying the resulting cloud assembly.

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(core.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.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());
```

Escape hatches

It's possible that neither the high-level constructs nor the low-level CFN Resource constructs have a specific feature you are looking for. There are three possible reasons for this lack of functionality:

- The AWS service feature is available through AWS CloudFormation, but there are no Construct classes for the service.
- The AWS service feature is available through AWS CloudFormation, and there are Construct classes for the service, but the Construct classes don't yet expose the feature.
- The feature is not yet available through AWS CloudFormation.

To determine whether a feature is available through AWS CloudFormation, see AWS Resource and Property Types Reference.

Using AWS CloudFormation constructs directly

If there are no Construct classes available for the service, you can fall back to the automatically generated CFN Resources, which map 1:1 onto all available AWS CloudFormation resources and properties. These resources can be recognized by their name starting with `Cfn`, such as `CfnBucket` or `CfnRole`. You instantiate them exactly as you would use the equivalent AWS CloudFormation resource. For more information, see AWS Resource and Property Types Reference.

For example, to instantiate a low-level Amazon S3 bucket CFN Resource with analytics enabled, you would write something like the following.

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(new HashMap<String, String>() {{
        put("id", "Config");
        // ...
    }})).build();
```

C#

```
new CfnBucket(this, 'MyBucket', new CfnBucketProps {
    AnalyticsConfigurations = new Dictionary<string, string>
    {
        ["id"] = "Config",
        // ...
    }
});
```

In the rare case where you want to define a resource that doesn't have a corresponding `CfnXxx` class, such as a new resource type that hasn't yet been published in the AWS CloudFormation resource specification, you can instantiate the `cdk.CfnResource` directly and specify the resource type and properties. This is shown in the following example.

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(new HashMap<String, Object>() {{
            // Note the PascalCase here! These are CloudFormation identifiers
            put("AnalyticsConfigurations", Arrays.asList(
                    new HashMap<String, String>() {{
                        put("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 List<Dictionary<string, string>>
        {
            new Dictionary<string, string> {
                ["Id"] = "Config"
            }
        }
    }
});
```

For more information, see AWS Resource and Property Types Reference.

Modifying the AWS CloudFormation resource behind AWS constructs

If a Construct is missing a feature or you are trying to work around an issue, you can modify the CFN Resource that is encapsulated by the Construct.

All Constructs contain within them the corresponding CFN Resource. For example, the high-level `Bucket` construct wraps the low-level `CfnBucket` construct. Because the `CfnBucket` corresponds directly to the AWS CloudFormation resource, it exposes all features that are available through AWS CloudFormation.

The basic approach to get access to the CFN Resource class is to use `construct.node.defaultChild` (Python: `default_child`), cast it to the right type (if necessary), and modify its properties. Again, let's take the example of a `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(new HashMap<String, String>() {{
            put("Id", "Config");
            // ...
        }}));
```

C#

```
// Get the CloudFormation resource
```

```
var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List<object> {
    new Dictionary<string, string>
    {
        ["Id"] = "Config",
        // ...
    }
};
```

You can also use this object to change AWS CloudFormation options such as `Metadata` and `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(new HashMap<String, Object>() {{
    put("MetadataKey", "Metadatavalue");
}});
```

C#

```
cfnBucket.CfnOptions.Metadata = new Dictionary<string, object>
{
    ["MetadataKey"] = "Metadatavalue"
};
```

Raw overrides

If there are properties that are missing from the CFN Resource, you can bypass all typing using raw overrides. This also makes it possible to delete synthesized properties.

Use one of the `addOverride` methods (Python: `add_override`) methods, as shown in the following example.

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");
```

Custom resources

If the feature isn't available through AWS CloudFormation, but only through a direct API call, the only solution is to write an AWS CloudFormation Custom Resource to make the API call you need. Don't worry, the AWS CDK makes it easier to write these, and wrap them up into a regular construct interface, so from another user's perspective the feature feels native.

Building a custom resource involves writing a Lambda function that responds to a resource's CREATE, UPDATE and DELETE lifecycle events. If your custom resource needs to make only a single API call, consider using the AwsCustomResource. This makes it possible to perform arbitrary SDK calls during an AWS CloudFormation deployment. Otherwise, you should write your own Lambda function to perform the work you need to get done.

The subject is too broad to completely cover here, but the following links should get you started:

- Custom Resources
- Custom-Resource Example
- For a more fully fledged example, see the DnsValidatedCertificate class in the CDK standard library. This is implemented as a custom resource.

Bootstrapping

Deploying AWS CDK apps into an AWS environment (p. 92) (a combination of an AWS account and region) may require that you provision resources the AWS CDK needs to perform the deployment. These resources include an Amazon S3 bucket for storing files and IAM roles that grant permissions needed to perform deployments. The process of provisioning these initial resources is called *bootstrapping*.

An environment needs to be bootstrapped if any of the following apply.

- An AWS CDK stack being deployed uses the section called “Assets” (p. 138).
- An AWS CloudFormation template generated by the app exceeds 50 kilobytes.
- One or more of the stacks uses the `DefaultSynthesizer`. We will explain stack synthesizers in more detail shortly, but in brief, the `DefaultSynthesizer` is used if you have set the `@aws-cdk/core:newStyleStackSynthesis` feature flag (p. 164) in your app's `cdk.json` *or* if you explicitly create a `DefaultSynthesizer` and pass it to your stack. CDK Pipelines (p. 255) use the `DefaultSynthesizer`, so if your app uses CDK Pipelines, you must bootstrap the environments you will deploy into as well as the environment that contains the pipeline.

The required resources are defined in a AWS CloudFormation stack, called the *bootstrap stack*, which is usually named `CDKToolkit`. Like any AWS CloudFormation stack, it appears in the AWS CloudFormation console once it has been deployed.

The AWS CDK supports two bootstrap templates. At this writing, the AWS CDK is transitioning from one of these templates to the other, but the original template (dubbed "legacy") is still the default. The newer template ("modern") is required by CDK Pipelines today, and will become the default at some point in the future. For details, see the section called “Bootstrapping templates” (p. 177).

Environments are independent, so if you want to deploy to multiple environments (different AWS accounts or different regions in the same account), each environment must be bootstrapped separately.

Important
You may incur AWS charges for data stored in the bootstrapped resources.

Note
Older versions of the modern template created a Customer Master Key (CMK) in each bootstrapped environment by default. To avoid charges for the CMK, re-bootstrap these environments using `--no-bootstrap-customer-key`. The current default is to not use a CMK to avoid these charges.

If you attempt to deploy an AWS CDK application that requires bootstrap resources into an environment that does not have them, you receive an error message telling you that you need to bootstrap.

If you are using CDK Pipelines to deploy into another account's environment, and you receive a message like the following:

```
Policy contains a statement with one or more invalid principals
```

This error message means that the appropriate IAM roles do not exist in the other environment, which is most likely caused by a lack of bootstrapping.

Note
Do not delete and recreate an account's bootstrap stack if you are using CDK Pipelines to deploy into that account. The pipeline will stop working. To update the bootstrap stack to a new version, instead re-run `cdk bootstrap` to update the bootstrap stack in place.

How to bootstrap

Bootstrapping is the deployment of a AWS CloudFormation template to a specific AWS environment (account and region). The bootstrapping template accepts parameters that customize some aspects of the bootstrapped resources (see the section called “Customizing bootstrapping” (p. 178)). Thus, you can bootstrap in one of two ways.

- Use the AWS CDK Toolkit's **cdk bootstrap** command. This is the simplest method and works well if you have only a few environments to bootstrap.

- Deploy the template provided by the AWS CDK Toolkit using another AWS CloudFormation deployment tool. This lets you use AWS CloudFormation Stack Sets or AWS Control Tower as well as the AWS CloudFormation console or the AWS CLI. You can even make small modifications to the template before deployment. This approach is more flexible and is suitable for large-scale deployments.

It is not an error to bootstrap an environment more than once. If an environment you bootstrap has already been bootstrapped, its bootstrap stack will be upgraded if necessary; otherwise, nothing happens.

Bootstrapping with the AWS CDK Toolkit

Use the `cdk bootstrap` command to bootstrap one or more AWS environments. In its basic form, this command bootstraps one or more specified AWS environments (two, in this example).

```
cdk bootstrap aws://ACCOUNT-NUMBER-1/REGION-1 aws://ACCOUNT-NUMBER-2/REGION-2 ...
```

The following examples illustrate bootstrapping of one and two environments, respectively. (Both use the same AWS account.) As shown in the second example, the `aws://` prefix is optional when specifying an environment.

```
cdk bootstrap aws://123456789012/us-east-1
cdk bootstrap 123456789012/us-east-1 123456789012/us-west-1
```

If you do not specify at least one environment in the `cdk bootstrap` command, the AWS CDK Toolkit synthesizes the AWS CDK app in the current directory and bootstraps all the environments referenced in the app. If a stack is environment-agnostic (that is, it does not have an `env` property), the CDK's environment (for example, the one specified using **--profile**, or the default AWS environment otherwise) is applied to make the stack environment-specific, and that environment is then bootstrapped.

For example, the following command synthesizes the current AWS CDK app using the `prod` AWS profile, then bootstraps its environments.

```
cdk bootstrap --profile prod
```

Bootstrapping from the AWS CloudFormation template

AWS CDK bootstrapping is performed by an AWS CloudFormation template. To get a copy of this template in the file `bootstrap-template.yaml`, run the following command.

macOS/Linux

```
cdk bootstrap --show-template > bootstrap-template.yaml
```

Windows

On Windows, PowerShell must be used to preserve the encoding of the template.

```
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-
template.yaml"
```

The template is also available in the AWS CDK GitHub repository.

Deploy this template using your preferred deployment mechanism for AWS CloudFormation templates. For example, the following command deploys the template using the AWS CLI:

macOS/Linux

```
aws cloudformation create-stack \
  --stack-name CDKToolkit \
  --template-body file://bootstrap-template.yaml
```

Windows

```
aws cloudformation create-stack ^
  --stack-name CDKToolkit ^
  --template-body file://bootstrap-template.yaml
```

Bootstrapping templates

At this writing, the AWS CDK is transitioning from one set of bootstrap resources to another. The original bootstrap template, which shipped with the very first version of the AWS CDK, is called the **legacy** template. A newer version of the template with additional resources was added in version 1.25.0. This newer template is called the **modern** template.

The legacy template is still fully supported by the AWS CDK and is in fact the template that is selected by default when you issue `cdk bootstrap`. The modern template is required primarily by the CDK Pipelines module, which can be used to set up a continuous delivery pipeline for your CDK applications. More precisely, the modern template is used by the `DefaultSynthesizer` (see the section called “Stack synthesizers” (p. 180)), and CDK Pipelines requires this synthesizer,

The main differences between the templates are as follows.

Feature	Legacy	Modern
Cross-account deployments	Not allowed	Allowed
AWS CloudFormation Permissions	Deploys using current user's permissions (determined by AWS profile, environment variables, etc.)	Deploys using the permissions specified when the bootstrap stack was provisioned (e.g. using `--trust`)
Versioning	Only one version of bootstrap stack is available	Bootstrap stack is versioned; new resources can be added in future versions, and AWS CDK apps can require a minimum version
Resources*	Amazon S3 bucket	Amazon S3 bucket
		AWS KMS key
		IAM roles
		Amazon ECR repository
		SSM parameter for versioning
Resource naming	Automatically generated	Deterministic
Bucket encryption	Default key	Customer-managed key

* *We will add additional resources to the modern template as needed.*

In AWS CDK version 2, the modern template will be the default bootstrapping template. In version 1, manually select the modern template when bootstrapping by setting the `CDK_NEW_BOOTSTRAP` environment variable.

macOS/Linux

```
export CDK_NEW_BOOTSTRAP=1
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

Windows

```
set CDK_NEW_BOOTSTRAP=1
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

The modern template is also selected when you issue **cdk bootstrap** in an AWS CDK app directory where the `@aws-cdk/core:newStyleStackSynthesis` feature flag is set in the app's `cdk.json` file.

```
{
  // ...
  "context": {
    "@aws-cdk/core:newStyleStackSynthesis": true
  }
}
```

Tip
We recommend always setting `CDK_NEW_BOOTSTRAP` when you want to bootstrap using the modern template. The context key is supported to make sure you bootstrap correctly if your app uses the `DefaultStackSynthesizer`, but relies on you being in an app's directory when bootstrapping.

These two ways to specify the modern template also apply to `cdk bootstrap --show-template`, which will display the modern template if either of these flags is present.

If the environment you are bootstrapping with the modern template has already been bootstrapped with the legacy template, the environment is upgraded to the modern template. The Amazon S3 bucket from the legacy stack is orphaned in the process. Re-deploy all AWS CDK applications in the environment at least once before deleting the legacy bucket.

Customizing bootstrapping

There are two ways to customize the bootstrapping resources.

- Use command-line parameters with the `cdk bootstrap` command. This lets you modify a few aspects of the template.
- Modify the default bootstrap template and deploy it yourself. This gives you unlimited control over the bootstrap resources.

The following command-line options, when used with CDK Toolkit's **cdk bootstrap**, provide commonly-needed adjustments to the bootstrapping template.

- **--bootstrap-bucket-name** overrides the name of the Amazon S3 bucket. May require changes to your CDK app (see the section called “Stack synthesizers” (p. 180)).
- **--bootstrap-kms-key-id** overrides the AWS KMS key used to encrypt the S3 bucket.

- **--tags** adds one or more AWS CloudFormation tags to the bootstrap stack.
- **--termination-protection** prevents the bootstrap stack from being deleted (see Protecting a stack from being deleted in the AWS CloudFormation User Guide)

The following additional switches are available only with the modern bootstrapping template.

- **--cloudformation-execution-policies** specifies the ARNs of managed policies that should be attached to the deployment role assumed by AWS CloudFormation during deployment of your stacks. At least one policy is required; otherwise, AWS CloudFormation will attempt to deploy without permissions and deployments will fail.

  Tip
  The policies must be passed as a single string argument, with the policy ARNs separated by commas, like this:

  ```
  --cloudformation-execution-policies "arn:aws:iam::aws:policy/
  AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCo
  deDeployFullAccess".
  ```

- **--trust** lists the AWS accounts that may deploy into the environment being bootstrapped. Use this flag when bootstrapping an environment that a CDK Pipeline in another environment will deploy into. The account doing the bootstrapping is always trusted.
- **--trust-for-lookup** lists the AWS accounts that may look up context information from the environment being bootstrapped. Use this flag to give accounts permission to synthesize stacks that will be deployed into the environment, without actually giving them permission to deploy those stacks directly. Accounts specified under **--trust** are always trusted for context lookup.
- **--qualifier** a string that is added to the names of all resources in the bootstrap stack. A qualifier lets you avoid name clashes when you provision two bootstrap stacks in the same environment. The default is `hnb659fds` (this value has no significance). Changing the qualifier will require changes to your AWS CDK app (see the section called “Stack synthesizers” (p. 180)).

  Important
  The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list, which by default will extend permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack (p. 178) with policies and trusted accounts you are comfortable with.

Customizing the template

When you need more customization than the AWS CDK Toolkit switches can provide, you can modify the bootstrap template to suit your needs. Remember that you can obtain the template by using the **--show-template** flag. Optionally, set the **CDK_NEW_BOOTSTRAP** environment variable to get the modern template (otherwise, you'll get the legacy template).

macOS/Linux

```
export CDK_NEW_BOOTSTRAP=1
cdk bootstrap --show-template
```

Windows

```
set CDK_NEW_BOOTSTRAP=1
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-
template.yaml"
```

Any modifications you make must adhere to the bootstrapping template contract (p. 186).

Deploy your modified template as described in the section called "Bootstrapping from the AWS CloudFormation template" (p. 176), or using **cdk bootstrap --template**.

```
cdk bootstrap --template bootstrap-template.yaml
```

Stack synthesizers

Your AWS CDK app needs to know about the bootstrapping resources available to it in order to successfully synthesize a stack that can be deployed. The *stack synthesizer* is an AWS CDK class that controls how the stack's template is synthesized, including how it uses bootstrapping resources (for example, how it refers to assets stored in the bootstrap bucket).

The AWS CDK includes two stack synthesizers:

- `LegacyStackSynthesizer` can be used with either bootstrap template. (It requires only an Amazon S3 bucket, and both templates include one.)
- `DefaultStackSynthesizer` requires the modern bootstrap template. It includes capabilities for cross-account deployments and CDK Pipelines (p. 255) deployments.

You can pass a stack synthesizer to a stack when you instantiate it using the `synthesizer` property.

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
    })
});
```

If you don't provide the `synthesizer` property, the default behavior depends on whether the context key `@aws-cdk/core:newStyleStackSynthesis` is set, either in the AWS CDK app's source code or in `cdk.json`. If it is set, synthesis uses a `DefaultStackSynthesizer`; otherwise, a `LegacyStackSynthesizer` is used. This is the usual way of choosing a synthesizer unless you have customized the bootstrap template.

The most important differences between the two built-in stack synthesizers are summarized here.

Feature	LegacyStackSynthesizer	DefaultStackSynthesizer
Bootstrap stack	Both legacy and modern bootstrap stack	Modern bootstrap stack only
Deployments	AWS CDK Toolkit deployments only	AWS CDK Toolkit and CDK Pipelines deployments
Assets	Uses AWS CloudFormation parameters to reference assets	Expects assets to be in a predictable location
Docker image assets	Creates Amazon ECR repository on demand	Pushes images to Amazon ECR repository provisioned by bootstrapping
Roles	Uses AWS CDK Toolkit's current permissions to deploy	Uses roles and permissions provisioned by bootstrapping to deploy
Versioning	Not supported	Confirms versions of bootstrapping resources via embedded AWS CloudFormation rule

Customizing synthesis

Depending on the changes you made to the bootstrap template, you may also need to customize synthesis. The `DefaultStackSynthesizer` can be customized using the properties described below. If none of these properties provide the customizations you require, you can write your synthesizer as a class that implements `IStackSynthesizer` (perhaps deriving from `DefaultStackSynthesizer`).

Note
The `LegacyStackSynthesizer` does not offer any customization properties.

Changing the qualifier

The *qualifier* is added to the name of bootstrap resources to distinguish the resources in separate bootstrap stacks. To deploy two different versions of the bootstrap stack in the same environment (AWS account and region), then, the stacks must have different qualifiers. This feature is intended for name isolation between automated tests of the CDK itself. Unless you can very precisely scope down the IAM permissions given to the AWS CloudFormation execution role, there are no privilege isolation benefits to having two different bootstrap stacks in a single account, so there is usually no need to change this value.

To change the qualifier, configure the `DefaultStackSynthesizer` either by instantiating the synthesizer with the property:

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"
    })
});
```

Or by configuring the qualifier as a context key in `cdk.json`.

```
{
```

```
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "MYQUALIFIER"
  }
}
```

Changing the resource names

All the other `DefaultStackSynthesizer` properties relate to the names of the resources in the modern bootstrapping template. You only need to provide any of these properties if you modified the bootstrap template and changed the resource names or naming scheme.

All properties accept the special placeholders `${Qualifier}`, `${AWS::Partition}`, `${AWS::AccountId}`, and `${AWS::Region}`. These placeholders are replaced with the values of the `qualifier` parameter and with the values of the AWS partition, account ID, and region for the stack's environment, respectively.

The following example shows all the available properties for `DefaultStackSynthesizer` along with their default values, as if you were instantiating the synthesizer.

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 deploy 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 deploy 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}',

  // 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 deploy 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 deploy 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}',

  // 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 deploy role)
  file_sset_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 deploy 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}"

  // 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 deploy 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 deploy 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}")

    // 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 deploy 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 deploy 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}"

    // 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,
})
```

The bootstrapping template contract

The requirements of the bootstrapping stack depend on the stack synthesizer in use. If you write your own stack synthesizer, you have complete control of the bootstrap resources that your synthesizer requires and how the synthesizer finds them. This section describes the expectations that the `DefaultStackSynthesizer` has of the bootstrapping template.

Versioning

The template should contain a resource to create an SSM parameter with a well-known name and an output to reflect the template's version.

```
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]
```

Roles

The `DefaultStackSynthesizer` requires five IAM roles for five different purposes. If you are not using the default roles, the synthesizer needs to be told the ARNs for the roles you want to use. The roles are:

- The *deployment role* is assumed by the AWS CDK Toolkit and by AWS CodePipeline to deploy into an environment. Its `AssumeRolePolicy` controls who can deploy into the environment. The permissions this role needs can be seen in the template.
- The *lookup role* is assumed by the AWS CDK Toolkit to perform context lookups in an environment. Its `AssumeRolePolicy` controls who can deploy into the environment. The permissions this role needs can be seen in the template.
- The *file publishing role* and the *image publishing role* are assumed by the AWS CDK Toolkit and by AWS CodeBuild projects to publish assets into an environment: that is, to write to the S3 bucket and the ECR repository, respectively. These roles require write access to these resources.
- *The AWS CloudFormation execution role* is passed to AWS CloudFormation to perform the actual deployment. Its permissions are the permissions that the deployment will execute under. The permissions are passed to the stack as a parameter that lists managed policy ARNs.

Outputs

The AWS CDK Toolkit requires that the following CloudFormation outputs exist on the bootstrap stack.

- `BucketName`: the name of the file asset bucket
- `BucketDomainName`: the file asset bucket in domain name format
- `BootstrapVersion`: the current version of the bootstrap stack

Template history

The bootstrap template is versioned and evolves over time with the AWS CDK itself. If you provide your own bootstrap template, keep it up-to-date with the canonical default template to ensure that yours continues to work with all CDK features. This section contains a list of the changes made in each version.

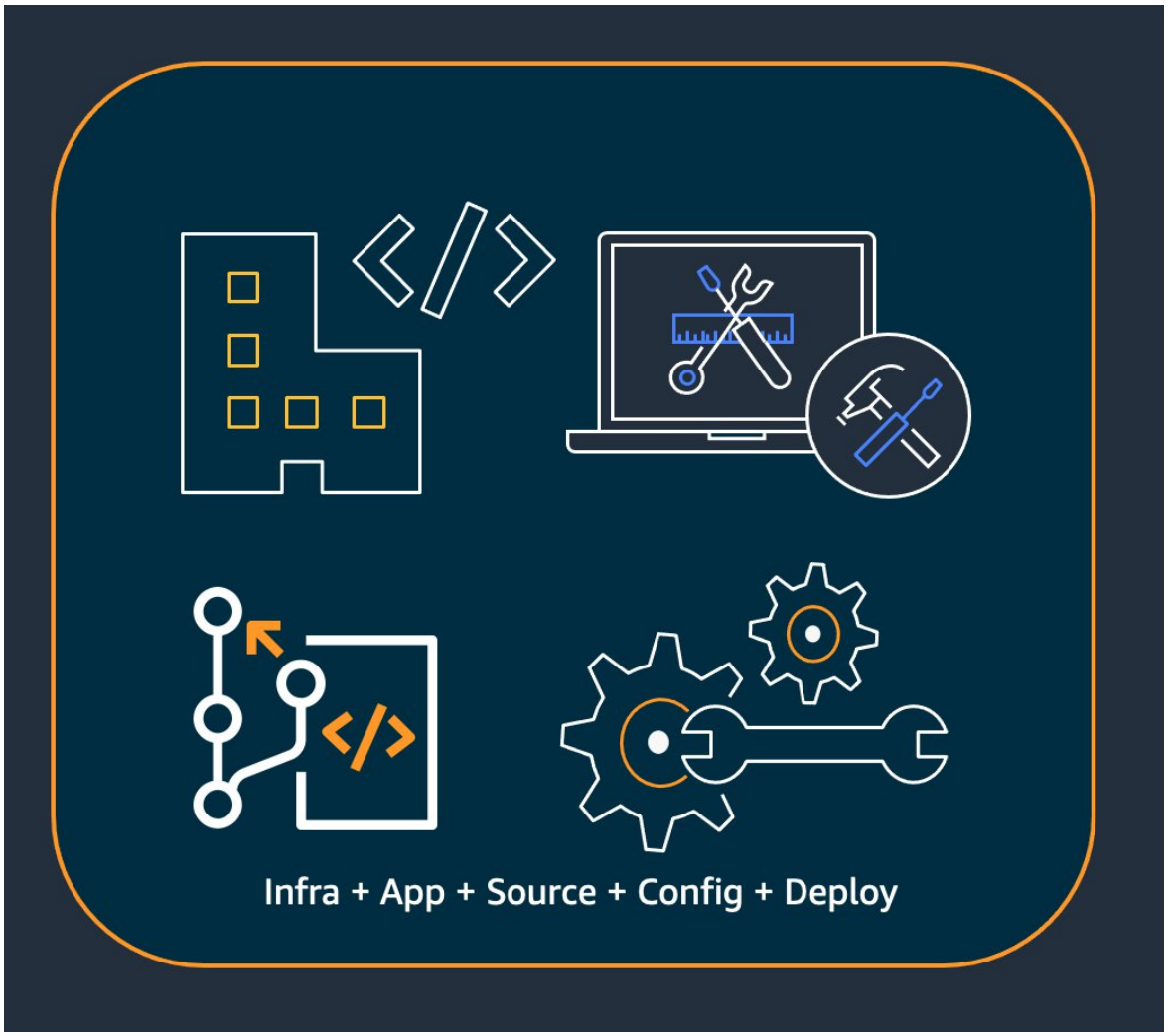
Template version	AWS CDK version	Changes
1	1.40.0	Initial version of template with Bucket, Key, Repository and Roles.
2	1.45.0	Split asset publishing role into separate file and image publishing roles.
3	1.46.0	Add `FileAssetKeyArn` export to be able to add decrypt permissions to asset consumers.
4	1.61.0	KMS permissions are now implicit via S3 and no longer require `FileAsetKeyArn`, Add `CdkBootstrapVersion` SSM parameter so the bootstrap stack version can be verified without knowing the stack name.
5	1.87.0	Deployment role can read SSM parameter.
6	1.108.0	Add lookup role separate from deployment role.
6	1.109.0	Attach `aws-cdk:bootstrap-role` tag to deployment, file publishing, and image publishing roles.
7	1.110.0	Deployment role can no longer read Buckets in the target account directly (however, this role is effectively an administrator, and could always use its AWS CloudFormation permissions to make the bucket readable anyway).

Template version	AWS CDK version	Changes
8	1.114.0	The lookup role has full read-only permissions to the target environment, and has a `aws-cdk:bootstrap-role` tag as well.

Best practices for developing and deploying cloud infrastructure with the AWS CDK

The AWS CDK allows developers or administrators to define their cloud infrastructure using a supported programming language. CDK applications should be organized into logical units, such as API, database, and monitoring resources, and optionally have a pipeline for automated deployments. The logical units should be implemented as constructs including the infrastructure (e.g. Amazon S3 buckets, Amazon RDS databases, Amazon VPC network), runtime code (e.g. AWS Lambda functions), and configuration code. Stacks define the deployment model of these logical units. For a more detailed introduction to the concepts behind the CDK, see *Getting started* (p. 8).

The AWS CDK reflects careful consideration of the needs of our customers and internal teams and of the failure patterns that often arise during the deployment and ongoing maintenance of complex cloud applications. We discovered that failures are often related to "out-of-band" changes to an application, such as configuration changes, that were not fully tested. Therefore, we developed the AWS CDK around a model in which your entire application, not just business logic but also infrastructure and configuration, is defined in code. That way, proposed changes can be carefully reviewed, comprehensively tested in environments resembling production to varying degrees, and fully rolled back if something goes wrong.



At deployment time, the AWS CDK synthesizes a cloud assembly that contains not only AWS CloudFormation templates describing your infrastructure in all target environments, but file assets containing your runtime code and their supporting files. With the CDK, every commit in your application's main version control branch can represent a complete, consistent, deployable version of your application. Your application can then be deployed automatically whenever a change is made.

The philosophy behind the AWS CDK leads to our recommended best practices, which we have divided into four broad categories.

> **Tip**
> In addition to the guidance in this document, you should also consider best practices for AWS CloudFormation as well as for the individual AWS services you use, where they are obviously applicable to CDK-defined infrastructure.

- the section called “Organization best practices” (p. 191)
- the section called “Coding best practices” (p. 191)
- the section called “Construct best practices” (p. 193)
- the section called “Application best practices” (p. 195)

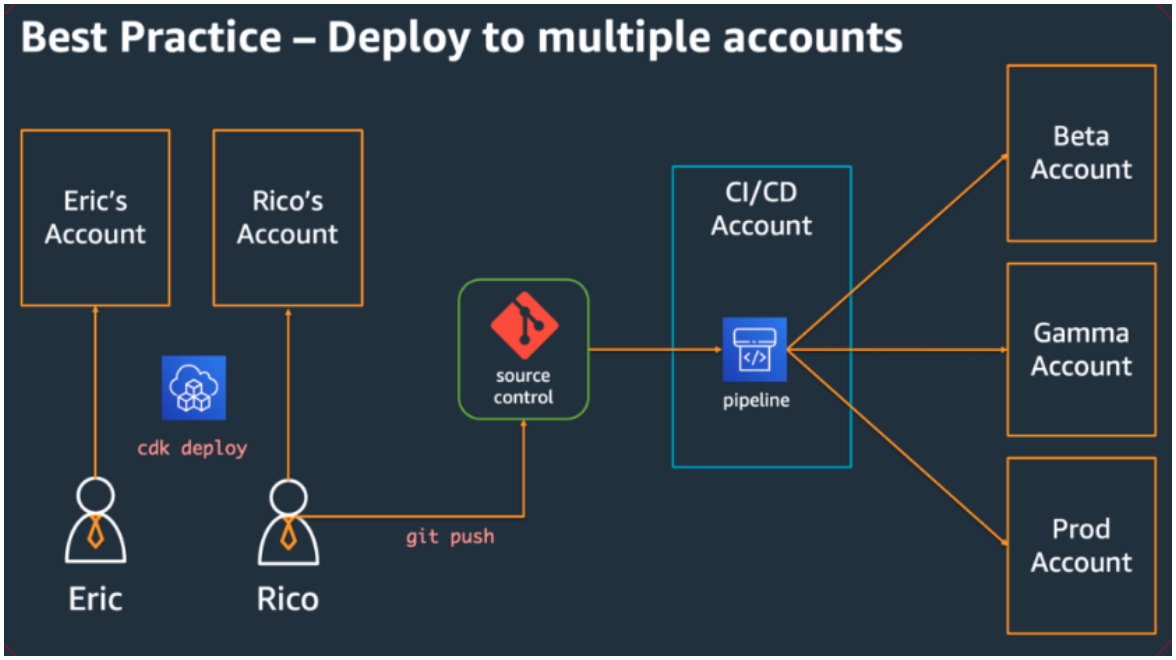
Organization best practices

In the beginning stages of AWS CDK adoption, it's important to consider how to set up your organization for success. It's a best practice to have a team of experts responsible for training and guiding the rest of the company as they adopt the CDK The size of this team may vary, from one or two people at a small company to a full-fledged Cloud Center of Excellence (CCoE) at a larger company. This team is responsible for setting standards and policies for how cloud infrastructure will be done at your company, as well as for training and mentoring developers.

The CCoE may provide guidance on what programming languages should be used for cloud infrastructure. The details will vary from one organization to the next, but a good policy helps make sure developers can easily understand and maintain all cloud infrastructure throughout the company.

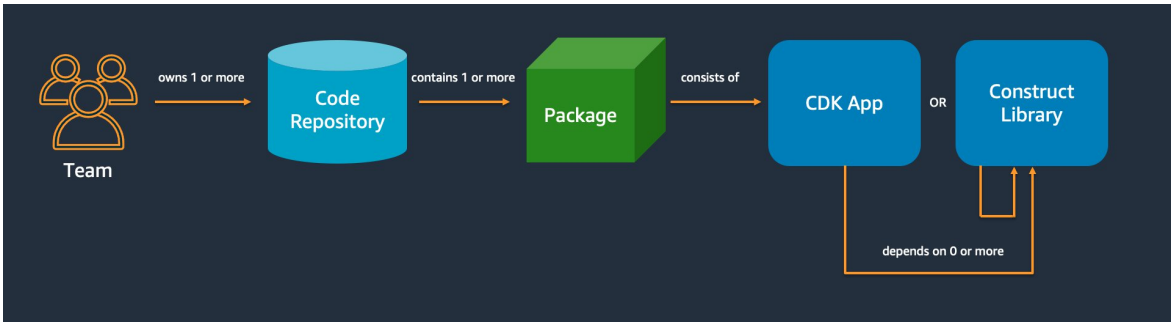
The CCoE also creates a "landing zone" that defines your organizational units within AWS. A landing zone is a pre-configured, secure, scalable, multi-account AWS environment based on best practice blueprints. You can tie together the services that make up your landing zone with AWS Control Tower, a high-level service configures and manages your entire multi-account system from a single user interface.

Development teams should be able use their own accounts for testing and have the ability to deploy new resources in these accounts as needed. Individual developers can treat these resources as extensions of their own development workstation. Using CDK Pipelines (p. 255), the AWS CDK applications can then be deployed via a CI/CD account to testing, integration, and production environments (each isolated in its own AWS region and/or account) by merging the developers' code into your organization's canonical repository.



Coding best practices

This section presents best practices for organizing your AWS CDK code. The diagram below shows the relationship between a team and that team's code repositories, packages, applications, and construct libraries.



Start simple and add complexity only when you need it

The guiding principle for most of our best practices is to keep things simple as possible—but no simpler. Add complexity only when your requirements dictate a more complicated solution. With the AWS CDK, you can always refactor your code as necessary to support new requirements, so it doesn't make sense to architect for all possible scenarios up front.

Align with the AWS Well-Architected framework

The AWS Well-Architected framework defines a *component* as the code, configuration, and AWS resources that together deliver against a requirement. A component is often the unit of technical ownership, and is decoupled from other components. The term *workload* is used to identify a set of components that together deliver business value. A workload is usually the level of detail that business and technology leaders communicate about.

An AWS CDK application maps to a component as defined by the AWS Well-Architected Framework. AWS CDK apps are a mechanism to codify and deliver Well-Architected cloud application best practices. You can also create and share components as reusable code libraries through artifact repositories, such as AWS CodeArtifact.

Every application starts with a single package in a single repository

A single package is the entry point of your AWS CDK app. This is where you define how and where the different logical units of your application are deployed, as well as the CI/CD pipeline to deploy the application. The app's constructs define the logical units of your solution.

Use additional packages for constructs that you use in more than one application. (Shared constructs should also have their own lifecycle and testing strategy.) Dependencies between packages in the same repository are managed by your repo's build tooling.

Though it is possible, it is not recommended to put multiple applications in the same repository, especially when using automated deployment pipelines, because this increases the "blast radius" of changes during deployment. With multiple applications in a repository, not only do changes to one application trigger deployment of the others (even if they have not changed), but a break in one application prevents the other applications from being deployed.

Move code into repositories based on code lifecycle or team ownership

When packages begin to be used in multiple applications, move them to their own repository, so they can be referenced by the build systems of the applications that use them, but updated on cadences independent of the lifecycles of those applications. It may make sense to put all shared constructs in one repository at first.

Also move packages to their own repo when different teams are working on them, to help enforce access control.

To consume packages across repository boundaries, you now need a private package repository—similar to NPM, PyPi, or Maven Central, but internal to your organization, and a release process that builds, tests, and publishes the package to the private package repository. CodeArtifact can host packages for most popular programming languages.

Dependencies on packages in the package repository are managed by your language's package manager, for example NPM for TypeScript or JavaScript applications. Your package manager helps to make sure builds are repeatable by recording the specific versions of every package your application depends on and allows you to upgrade those dependencies in a controlled manner.

Shared packages need a different testing strategy: although for a single application it might be good enough to deploy the application to a testing environment and confirm that it still works, shared packages need to be tested independently of the consuming application, as if they were being released to the public. (Your organization might in fact choose to actually release some shared packages to the public.)

Keep in mind that a construct can be arbitrarily simple or complex. A `Bucket` is a construct, but `CameraShopWebsite` could be a construct, too.

Infrastructure and runtime code live in the same package

The AWS CDK not only generates AWS CloudFormation templates for deploying infrastructure, it also bundles runtime assets like Lambda functions and Docker images and deploys them alongside your infrastructure. So it's not only possible to combine the code that defines your infrastructure and the code that implements your runtime logic into a single construct— it's a best practice. These two kinds of code don't need to live in separate repositories or even in separate packages.

A construct that is self-contained, in other words that completely describes a piece of functionality including its infrastructure and logic, makes it easy to evolve the two kinds of code together, test them in isolation, share and reuse the code across projects, and version all the code in sync.

Construct best practices

This section contains best practices for developing constructs. Constructs are reusable, composable modules that encapsulate resources, and the building blocks of AWS CDK apps.

Model your app through constructs, not stacks

When breaking down your application into logical units, represent each unit as a descendant of `Construct` and not of `Stack`. Stacks are a unit of deployment, and so tend to be oriented to specific

applications. By using constructs instead of stacks, you give yourself and your users the flexibility to build stacks in the way that makes the most sense for each deployment scenario. For example, you could compose multiple constructs into a `DevStack` with some configuration for development environments and then have a different composition for your `ProdStack`.

Configure with properties and methods, not environment variables

Environment variable lookups inside constructs and stacks are a common anti-pattern. Both constructs and stacks should accept a properties object to allow for full configurability completely in code. To do otherwise is to introduce a dependency on the machine that the code will run on, which becomes another bit of configuration information you have to keep track of and manage.

In general, environment variable lookups should be limited to the top level of an AWS CDK app, and should be used to pass in information needed for running in a development environment; see the section called “Environments” (p. 92).

Unit test your infrastructure

If you avoid network lookups during synthesis and model all your production stages in code (best practices we cover later), you can run a full suite of unit tests at build time, consistently, in all environments. If any single commit always results in the same generated template, you can trust the unit tests that you write to confirm that the generated templates look how you expect them to. See *Testing constructs* (p. 298).

Don't change the logical ID of stateful resources

Changing the logical ID of a resource results in the resource being replaced with a new one at the next deployment. For stateful resources like databases and buckets, or persistent infrastructure like an Amazon VPC, this is almost never what you want. Be careful about any refactor of your AWS CDK code that could cause the ID to change, and write unit tests that assert that the logical IDs of your stateful resources remain static. The logical ID is derived from the `id` you specify when you instantiate the construct, and the construct's position in the construct tree; see the section called “Logical IDs” (p. 121).

Constructs aren't enough for compliance

Many enterprise customers are writing their own wrappers for L2 constructs (the "curated" constructs that represent individual AWS resources with built-in sane defaults and best practices) to enforce security best practices such as static encryption and specific IAM policies. For example, you might create a `MyCompanyBucket` that you then use in your applications in place of the usual Amazon S3 `Bucket` construct. This pattern is useful for surfacing security guidance early in the software development lifecycle, but it cannot be relied on as the sole means of enforcement.

Instead, use AWS features such as service control policies and permission boundaries to enforce your security guardrails at the organization level. Use the section called “Aspects” (p. 165) or tools like CloudFormation Guard to make assertions about the security properties of infrastructure elements before deployment. Use AWS CDK for what it does best.

Finally, keep in mind that writing your own "L2+" constructs like these may prevent your developers from taking advantage of the growing ecosystems of AWS CDK packages, such as AWS Solutions Constructs, as these are typically built upon standard AWS CDK constructs and won't be able to use your custom versions.

Application best practices

In this section we discuss how best to write your AWS CDK applications, combining constructs to define how your AWS resources are connected.

Make decisions at synthesis time

Although AWS CloudFormation lets you make decisions at deployment time (using `Conditions`, `{ Fn::If }`, and `Parameters`), and the AWS CDK gives you some access to these mechanisms, we recommend against using them. The types of values you can use, and the types of operations you can perform on them, are quite limited compared to those available in a general-purpose programming language.

Instead, try to make all decisions, such as which construct to instantiate, in your AWS CDK application, using your programming language's `if` statements and other features. For example, a common CDK idiom, iterating over a list and instantiating a construct with values from each item in the list, simply isn't possible using AWS CloudFormation expressions.

Treat AWS CloudFormation as an implementation detail that the AWS CDK uses for robust cloud deployments, not as a language target. You're not writing AWS CloudFormation templates in TypeScript or Python, you're writing CDK code that happens to use CloudFormation for deployment.

Use generated resource names, not physical names

Names are a precious resource. Every name can only be used once, so if you hard-code a table name or bucket name into your infrastructure and application, you can't deploy that piece of infrastructure twice in the same account. (The name we're talking about here is the name specified by, for example, the `bucketName` property on an Amazon S3 bucket construct.)

What's worse, you can't make changes to the resource that require it to be replaced. If a property can only be set at resource creation, for example the `KeySchema` of an Amazon DynamoDB table, that property is immutable, and changing it requires a new resource. But the new resource must have the same name in order to be a true replacement, and it can't while the existing resource is still using that name.

A better approach is to specify as few names as possible. If you leave out resource names, the AWS CDK will generate them for you, and it'll do so in a way that won't cause these problems. You then, for example, pass the generated table name (which you can reference as `table.tableName` in your AWS CDK application) as an environment variable into your AWS Lambda function, or you generate a configuration file on your Amazon EC2 instance on startup, or you write the actual table name to AWS Systems Manager Parameter Store and your application reads it from there.

If the place you need it is another AWS CDK stack, that's even easier. Given one stack that defines the resource and another than needs to use it:

- If the two stacks are in the same AWS CDK app, just pass a reference between the two stacks. For example, save a reference to the resource's construct as an attribute of the defining stack (`this.stack.uploadBucket = myBucket`), then pass that attribute to the constructor of the stack that needs the resource.
- When the two stacks are in different AWS CDK apps, use a static `from` method to import an externally-defined resource based on its ARN, name, or other attributes (for example, `Table.fromarn()` for a DynamoDB table). Use the `CfnOutput` consturct to print the ARN or other required value in the output of `cdk deploy`, or look in the AWS console. Or the second app can parse the CloudFormation template generated by the first app and retrieve that value from the Outputs section.

Define removal policies and log retention

The AWS CDK does its best to keep you from losing data by defaulting to policies that retain everything you create. For example, the default removal policy on resources that contain data (such as Amazon S3 buckets and database tables) is to never delete the resource when it is removed from the stack, but rather orphan the resource from the stack. Similarly, the CDK's default is to retain all logs forever. In production environments, these defaults can quickly result in the storage of large amounts of data you don't actually need, and a corresponding AWS bill.

Consider carefully what you want these policies to actually be for each production resource and specify them accordingly. Use the section called “Aspects” (p. 165) to validate the removal and logging policies in your stack.

Separate your application into multiple stacks as dictated by deployment requirements

There is no hard and fast rule to how many stacks your application needs. You'll usually end up basing the decision on your deployment patterns. Keep in mind the following guidelines:

- It's typically easier to keep as many resources in the same stack as possible, so keep them together unless you know you want them separated.
- Consider keeping stateful resources (like databases) in a separate stack from stateless resources. You can then turn on termination protection on the stateful stack, and can freely destroy or create multiple copies of the stateless stack without risk of data loss.
- Stateful resources are more sensitive to construct renaming—renaming leads to resource replacement—so it makes sense not to nest them inside constructs that are likely to be moved around or renamed (unless the state can be rebuilt if lost, like a cache). This is another good reason to put stateful resources in their own stack.

Commit `cdk.context.json` to avoid non-deterministic behavior

Determinism is key to successful AWS CDK deployments. A AWS CDK app should have essentially the same result whenever it is deployed to a given environment.

Since your AWS CDK app is written in a a general-purpose programming language, it can execute arbitrary code, use arbitrary libraries, and make arbitrary network calls. For example, you could use an AWS SDK to retrieve some information from your AWS account while synthesizing your app. Recognize that doing so will result in additional credential setup requirements, increased latency, and a chance, however small, of failure every time you run `cdk synth`.

You should never modify your AWS account or resources during synthesis; synthesizing an app should not have side effects. Changes to your infrastructure should happen only in the deployment phase, after the AWS CloudFormation template has been generated. This way, if there's a problem, AWS CloudFormation will automatically roll back the change. To make changes that can't be easily made within the AWS CDK framework, use custom resources to execute arbitrary code at deployment time.

Even strictly read-only calls are not necessarily safe. Consider what happens if the value returned by a network call changes. What part of your infrastructure will that impact? What will happen to already-deployed resources? Here are just two of the situations in which a sudden change in values might cause a problem.

- If you provision an Amazon VPC to all available Availability Zones in a specified region, and the number of AZs is two on deployment day, your IP space gets split in half. If AWS launches a new

Availability Zone the next day, the next deployment after that tries to split your IP space into thirds, requiring all subnets to be recreated. This probably won't be possible because your Amazon EC2 instances are still running, and you'll have to clean this up manually.

- If you query for the latest Amazon Linux machine image and deploy an Amazon EC2 instance, and the next day a new image is released, a subsequent deployment picks up the new AMI and replaces all your instances. This may not be what you expected to happen.

These situations can be particularly pernicious because the AWS-side change may occur after months or years of successful deployments. Suddenly your deployments are failing "for no reason" and you long ago forgot what you did and why.

Fortunately, the AWS CDK includes a mechanism called *context providers* to record a snapshot of non-deterministic values, allowing future synthesis operations produce exactly the same template. The only changes in the new template are the changes *you* made in your code. When you use a construct's `.fromLookup()` method, the result of the call is cached in `cdk.context.json`, which you should commit to version control along with the rest of your code to ensure future executions of your CDK app use the same value. The CDK Toolkit includes commands to manage the context cache, so you can refresh specific entries when you need to. For more information, see the section called “Context” (p. 159).

If you need some value (from AWS or elsewhere) for which there is no native CDK context provider, we recommend writing a separate script to retrieve the value and write it to a file, then read that file in your CDK app. Run the script only when you want to refresh the stored value, not as part of your regular build process.

Let the AWS CDK manage roles and security groups

One of the great features of the AWS CDK construct library is the `grant()` convenience methods that allow quick and simple creation of AWS Identity and Access Management roles granting access to one resource by another using minimally-scoped permissions. For example, consider a line like the following:

```
myBucket.grantRead(myLambda)
```

This single line results in a policy being added to the Lambda function's role (which is also created for you). That role and its policies are more than a dozen lines of CloudFormation that you don't have to write, and the AWS CDK grants only the minimal permissions required for the function to read from the bucket.

If you require developers to always use predefined roles that were created by a security team, AWS CDK coding becomes much more complicated, and your teams lose a lot of flexibility in how they design their applications. A better alternative is to use service control policies and permission boundaries to ensure that developers stay within the guardrails.

Model all production stages in code

In traditional AWS CloudFormation scenarios, your goal is to produce a single artifact that is parameterized so that it can be deployed to various target environments after applying configuration values specific to those environments. In the CDK, you can, and should, build that configuration right into your source code. Create a stack for your production environment, and a separate one for each of your other stages, and put the configuration values for each right there in the code. Use services like Secrets Manager and Systems Manager Parameter Store for sensitive values that you don't want to check in to source control, using the names or ARNs of those resources.

When you synthesize your application, the cloud assembly created in the `cdk.out` folder contains a separate template for each environment. Your entire build is deterministic: there are no out-of-band changes to your application, and any given commit always yields the exact same AWS CloudFormation template and accompanying assets, which makes unit testing much more reliable.

Measure everything

Achieving the goal of full continuous deployment, with no human intervention, requires a high level of automation, and that automation isn't possible without extensive amounts of monitoring. Create metrics, alarms, and dashboards to measure all aspects of your deployed resources. And don't just measure simple things like CPU usage and disk space: also record your business metrics, and use those measurements to automate deployment decisions like rollbacks. Most of the L2 constructs in AWS CDK have convenience methods to help you create metrics, such as the `metricUserErrors()` method on the dynamodb.Table class.

API reference

The API Reference contains information about the AWS CDK libraries.

Each library contains information about how to use the library. For example, the S3 library demonstrates how to set default encryption on an Amazon S3 bucket.

Versioning

Version numbers consist of three numeric version parts: *major.minor.patch*, and generally adhere to the semantic versioning model. This means that breaking changes to stable APIs are limited to major releases. Minor and patch releases are backward compatible, meaning that the code written in a previous version with the same major version can be upgraded to a newer version and be expected to continue to build and run, producing the same output.

> **Note**
> This compatibility promise does not apply to APIs under active development, which are designated as experimental. See the section called "AWS CDK stability index" (p. 199) for more details.

AWS CDK Toolkit (CLI) compatibility

The AWS CDK Toolkit (that is, the `cdk` command line command) is *always* compatible with construct libraries of a semantically *lower* or *equal* version number. It is, therefore, always safe to upgrade the AWS CDK Toolkit within the same major version.

The AWS CDK Toolkit may be, but is *not always*, compatible with construct libraries of a semantically *higher* version, depending on whether the same cloud assembly schema version is employed by the two components. The AWS CDK framework generates a cloud assembly during synthesis; the AWS CDK Toolkit consumes it for deployment. The schema that defines the format of the cloud assembly is strictly specified and versioned. AWS construct libraries using a given cloud assembly schema version are compatible with AWS CDK toolkit versions using that schema version or later, which may include releases of the AWS CDK Toolkit *older than* a given construct library release.

When the cloud assembly version required by the construct library is not compatible with the version supported by the AWS CDK Toolkit, you receive an error message like this one.

```
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.
```

To resolve this error, update the AWS CDK Toolkit to a version compatible with the required cloud assembly version, or simply to the latest available version. The alternative (downgrading the construct library modules your app uses) is generally not desirable.

> **Note**
> For more details on the cloud assembly schema, see Cloud Assembly Versioning.

AWS CDK stability index

The modules in the AWS Construct Library move through various stages as they are developed from concept to mature API. Different stages imply different promises for API stability in subsequent versions of the AWS CDK.

Stage 0: CFN resources

All construct library modules start in stage 0 when they are auto-generated from the AWS CloudFormation resource specification. The goal of stage 0 is to make new AWS CloudFormation resources/properties available to AWS CDK customers as quickly as possible. We capture feedback from customers to better understand what L2 resources to add.

AWS CloudFormation resources themselves are considered stable APIs, regardless of whether other constructs in the module are under active development.

Stage 1: Experimental

The goal of the experimental stage is to retain the freedom to make breaking changes to APIs while we design and build a module. During this stage, the primary use cases and the set of L2 constructs required to support them are incrementally identified, implemented, and validated.

Development of L2 constructs is community-oriented and transparent. For large and/or complex changes, we author a Request for Comments (RFC) that outlines our intended design and publish it for feedback. We also use pull requests to conduct API design reviews.

At this stage, individual APIs may be in flux, and breaking changes may occur from release to release if we deem these necessary to support customer use cases.

Stage 2: Developer preview (DP)

At the developer preview stage, our aim is to deliver a release candidate with a stable API with which to conduct user acceptance testing. When the API passes acceptance, it is deemed suitable for general availability.

We make breaking changes at this stage only when required to address unforeseen customer use cases or issues. Since breaking changes are still possible, the package itself retains the "experimental" label while in developer preview.

Stage 3: General availability (GA)

The module is generally available with a compatibility guarantee across minor versions. We will only make backward-compatible changes to the API, so that your existing apps will continue to work until the next major AWS CDK release.

In some cases, we may use feature flags (p. 164) to optionally enable new behavior while retaining the previous behavior to support existing apps.

Each module's Overview in the API Reference describes its stability level.

For more information about these maturity stages, see AWS Construct Library Module Lifecycle.

Language binding stability

From time to time, we may add support to the AWS CDK for additional programming languages. Although the API described in all the languages is the same, the way that API is expressed varies by language and may change as the language support evolves. For this reason, language bindings are deemed experimental for a time until they are considered ready for production use. Currently, all supported languages are considered stable.

Language	Stability
TypeScript	Stable
JavaScript	Stable

Language	Stability
Python	Stable
Java	Stable
C#/.NET	Stable
Go	Experimental

Examples

This topic contains the following examples:

- Creating a serverless application using the AWS CDK (p. 202) Creates a serverless application using Lambda, API Gateway, and Amazon S3.
- Creating an AWS Fargate service using the AWS CDK (p. 215) Creates an Amazon ECS Fargate service from an image on DockerHub.

Creating a serverless application using the AWS CDK

This example walks you through creating the resources for a simple widget dispensing service. (For the purpose of this example, a widget is just a name or identifier that can be added to, retrieved from, and deleted from a collection.) The example includes:

- An AWS Lambda function.
- An Amazon API Gateway API to call the Lambda function.
- An Amazon S3 bucket that holds the widgets.

This tutorial contains the following steps.

1. Create a AWS CDK app
2. Create a Lambda function that gets a list of widgets with **HTTP GET /**
3. Create the service that calls the Lambda function
4. Add the service to the AWS CDK app
5. Test the app
6. Add Lambda functions to do the following:
   - Create a widget with **POST /{name}**
   - Get a widget by name with **GET /{name}**
   - Delete a widget by name with **DELETE /{name}**
7. Tear everything down when you're finished

Create a AWS CDK app

Create the app **MyWidgetService** in the current folder.

TypeScript

```
mkdir MyWidgetService
cd MyWidgetService
cdk init --language typescript
```

JavaScript

```
mkdir MyWidgetService
```

```
cd MyWidgetService
cdk init --language javascript
```

Python

```
mkdir MyWidgetService
cd MyWidgetService
cdk init --language python
source .venv/bin/activate
pip install -r requirements.txt
```

Java

```
mkdir MyWidgetService
cd MyWidgetService
cdk init --language java
```

You may now import the Maven project into your IDE.

C#

```
mkdir MyWidgetService
cd MyWidgetService
cdk init --language csharp
```

You may now open `src/MyWidgetService.sln` in Visual Studio.

Note
The CDK names source files and classes based on the name of the project directory. If you don't use the name `MyWidgetService` as shown above, you'll have trouble following the rest of the steps because some of the files the instructions tell you to modify aren't there (they'll have different names).

The important files in the blank project are as follows. (We will also be adding a couple of new files.)

TypeScript

- `bin/my_widget_service.ts` – Main entry point for the application
- `lib/my_widget_service-stack.ts` – Defines the widget service stack

JavaScript

- `bin/my_widget_service.js` – Main entry point for the application
- `lib/my_widget_service-stack.js` – Defines the widget service stack

Python

- `app.py` – Main entry point for the application
- `my_widget_service/my_widget_service_stack.py` – Defines the widget service stack

Java

- `src/main/java/com/myorg/MyWidgetServiceApp.java` – Main entry point for the application

- `src/main/java/com/myorg/MyWidgetServiceStack.java` – Defines the widget service stack

C#

- `src/MyWidgetService/Program.cs` – Main entry point for the application
- `src/MyWidgetService/MyWidgetServiceStack.cs` – Defines the widget service stack

Run the app and note that it synthesizes an empty stack.

```
cdk synth
```

You should see output beginning with YAML code like the following.

```
Resources:
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Modules: "..."
```

Create a Lambda function to list all widgets

The next step is to create a Lambda function to list all of the widgets in our Amazon S3 bucket. We will provide the Lambda function's code in JavaScript.

Create the `resources` directory in the project's main directory.

```
mkdir resources
```

Create the following JavaScript file, `widgets.js`, in the `resources` directory.

```
/*
This code uses callbacks to handle asynchronous function responses.
It currently demonstrates using an async-await pattern.
AWS supports both the async-await and promises patterns.
For more information, see the following:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/calling-services-
asynchronously.html
https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html
*/
const AWS = require('aws-sdk');
const S3 = new AWS.S3();

const bucketName = process.env.BUCKET;

exports.main = async function(event, context) {
  try {
    var method = event.httpMethod;

    if (method === "GET") {
      if (event.path === "/") {
        const data = await S3.listObjectsV2({ Bucket: bucketName }).promise();
        var body = {
          widgets: data.Contents.map(function(e) { return e.Key })
        };
```

```
        return {
          statusCode: 200,
          headers: {},
          body: JSON.stringify(body)
        };
      }
    }

    // We only accept GET for now
    return {
      statusCode: 400,
      headers: {},
      body: "We only accept GET /"
    };
  } catch(error) {
    var body = error.stack || JSON.stringify(error, null, 2);
    return {
      statusCode: 400,
        headers: {},
        body: JSON.stringify(body)
    }
  }
}
```

Save it and be sure the project still results in an empty stack. We haven't yet wired the Lambda function to the AWS CDK app, so the Lambda asset doesn't appear in the output.

```
cdk synth
```

Creating a widget service

Add the API Gateway, Lambda, and Amazon S3 packages to the app.

TypeScript

```
npm install @aws-cdk/aws-apigateway @aws-cdk/aws-lambda @aws-cdk/aws-s3
```

JavaScript

```
npm install @aws-cdk/aws-apigateway @aws-cdk/aws-lambda @aws-cdk/aws-s3
```

Python

```
pip install aws_cdk.aws_apigateway aws_cdk.aws_lambda aws_cdk.aws_s3
```

Java

Add the following dependencies in the `dependencies` element of your project's `pom.xml` file.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>apigateway</artifactId>
    <version>${cdk.version}</version>
</dependency>
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>lambda</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

We recommend editing `pom.xml` directory rather than using your IDE's dependency management tools to make sure the versions of all AWS CDK libraries are synchronized using the `$cdk.version` variable.

C#

Choose **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio and add the following packages.

```
Amazon.CDK.AWS.ApiGateway
Amazon.CDK.AWS.Lambda
Amazon.CDK.AWS.S3
```

Tip
If you don't see these packages in the **Browse** tab of the **Manage Packages for Solution** page, make sure the **Include prerelease** checkbox is ticked.
For a better experience, also add the `Amazon.Jsii.Analyzers` package to provide compile-time checks for missing required properties.

Create a new source file to define the widget service with the source code shown below.

TypeScript

File: `lib/widget_service.ts`

```
import * as core from "@aws-cdk/core";
import * as apigateway from "@aws-cdk/aws-apigateway";
import * as lambda from "@aws-cdk/aws-lambda";
import * as s3 from "@aws-cdk/aws-s3";

export class WidgetService extends core.Construct {
  constructor(scope: core.Construct, id: string) {
    super(scope, id);

    const bucket = new s3.Bucket(this, "WidgetStore");

    const handler = new lambda.Function(this, "WidgetHandler", {
      runtime: lambda.Runtime.NODEJS_10_X, // So we can use async in widget.js
      code: lambda.Code.fromAsset("resources"),
      handler: "widgets.main",
      environment: {
        BUCKET: bucket.bucketName
      }
    });

    bucket.grantReadWrite(handler); // was: handler.role);

    const api = new apigateway.RestApi(this, "widgets-api", {
      restApiName: "Widget Service",
      description: "This service serves widgets."
    });

    const getWidgetsIntegration = new apigateway.LambdaIntegration(handler, {
      requestTemplates: { "application/json": '{ "statusCode": "200" }' }
    });
```

```
    api.root.addMethod("GET", getWidgetsIntegration); // GET /
  }
}
```

JavaScript

File: `lib/widget_service.js`

```
const core = require("@aws-cdk/core");
const apigateway = require("@aws-cdk/aws-apigateway");
const lambda = require("@aws-cdk/aws-lambda");
const s3 = require("@aws-cdk/aws-s3");

class WidgetService extends core.Construct {
  constructor(scope, id) {
    super(scope, id);

    const bucket = new s3.Bucket(this, "WidgetStore");

    const handler = new lambda.Function(this, "WidgetHandler", {
      runtime: lambda.Runtime.NODEJS_10_X, // So we can use async in widget.js
      code: lambda.Code.fromAsset("resources"),
      handler: "widgets.main",
      environment: {
        BUCKET: bucket.bucketName
      }
    });

    bucket.grantReadWrite(handler); // was: handler.role);

    const api = new apigateway.RestApi(this, "widgets-api", {
      restApiName: "Widget Service",
      description: "This service serves widgets."
    });

    const getWidgetsIntegration = new apigateway.LambdaIntegration(handler, {
      requestTemplates: { "application/json": '{ "statusCode": "200" }' }
    });

    api.root.addMethod("GET", getWidgetsIntegration); // GET /
  }
}

module.exports = { WidgetService }
```

Python

File: `my_widget_service/widget_service.py`

```
from aws_cdk import (core,
                     aws_apigateway as apigateway,
                     aws_s3 as s3,
                     aws_lambda as lambda_)

class WidgetService(core.Construct):
    def __init__(self, scope: core.Construct, id: str):
        super().__init__(scope, id)

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

        handler = lambda_.Function(self, "WidgetHandler",
                    runtime=lambda_.Runtime.NODEJS_10_X,
                    code=lambda_.Code.from_asset("resources"),
                    handler="widgets.main",
```

```
                    environment=dict(
                    BUCKET=bucket.bucket_name)
                    )

        bucket.grant_read_write(handler)

        api = apigateway.RestApi(self, "widgets-api",
                  rest_api_name="Widget Service",
                  description="This service serves widgets.")

        get_widgets_integration = apigateway.LambdaIntegration(handler,
                request_templates={"application/json": '{ "statusCode": "200" }'})

        api.root.add_method("GET", get_widgets_integration)   # GET /
```

Java

File: src/src/main/java/com/myorg/WidgetService.java

```
package com.myorg;

import java.util.HashMap;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.services.apigateway.LambdaIntegration;
import software.amazon.awscdk.services.apigateway.Resource;
import software.amazon.awscdk.services.apigateway.RestApi;
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.s3.Bucket;

public class WidgetService extends Construct {

    @SuppressWarnings("serial")
    public WidgetService(Construct scope, String id) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "WidgetStore");

        Function handler = Function.Builder.create(this, "WidgetHandler")
            .runtime(Runtime.NODEJS_10_X)
            .code(Code.fromAsset("resources"))
            .handler("widgets.main")
            .environment(new HashMap<String, String>() {{
               put("BUCKET", bucket.getBucketName());
            }}).build();

        bucket.grantReadWrite(handler);

        RestApi api = RestApi.Builder.create(this, "Widgets-API")
                .restApiName("Widget Service").description("This service services
 widgets.")
                .build();

        LambdaIntegration getWidgetsIntegration =
 LambdaIntegration.Builder.create(handler)
                .requestTemplates(new HashMap<String, String>() {{
                    put("application/json", "{ \"statusCode\": \"200\" }");
                }}).build();

        api.getRoot().addMethod("GET", getWidgetsIntegration);
    }
}
```

C#

File: `src/MyWidgetService/WidgetService.cs`

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3;
using System.Collections.Generic;

namespace MyWidgetService
{

    public class WidgetService : Construct
    {
        public WidgetService(Construct scope, string id) : base(scope, id)
        {
            var bucket = new Bucket(this, "WidgetStore");

            var handler = new Function(this, "WidgetHandler", new FunctionProps
            {
                Runtime = Runtime.NODEJS_10_X,
                Code = Code.FromAsset("resources"),
                Handler = "widgets.main",
                Environment = new Dictionary<string, string>
                {
                    ["BUCKET"] = bucket.BucketName
                }
            });

            bucket.GrantReadWrite(handler);

            var api = new RestApi(this, "Widgets-API", new RestApiProps
            {
                RestApiName = "Widget Service",
                Description = "This service services widgets."
            });

            var getWidgetsIntegration = new LambdaIntegration(handler, new
 LambdaIntegrationOptions
            {
                RequestTemplates = new Dictionary<string, string>
                {
                    ["application/json"] = "{ \"statusCode\": \"200\" }"
                }
            });

            api.Root.AddMethod("GET", getWidgetsIntegration);

        }
    }
}
```

Tip
We're using a `lambda.Function` in to deploy this function because it supports a wide variety of programming languages. For JavaScript and TypeScript specifically, you might consider a `lambda-nodejs.NodejsFunction`. The latter uses **esbuild** to bundle up the script and converts code written in TypeScript automatically.

Save the app and make sure it still synthesizes an empty stack.

```
cdk synth
```

Add the service to the app

To add the widget service to our AWS CDK app, we'll need to modify the source file that defines the stack to instantiate the service construct.

TypeScript

File: `lib/my_widget_service-stack.ts`

Add the following line of code after the existing `import` statement.

```
import * as widget_service from '../lib/widget_service';
```

Replace the comment in the constructor with the following line of code.

```
    new widget_service.WidgetService(this, 'Widgets');
```

JavaScript

File: `lib/my_widget_service-stack.js`

Add the following line of code after the existing `require()` line.

```
const widget_service = require('../lib/widget_service');
```

Replace the comment in the constructor with the following line of code.

```
    new widget_service.WidgetService(this, 'Widgets');
```

Python

File: `my_widget_service/my_widget_service_stack.py`

Add the following line of code after the existing `import` statement.

```
from . import widget_service
```

Replace the comment in the constructor with the following line of code.

```
        widget_service.WidgetService(self, "Widgets")
```

Java

File: `src/src/main/java/com/myorg/MyWidgetServiceStack.java`

Replace the comment in the constructor with the following line of code.

```
new WidgetService(this, "Widgets");
```

C#

File: `src/MyWidgetService/MyWidgetServiceStack.cs`

Replace the comment in the constructor with the following line of code.

```
new WidgetService(this, "Widgets");
```

Be sure the app runs and synthesizes a stack (we won't show the stack here: it's over 250 lines).

```
cdk synth
```

Deploy and test the app

Before you can deploy your first AWS CDK app containing a lambda function, you must bootstrap your AWS environment. This creates a staging bucket that the AWS CDK uses to deploy stacks containing assets. For details, see the section called "Bootstrapping your AWS environment" (p. 284). If you've already bootstrapped, you'll get a warning and nothing will change.

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

Now we're ready to deploy the app as follows.

```
cdk deploy
```

If the deployment succeeds, save the URL for your server. This URL appears in one of the last lines in the window, where *GUID* is an alphanumeric GUID and *REGION* is your AWS Region.

```
https://GUID.execute-api-REGION.amazonaws.com/prod/
```

Test your app by getting the list of widgets (currently empty) by navigating to this URL in a browser, or use the following command.

```
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
```

You can also test the app by:

1. Opening the AWS Management Console.
2. Navigating to the API Gateway service.
3. Finding **Widget Service** in the list.
4. Selecting **GET** and **Test** to test the function.

Because we haven't stored any widgets yet, the output should be similar to the following.

```
{ "widgets": [] }
```

Add the individual widget functions

The next step is to create Lambda functions to create, show, and delete individual widgets.

Replace the code in `widgets.js` (in `resources`) with the following.

```
const AWS = require('aws-sdk');
const S3 = new AWS.S3();

const bucketName = process.env.BUCKET;

/*
This code uses callbacks to handle asynchronous function responses.
It currently demonstrates using an async-await pattern.
```

```
AWS supports both the async-await and promises patterns.
For more information, see the following:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/calling-services-
asynchronously.html
https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html
*/
exports.main = async function(event, context) {
  try {
    var method = event.httpMethod;
    // Get name, if present
    var widgetName = event.path.startsWith('/') ? event.path.substring(1) : event.path;

    if (method === "GET") {
      // GET / to get the names of all widgets
      if (event.path === "/") {
        const data = await S3.listObjectsV2({ Bucket: bucketName }).promise();
        var body = {
          widgets: data.Contents.map(function(e) { return e.Key })
        };
        return {
          statusCode: 200,
          headers: {},
          body: JSON.stringify(body)
        };
      }

      if (widgetName) {
        // GET /name to get info on widget name
        const data = await S3.getObject({ Bucket: bucketName, Key: widgetName}).promise();
        var body = data.Body.toString('utf-8');

        return {
          statusCode: 200,
          headers: {},
          body: JSON.stringify(body)
        };
      }
    }

    if (method === "POST") {
      // POST /name
      // Return error if we do not have a name
      if (!widgetName) {
        return {
          statusCode: 400,
          headers: {},
          body: "Widget name missing"
        };
      }

      // Create some dummy data to populate object
      const now = new Date();
      var data = widgetName + " created: " + now;

      var base64data = new Buffer(data, 'binary');

      await S3.putObject({
        Bucket: bucketName,
        Key: widgetName,
        Body: base64data,
        ContentType: 'application/json'
      }).promise();

      return {
```

```
        statusCode: 200,
        headers: {},
        body: JSON.stringify(event.widgets)
      };
    }

    if (method === "DELETE") {
      // DELETE /name
      // Return an error if we do not have a name
      if (!widgetName) {
        return {
          statusCode: 400,
          headers: {},
          body: "Widget name missing"
        };
      }

      await S3.deleteObject({
        Bucket: bucketName, Key: widgetName
      }).promise();

      return {
        statusCode: 200,
        headers: {},
        body: "Successfully deleted widget " + widgetName
      };
    }

    // We got something besides a GET, POST, or DELETE
    return {
      statusCode: 400,
      headers: {},
      body: "We only accept GET, POST, and DELETE, not " + method
    };
  } catch(error) {
    var body = error.stack || JSON.stringify(error, null, 2);
    return {
      statusCode: 400,
      headers: {},
      body: body
    }
  }
}
```

Wire up these functions to your API Gateway code at the end of the `WidgetService` constructor.

TypeScript

File: `lib/widget_service.ts`

```
    const widget = api.root.addResource("{id}");

    // Add new widget to bucket with: POST /{id}
    const postWidgetIntegration = new apigateway.LambdaIntegration(handler);

    // Get a specific widget from bucket with: GET /{id}
    const getWidgetIntegration = new apigateway.LambdaIntegration(handler);

    // Remove a specific widget from the bucket with: DELETE /{id}
    const deleteWidgetIntegration = new apigateway.LambdaIntegration(handler);

    widget.addMethod("POST", postWidgetIntegration); // POST /{id}
    widget.addMethod("GET", getWidgetIntegration); // GET /{id}
    widget.addMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}
```

JavaScript

File: `lib/widget_service.js`

```
const widget = api.root.addResource("{id}");

// Add new widget to bucket with: POST /{id}
const postWidgetIntegration = new apigateway.LambdaIntegration(handler);

// Get a specific widget from bucket with: GET /{id}
const getWidgetIntegration = new apigateway.LambdaIntegration(handler);

// Remove a specific widget from the bucket with: DELETE /{id}
const deleteWidgetIntegration = new apigateway.LambdaIntegration(handler);

widget.addMethod("POST", postWidgetIntegration); // POST /{id}
widget.addMethod("GET", getWidgetIntegration); // GET /{id}
widget.addMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}
```

Python

File: `my_widget_service/widget_service.py`

```
        widget = api.root.add_resource("{id}")

        # Add new widget to bucket with: POST /{id}
        post_widget_integration = apigateway.LambdaIntegration(handler)

        # Get a specific widget from bucket with: GET /{id}
        get_widget_integration = apigateway.LambdaIntegration(handler)

        # Remove a specific widget from the bucket with: DELETE /{id}
        delete_widget_integration = apigateway.LambdaIntegration(handler)

        widget.add_method("POST", post_widget_integration);     # POST /{id}
        widget.add_method("GET", get_widget_integration);       # GET /{id}
        widget.add_method("DELETE", delete_widget_integration); # DELETE /{id}
```

Java

File: `src/src/main/java/com/myorg/WidgetService.java`

```
        Resource widget = api.getRoot().addResource("{id}");

        // Add new widget to bucket with: POST /{id}
        LambdaIntegration postWidgetIntegration = new LambdaIntegration(handler);

        // Get a specific widget from bucket with: GET /{id}
        LambdaIntegration getWidgetIntegration = new LambdaIntegration(handler);

        // Remove a specific widget from the bucket with: DELETE /{id}
        LambdaIntegration deleteWidgetIntegration = new LambdaIntegration(handler);

        widget.addMethod("POST", postWidgetIntegration);     // POST /{id}
        widget.addMethod("GET", getWidgetIntegration);       // GET /{id}
        widget.addMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}
```

C#

File: `src/MyWidgetService/WidgetService.cs`

```
            var widget = api.Root.AddResource("{id}");
```

```
            // Add new widget to bucket with: POST /{id}
            var postWidgetIntegration = new LambdaIntegration(handler);

            // Get a specific widget from bucket with: GET /{id}
            var getWidgetIntegration = new LambdaIntegration(handler);

            // Remove a specific widget from the bucket with: DELETE /{id}
            var deleteWidgetIntegration = new LambdaIntegration(handler);

            widget.AddMethod("POST", postWidgetIntegration);        // POST /{id}
            widget.AddMethod("GET", getWidgetIntegration);          // GET /{id}
            widget.AdMethod("DELETE", deleteWidgetIntegration);    // DELETE /{id}
```

Save and deploy the app.

```
cdk deploy
```

We can now store, show, or delete an individual widget. Use the following commands to list the widgets, create the widget **example**, list all of the widgets, show the contents of **example** (it should show today's date), delete **example**, and then show the list of widgets again.

```
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
curl -X POST 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X DELETE 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
```

You can also use the API Gateway console to test these functions. Set the **name** value to the name of a widget, such as **example**.

Clean up

To avoid unexpected AWS charges, destroy your AWS CDK stack after you're done with this exercise.

```
cdk destroy
```

Creating an AWS Fargate service using the AWS CDK

This example walks you through how to create an AWS Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster that's fronted by an internet-facing Application Load Balancer from an image on Amazon ECR.

Amazon ECS is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on a serverless infrastructure that's managed by Amazon ECS by launching your services or tasks using the Fargate launch type. For more control, you can host your tasks on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances that you manage by using the Amazon EC2 launch type.

This tutorial shows you how to launch some services using the Fargate launch type. If you've used the AWS Management Console to create a Fargate service, you know that there are many steps to follow

to accomplish that task. AWS has several tutorials and documentation topics that walk you through creating a Fargate service, including:

- How to Deploy Docker Containers - AWS
- Setting Up with Amazon ECS
- Getting Started with Amazon ECS Using Fargate

This example creates a similar Fargate service in AWS CDK code.

The Amazon ECS construct used in this tutorial helps you use AWS services by providing the following benefits:

- Automatically configures a load balancer.
- Automatically opens a security group for load balancers. This enables load balancers to communicate with instances without you explicitly creating a security group.
- Automatically orders dependency between the service and the load balancer attaching to a target group, where the AWS CDK enforces the correct order of creating the listener before an instance is created.
- Automatically configures user data on automatically scaling groups. This creates the correct configuration to associate a cluster to AMIs.
- Validates parameter combinations early. This exposes AWS CloudFormation issues earlier, thus saving you deployment time. For example, depending on the task, it's easy to misconfigure the memory settings. Previously, you would not encounter an error until you deployed your app. But now the AWS CDK can detect a misconfiguration and emit an error when you synthesize your app.
- Automatically adds permissions for Amazon Elastic Container Registry (Amazon ECR) if you use an image from Amazon ECR.
- Automatically scales. The AWS CDK supplies a method so you can autoscalinginstances when you use an Amazon EC2 cluster. This happens automatically when you use an instance in a Fargate cluster.

  In addition, the AWS CDK prevents an instance from being deleted when automatic scaling tries to kill an instance, but either a task is running or is scheduled on that instance.

  Previously, you had to create a Lambda function to have this functionality.
- Provides asset support, so that you can deploy a source from your machine to Amazon ECS in one step. Previously, to use an application source you had to perform several manual steps, such as uploading to Amazon ECR and creating a Docker image.

See ECS for details.

Creating the directory and initializing the AWS CDK

Let's start by creating a directory to hold the AWS CDK code, and then creating a AWS CDK app in that directory.

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
pip install -r requirements.txt
```

Java

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language java
```

You may now import the Maven project into your IDE.

C#

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language csharp
```

You may now open `src/MyEcsConstruct.sln` in Visual Studio.

Run the app and confirm that it creates an empty stack.

```
cdk synth
```

You should see a stack like the following, where *CDK-VERSION* is the version of the CDK and *NODE-VERSION* is the version of Node.js. (Your output may differ slightly from what's shown here.)

```
Resources:
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Modules: aws-cdk=CDK-VERSION,@aws-cdk/core=CDK-VERSION,@aws-cdk/cx-api=CDK-
VERSION,jsii-runtime=node.js/NODE-VERSION
```

Add the Amazon EC2 and Amazon ECS packages

Install the AWS construct library modules for Amazon EC2 and Amazon ECS.

TypeScript

```
npm install @aws-cdk/aws-ec2 @aws-cdk/aws-ecs @aws-cdk/aws-ecs-patterns
```

JavaScript

```
npm install @aws-cdk/aws-ec2 @aws-cdk/aws-ecs @aws-cdk/aws-ecs-patterns
```

Python

```
pip install aws_cdk.aws_ec2 aws_cdk.aws_ecs aws_cdk.aws_ecs_patterns
```

Java

Using your IDE's Maven integration (e.g., in Eclipse, right-click your project and choose **Maven** > **Add Dependency**), install the following artifacts from the group `software.amazon.awscdk`:

```
ec2
ecs
ecs-patterns
```

C#

Choose **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio and add the following packages.

```
Amazon.CDK.AWS.EC2
Amazon.CDK.AWS.ECS
Amazon.CDK.AWS.ECS.Patterns
```

Tip
If you don't see these packages in the **Browse** tab of the **Manage Packages for Solution** page, make sure the **Include prerelease** checkbox is ticked.
For a better experience, also add the `Amazon.Jsii.Analyzers` package to provide compile-time checks for missing required properties.

Create a Fargate service

There are two different ways to run your container tasks with Amazon ECS:

- Use the `Fargate` launch type, where Amazon ECS manages the physical machines that your containers are running on for you.
- Use the `EC2` launch type, where you do the managing, such as specifying automatic scaling.

For this example, we'll create a Fargate service running on an ECS cluster fronted by an internet-facing Application Load Balancer.

Add the following AWS Construct Library module imports to the indicated file.

TypeScript

File: `lib/my_ecs_construct-stack.ts`

```
import * as ec2 from "@aws-cdk/aws-ec2";
import * as ecs from "@aws-cdk/aws-ecs";
import * as ecs_patterns from "@aws-cdk/aws-ecs-patterns";
```

JavaScript

File: `lib/my_ecs_construct-stack.js`

```
const ec2 = require("@aws-cdk/aws-ec2");
const ecs = require("@aws-cdk/aws-ecs");
const ecs_patterns = require("@aws-cdk/aws-ecs-patterns");
```

Python

File: `my_ecs_construct/my_ecs_construct_stack.py`

```
from aws_cdk import (core, aws_ec2 as ec2, aws_ecs as ecs,
                     aws_ecs_patterns as ecs_patterns)
```

Java

File: `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#

File: `src/MyEcsConstruct/MyEcsConstructStack.cs`

```
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;
```

Replace the comment at the end of the constructor with the following code.

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 false
    });
```

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 false
    });
```

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 False
```

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 false
                      .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 false
            }
        );
```

Save it and make sure it runs and creates a stack.

```
cdk synth
```

The stack is hundreds of lines, so we don't show it here. The stack should contain one default instance, a private subnet and a public subnet for the three Availability Zones, and a security group.

Deploy the stack.

```
cdk deploy
```

AWS CloudFormation displays information about the dozens of steps that it takes as it deploys your app.

That's how easy it is to create a Fargate service to run a Docker image.

Clean up

To avoid unexpected AWS charges, destroy your AWS CDK stack after you're done with this exercise.

```
cdk destroy
```

AWS CDK examples

For more examples of AWS CDK stacks and apps in your favorite supported programming language, see the CDK Examples repository on GitHub.

AWS CDK how-tos

This section contains short code examples that show you how to accomplish a task using the AWS CDK.

Get a value from an environment variable

To get the value of an environment variable, use code like the following. This code gets the value of the environment variable `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";
```

Use an AWS CloudFormation parameter

See Parameters for information about using the optional *Parameters* section to customize your AWS CloudFormation templates.

You can also get a reference to a resource in an existing AWS CloudFormation template, as described in the section called “Import or migrate CloudFormation template” (p. 223).

Import or migrate an existing AWS CloudFormation template

The `cloudformation-include.CfnInclude` construct converts the resources in an imported AWS CloudFormation template to AWS CDK L1 constructs. You can work with these in your app just as if they were defined in AWS CDK code, even using them within higher-level AWS CDK constructs, letting you use (for example) the L2 permission grant methods with the resources they define.

This construct essentially adds an AWS CDK API wrapper to any resource in the template. You can use this capability to migrate your existing AWS CloudFormation templates to the AWS CDK a piece at a time in order to take advantage of the AWS CDK's convenient higher-level abstractions, or just to vend your AWS CloudFormation templates to AWS CDK developers by providing an AWS CDK construct API.

Note
The AWS CDK also includes `core.CfnInclude`, which was previously used for the same general purpose. However, `core.CfnInclude` lacks much of the functionality of `cloudformation-include.CfnInclude`. `core.CfnInclude` has been deprecated and will not be available in CDK 2.0.

Install the `cloudformation-include` module

Follow these instructions to install the `cloudformation-include` module.

TypeScript

```
npm install @aws-cdk/cloudformation-include
```

JavaScript

```
npm install @aws-cdk/cloudformation-include
```

Python

```
python -m pip install aws-cdk.cloudformation-include
```

Java

Add the following to the `<dependencies>` container of `pom.xml`.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>cdk-cloudformation-include</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

C#

Right-click the project in Visual Studio's Solution Explorer and choose **Manage NuGet Packages**. Search for and install the package `Amazon.CDK.CloudFormation.Include`. Or change to your project's directory and issue:

```
dotnet add package Amazon.CDK.CloudFormation.Include
```

Importing an AWS CloudFormation template

Here is a simple AWS CloudFormation template we'll use for the examples in this topic. Save it as `my-template.json`. After you've tried these examples with the provided template, you might explore further using a template for an actual stack you've already deployed, which you can obtain from the AWS CloudFormation console.

Tip
You can use either a JSON or YAML template. We recommend JSON if available, since YAML parsers can vary slightly in what they accept.

```
{
  "MyBucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": "MyBucket",
    }
  }
}
```

And here's how you import it into your stack using `cloudformation-include`.

TypeScript

```
import * as cdk from '@aws-cdk/core';
import * as cfninc from '@aws-cdk/cloudformation-include';

export class MyStack extends cdk.Stack {
  constructor(scope: cdk.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/core');
const cfninc = require('@aws-cdk/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

```
from aws_cdk import core
from aws_cdk import cloudformation_include as cfn_inc

class MyStack(core.Stack):

    def __init__(self, scope: core.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.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.cloudformation.include.CfnInclude;

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 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"
            });
        }
    }
```

```
}
```

By default, importing a resource preserves the resource's original logical ID from the template. This behavior is suitable for migrating an AWS CloudFormation template to the AWS CDK, where the logical IDs must be retained for AWS CloudFormation to recognize these as the same resources from the AWS CloudFormation template.

If you are instead developing an AWS CDK construct wrapper for the template so it can be used by AWS CDK developers ("vending"), have the AWS CDK generate new resource IDs instead, so the construct can be used multiple times in a stack without name conflicts. To do this, set the `preserveLogicalIds` property to false when importing the template.

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
});
```

To put the imported resources under the control of your AWS CDK app, add the stack to the `App` as usual.

TypeScript

```
import * as cdk from '@aws-cdk/core';
import { MyStack } from '../lib/my-stack';

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

JavaScript

```
const cdk = require('@aws-cdk/core');
const { MyStack } = require('../lib/my-stack');

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

Python

```
from aws_cdk import core
from mystack.my_stack import MyStack

app = core.App()
MyStack(app, "MyStack")
```

Java

```
import software.amazon.awscdk.core.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");
        }
    }
}
```

To verify that there will be no unintended changes to the AWS resources in the stack, perform a diff, omitting the AWS CDK-specific metadata.

```
cdk diff --no-version-reporting --no-path-metadata --no-asset-metadata
```

When you `cdk deploy` the stack, your AWS CDK app becomes the source of truth for the stack. Going forward, make changes to the AWS CDK app, not to the AWS CloudFormation template.

Accessing imported resources

The name `template` in the example code represents the imported AWS CloudFormation template. To access a resource from it, use this object's `getResource()` method. To access the returned resource as a specific kind of resource, cast the result to the desired type. (Casting is not necessary in Python and JavaScript.)

TypeScript

```
const cfnBucket = template.getResource('MyBucket') as s3.CfnBucket;
```

JavaScript

```
const cfnBucket = template.getResource('MyBucket') as s3.CfnBucket;
```

Python

```
cfn_bucket = template.get_resource("MyBucket")
```

Java

```
CfnBucket cfnBucket = (CfnBucket)template.getResource("MyBucket");
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("MyBucket");
```

In our example, `cfnBucket` is now an instance of the `aws-s3.CfnBucket` class, a L1 construct that exactly represents the corresponding AWS CloudFormation resource. You can treat it like any other resource of its type, for example getting its ARN by way of the `bucket.attrArn` property.

To wrap the L1 `CfnBucket` resource in a L2 `aws-s3.Bucket` instance instead, use the static methods `fromBucketArn()`, `fromBucketAttributes()`, or `fromBucketName()`. Usually the `fromBucketName()` method is the most convenient. For example:

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);
```

Other L2 constructs have similar methods for creating the construct from an existing resource.

Constructing the `Bucket` this way doesn't create a second Amazon S3 bucket; instead, the new `Bucket` instance encapsulates the existing `CfnBucket`.

In the example, `bucket` is now an L2 `Bucket` construct that you can use as you would one you declared yourself. For example, if `lambdaFunc` is an AWS Lambda function, and you wish to grant it write access to the bucket, you can do so using the bucket's convenient `grantWrite()` method, without needing to construct the necessary IAM policy yourself.

TypeScript

```
bucket.grantWrite(lambdaFunc);
```

JavaScript

```
bucket.grantWrite(lambdaFunc);
```

Python

```
bucket.grant_write(lambda_func)
```

Java

```
bucket.grantWrite(lambdaFunc);
```

C#

```
bucket.GrantWrite(lambdaFunc);
```

Replacing parameters

If your included AWS CloudFormation template has parameters, you can replace these with build-time values when you import the template, using the `parameters` property. In the example below, we replace the `UploadBucket` parameter with the ARN of a bucket defined elsewhere in our AWS CDK code.

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(new HashMap<String, String>() {{
   put("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 }
    }
});
```

Other template elements

You can import any AWS CloudFormation template element, not just resources. The imported elements become part of the AWS CDK stack. To import these elements, use the following methods of the `CfnInclude` object.

- `getCondition()` - AWS CloudFormation conditions
- `getHook()` - AWS CloudFormation hooks for blue-green deployments
- `getMapping()` - AWS CloudFormation mappings
- `getOutput()` - AWS CloudFormation outputs
- `getParameter()` - AWS CloudFormation parameters
- `getRule()` - AWS CloudFormation rules for Service Catalog templates

Each of these methods returns an instance of a class representing the specific type of AWS CloudFormation element. These objects are mutable; changes you make to them will appear in the template generated from the AWS CDK stack. The code below, for example, imports a parameter from the template and modifies its default.

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";
```

Nested stacks

You may import nested stacks by specifying them either when you import their main template, or at some later point. The nested template must be stored in a local file, but referenced as a `NestedStack` resource in the main template, and the resource name used in the AWS CDK code must match the name used for the nested stack in the main template.

Given this resource definition in the main template, the following code shows how to import the referenced nested stack both ways.

```
"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(new HashMap<String, CfnIncludeProps>() {{
  put("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'
});
```

You can import multiple nested stacks with either or both methods. When importing the main template, you provide a mapping between the resource name of each nested stack and its template file, and this mapping can contain any number of entries. To do it after the initial import, call `loadNestedStack()` once for each nested stack.

After importing a nested stack, you can access it using the main template's `getNestedStack()` method.

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;
```

The `getNestedStack()` method returns an `IncludedNestedStack` instance, from which you can access the AWS CDK `NestedStack` instance via the `stack` property (as shown in the example) or the original AWS CloudFormation template object via `includedTemplate`, from which you can load resources and other AWS CloudFormation elements.

Using resources from the AWS CloudFormation Public Registry

The AWS CloudFormation Public Registry is a collection of AWS CloudFormation extensions from both AWS and third parties that are available for use by all AWS customers. You can also publish your own extension for others to use. Extensions are of two types: resources and modules. You can use public resource extensions in your AWS CDK app using the `CfnResource` construct.

All public extensions published by AWS are available to all accounts in all regions without any action on your part. On the other hand, you must activate each third-party extension you want to use, in each account and region where you want to use it.

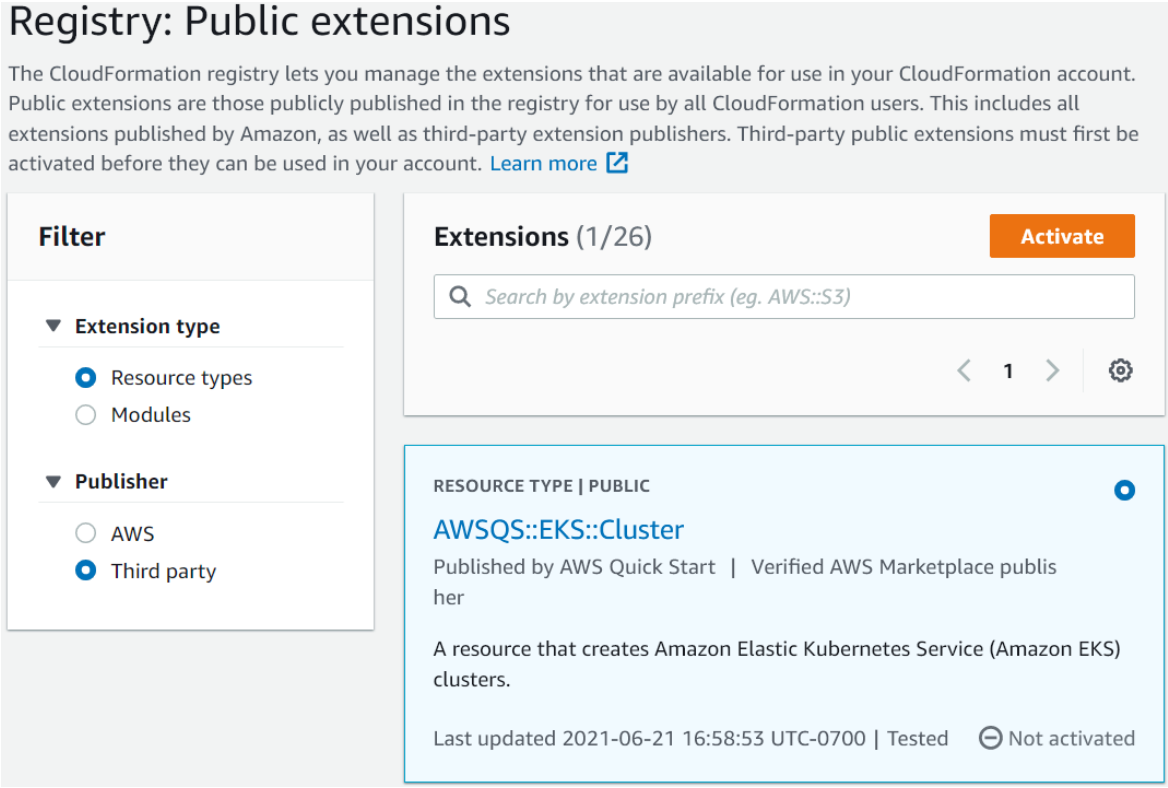
> **Note**
> When you use AWS CloudFormation with third-party resource types, you will incur charges based on the number of handler operations you run per month and handler operation duration. See CloudFormation pricing for complete details.

See Using public extensions in CloudFormation for complete documentation of this feature from the AWS CloudFormation side.

Activating a third-party resource in your account and region

Extensions published by AWS do not require activation; they are always available in every account and region. You can activate a third-party extension through the AWS Management Console, via the AWS Command Line Interface, or by deploying a special AWS CloudFormation resource.

To activate a third-party extension through the AWS Management Console, or to simply see what resources are available, follow these steps.



1. Log in to the AWS account in which you want to use the extension, then switch to the region where you want to use it.
2. Navigate to the CloudFormation console via the **Services** menu.
3. Click **Public extensions** on the navigation bar, then activate the **Third party** radio button under **Publisher**. A list of the available third-party public extensions appears. (You may also choose **AWS** to see a list of the public extensions published by AWS, though you don't need to activate them.)
4. Browse the list and find the extension you want to activate, or search for it, then activate the radio button in the upper right corner of the extension's card.
5. Click the **Activate** button at the top of the list to activate the selected extension. The extension's **Activate** page appears.
6. In the **Activate** page, you may override the extension's default name, specify an execution role and logging configuration, and choose whether to automatically update the extension when a new version is released. When you have set these options as you like, click **Activate extension** at the bottom of the page.

To activate a third-party extension using the AWS CLI, use the `activate-type` command, substituting the ARN of the custom type you want to use for the one given.

```
aws cloudformation activate-type --public-type-arn public_extension_ARN --auto-update-
activated
```

To activate an extension through CloudFormation or the CDK, deploy a resource of type `AWS::CloudFormation::TypeActivation`, specifying the following properties.

- `TypeName` - The name of the type, such as `AWSQS::EKS::Cluster`.

- `MajorVersion` - The major version number of the extension that you want; omit if you want the latest version.
- `AutoUpdate` - Whether to automatically update this extension when a new minor version is released by the publisher. (Major version updates require explicitly changing the `MajorVersion` property.)
- `ExecutionRoleArn` - The ARN of the IAM role under which this extension will run.
- `LoggingConfig` - The logging configuration for the extension.

The `TypeActivation` resource can be deployed by the CDK using the `CfnResource` construct, as shown below for the actual etxensions.

Adding a resource from the AWS CloudFormation Public Registry to your CDK app

Use the `CfnResource` construct to include a resource from the AWS CloudFormation Public Registry in your application. This construct is in the CDK's `core` module.

For example, suppose there is a public resource named `MY::S5::UltimateBucket` that you want to use in your AWS CDK application. This resource takes one property: the bucket name. The corresponding `CfnResource` instantiation looks like this.

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(new HashMap<String, String>() {{
     put("BucketName", "UltimateBucket");
 }});
```

C#

```
new CfnResource(this, "MyUltimateBucket", new CfnResourceProps
```

```
{
    Type = "MY::S5::UltimateBucket::MODULE",
    Properties = new Dictionary<string, object>
    {
        ["BucketName"] = "UltimateBucket"
    }
});
```

Get a value from the Systems Manager Parameter Store

The AWS CDK can retrieve the value of AWS Systems Manager Parameter Store attributes. During synthesis, the AWS CDK produces a token (p. 122) that is resolved by AWS CloudFormation during deployment.

The AWS CDK supports retrieving both plain and secure values. You may request a specific version of either kind of value. For plain values only, you may omit the version from your request to receive the latest version. You must always specify the version when requesting the value of a secure attribute.

Note
This topic shows how to read attributes from the AWS Systems Manager Parameter Store. You can also read secrets from the AWS Secrets Manager (see Get a value from AWS Secrets Manager (p. 239)).

Reading Systems Manager values at deployment time

To read values from the Systems Manager Parameter Store, use the valueForStringParameter and valueForSecureStringParameter methods, depending on whether the attribute you want is a plain string or a secure string value. These methods return tokens (p. 122), not the actual value. The value is resolved by AWS CloudFormation during deployment.

A limited number of AWS services currently support this feature.

TypeScript

```
import * as ssm from '@aws-cdk/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/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
```

Reading Systems Manager values at synthesis time

It is sometimes useful to "bake in" a parameter at synthesis time, so that the resulting AWS CloudFormation template always uses the same value, rather than resolving the value during deployment.

To read a value from the Systems Manager parameter store at synthesis time, use the valueFromLookup method (Python: `value_from_lookup`). This method returns the actual value of the parameter as a the section called “Context” (p. 159) value. If the value is not already cached in `cdk.json` or passed on the command line, it will be retrieved from the current AWS account. For this reason, the stack *must* be synthesized with explicit account and region information.

Only plain Systems Manager strings may be retrieved, not secure strings. It is not possible to request a specific version; the latest version is always returned.

Important
The retrieved value will end up in your synthenized AWS CloudFormation template, which may be a security risk depending on who has access to your your AWS CloudFormation templates and what kind of value it is. Generally, don't use this feature for passwords, keys, or other values you want to keep private.

TypeScript

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

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

JavaScript

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

Writing values to Systems Manager

You can use the AWS CLI, the AWS Management Console, or an AWS SDK to set Systems Manager parameter values. The following examples use the ssm put-parameter CLI command.

```
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"
```

When updating an SSM value that already exists, also include the `--overwrite` option.

```
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"
```

Get a value from AWS Secrets Manager

To use values from AWS Secrets Manager in your AWS CDK app, use the fromSecretAttributes method. It represents a value that is retrieved from Secrets Manager and used at AWS CloudFormation deployment time.

TypeScript

```
import * as sm from "@aws-cdk/aws-secretsmanager";

export class SecretsManagerStack extends core.Stack {
  constructor(scope: core.App, id: string, props?: core.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/aws-secretsmanager");

class SecretsManagerStack extends core.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(core.Stack):
    def __init__(self, scope: core.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 = ...,
        });
    }
```

Use the create-secret CLI command to create a secret from the command-line, such as when testing:

```
aws secretsmanager create-secret --name ImportedSecret --secret-string mygroovybucket
```

The command returns an ARN you can use for the example.

Create an app with multiple stacks

Most of the other code examples in the *AWS CDK Developer Guide* involve only a single stack. However, you can create apps containing any number of stacks. Each stack results in its own AWS CloudFormation template. Stacks are the *unit of deployment:* each stack in an app can be synthesized and deployed individually using the `cdk deploy` command.

This topic illustrates how to extend the `Stack` class to accept new properties or arguments, how to use these properties to affect what resources the stack contains and their configuration, and how to instantiate multiple stacks from this class. The example uses a Boolean property, named `encryptBucket` (Python: `encrypt_bucket`), to indicate whether an Amazon S3 bucket should be encrypted. If so, the stack enables encryption using a key managed by AWS Key Management Service (AWS KMS). The app creates two instances of this stack, one with encryption and one without.

Before you begin

First, install Node.js and the AWS CDK command line tools, if you haven't already. See Getting started with the AWS CDK (p. 8) for details.

Next, create an AWS CDK project by entering the following commands at the command line.

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
pip install -r requirements.txt
```

Java

```
mkdir multistack
cd multistack
cdk init --language=java
```

You can import the resulting Maven project into your Java IDE.

C#

```
mkdir multistack
cd multistack
cdk init --language=csharp
```

You can open the file `src/Pipeline.sln` in Visual Studio.

Finally, install the `core` and `s3` AWS Construct Library modules. We use these modules in our app.

TypeScript

```
npm install @aws-cdk/core @aws-cdk/aws-s3
```

JavaScript

```
npm install @aws-cdk/core @aws-cdk/aws-s3
```

Python

```
pip install aws_cdk.core aws_cdk.aws_s3
```

Java

Using the Maven integration in your IDE (for example, in Eclipse, right-click the project and choose **Maven** > **Add Dependency**), add the following packages in the group `software.amazon.awscdk`.

```
core
s3
```

C#

```
nuget install Amazon.CDK
nuget install Amazon.CDK.AWS.S3
```

Or **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio

Tip
If you don't see these packages in the **Browse** tab of the **Manage Packages for Solution** page, make sure the **Include prerelease** checkbox is ticked.
For a better experience, also add the `Amazon.Jsii.Analyzers` package to provide compile-time checks for missing required properties.

Add optional parameter

The `props` argument of the `Stack` constructor fulfills the interface `StackProps`. Because we want our stack to accept an additional property to tell us whether to encrypt the Amazon S3 bucket, we should create an interface or class that includes that property. This allows the compiler to make sure the property has a Boolean value and enables autocompletion for it in your IDE.

So open the indicated source file in your IDE or editor and add the new interface, class, or argument. The code should look like this after the changes. The lines we added are shown in boldface.

TypeScript

File: `lib/multistack-stack.ts`

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

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

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

    // The code that defines your stack goes here
  }
}
```

JavaScript

File: `lib/multistack-stack.js`

JavaScript doesn't have an interface feature; we don't need to add any code.

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

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

File: `multistack/multistack_stack.py`

Python does not have an interface feature, so we'll extend our stack to accept the new property by adding a keyword argument.

```
from aws_cdk import aws_s3 as s3

class MultistackStack(core.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: core.Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines your stack goes here
```

Java

File: `src/main/java/com/myorg/MultistackStack.java`

It's more complicated than we really want to get into to extend a props type in Java, so we'll simply write our stack's constructor to accept an optional Boolean parameter. Since `props` is an optional argument, we'll write an additional constructor that allows you to skip it. It will default to `false`.

```
package com.myorg;

import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.core.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#

File: `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, MultiStackProps props) :
 base(scope, id, props)
        {
            // The code that defines your stack goes here
        }
    }
}
```

The new property is optional. If `encryptBucket` (Python: `encrypt_bucket`) is not present, its value is `undefined`, or the local equivalent. The bucket will be unencrypted by default.

Define the stack class

Now let's define our stack class, using our new property. Make the code look like the following. The code you need to add or change is shown in boldface.

TypeScript

File: `lib/multistack-stack.ts`

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

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: cdk.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

File: `lib/multistack-stack.js`

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

File: `multistack/multistack_stack.py`

```
from aws_cdk import core
from aws_cdk import aws_s3 as s3

class MultistackStack(core.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: core.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=core.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                      removal_policy=core.RemovalPolicy.DESTROY)
```

Java

File: `src/main/java/com/myorg/MultistackStack.java`

```
package com.myorg;

import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.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#

File: `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
                });
            }
        }
    }
}
```

Create two stack instances

Now we'll add the code to instantiate two separate stacks. As before, the lines of code shown in boldface are the ones you need to add. Delete the existing `MultistackStack` definition.

TypeScript

File: `bin/multistack.ts`

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from '@aws-cdk/core';
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
});
```

JavaScript

File: `bin/multistack.js`

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

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

Python

File: `./app.py`

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

from aws_cdk import core

from multistack.multistack_stack import MultistackStack

app = core.App()
MultistackStack(app, "MyWestCdkStack",
                env=core.Environment(region="us-west-1"),
                encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
                env=core.Environment(region="us-east-1"),
                encrypt_bucket=True)
```

Java

File: `src/main/java/com/myorg/MultistackApp.java`

```
package com.myorg;

import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Environment;
import software.amazon.awscdk.core.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#

File: 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();
        }
    }
}
```

This code uses the new `encryptBucket` (Python: `encrypt_bucket`) property on the `MultistackStack` class to instantiate the following:

- One stack with an encrypted Amazon S3 bucket in the `us-east-1` AWS Region.
- One stack with an unencrypted Amazon S3 bucket in the `us-west-1` AWS Region.

Synthesize and deploy the stack

Now you can deploy stacks from the app. First, synthesize a AWS CloudFormation template for `MyEastCdkStack`—the stack in `us-east-1`. This is the stack with the encrypted S3 bucket.

```
$ cdk synth MyEastCdkStack
```

The output should look similar to the following AWS CloudFormation template (there might be slight differences).

```
Resources:
  MyGroovyBucketFD9882AC:
    Type: AWS::S3::Bucket
    Properties:
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
              SSEAlgorithm: aws:kms
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
    Metadata:
```

```
      aws:cdk:path: MyEastCdkStack/MyGroovyBucket/Resource
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Modules: aws-cdk=1.10.0,@aws-cdk/aws-events=1.10.0,@aws-cdk/aws-iam=1.10.0,@aws-cdk/
aws-kms=1.10.0,@aws-cdk/aws-s3=1.10.0,@aws-cdk/core=1.10.0,@aws-cdk/cx-api=1.10.0,@aws-cdk/
region-info=1.10.0,jsii-runtime=node.js/v10.16.2
```

To deploy this stack to your AWS account, issue one of the following commands. The first command uses your default AWS profile to obtain the credentials to deploy the stack. The second uses a profile you specify: for *PROFILE_NAME*, substitute the name of an AWS CLI profile that contains appropriate credentials for deploying to the `us-east-1` AWS Region.

```
cdk deploy MyEastCdkStack
```

```
cdk deploy MyEastCdkStack --profile=PROFILE_NAME
```

Clean up

To avoid charges for resources that you deployed, destroy the stack using the following command.

```
cdk destroy MyEastCdkStack
```

The destroy operation fails if there is anything stored in the stack's bucket. There shouldn't be if you've only followed the instructions in this topic. But if you did put something in the bucket, you must delete the bucket's contents, but not the bucket itself, using the AWS Management Console or the AWS CLI before destroying the stack.

Set a CloudWatch alarm

The aws-cloudwatch package supports setting CloudWatch alarms on CloudWatch metrics. So the first thing you need is a metric. You can use a predefined metric or you can create your own.

Using an existing metric

Many AWS Construct Library modules let you set an alarm on an existing metric by passing the metric's name to a convenience method on an instance of an object that has metrics. For example, given an Amazon SQS queue, you can get the metric **ApproximateNumberOfMessagesVisible** from the queue's metric() method.

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");
```

Creating your own metric

Create your own metric as follows, where the *namespace* value should be something like **AWS/SQS** for an Amazon SQS queue. You also need to specify your metric's name and dimension.

TypeScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensions: { MyDimension: 'MyDimensionValue' }
});
```

JavaScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensions: { MyDimension: 'MyDimensionValue' }
});
```

Python

```
metric = cloudwatch.Metric(
    namespace="MyNamespace",
    metric_name="MyMetric",
    dimensions=dict(MyDimension="MyDimensionValue")
)
```

Java

```
Metric metric = Metric.Builder.create()
        .namespace("MyNamespace")
        .metricName("MyMetric")
        .dimensions(new HashMap<String, Object>() {{
            put("MyDimension", "MyDimensionValue");
        }}).build();
```

C#

```
var metric = new Metric(this, "Metric", new MetricProps
{
    Namespace = "MyNamespace",
    MetricName = "MyMetric",
    Dimensions = new Dictionary<string, object>
    {
        { "MyDimension", "MyDimensionValue" }
    }
```

```
});
```

Creating the alarm

Once you have a metric, either an existing one or one you defined, you can create an alarm. In this example, the alarm is raised when there are more than 100 of your metric in two of the last three seconds. You can use comparisons such as less-than in your alarms via the `comparisonOperator` property; greater-than-or-equal-to is the AWS CDK default, so we don't need to psecify it.

Assuming the metric is the **ApproximateNumberOfMessagesVisible** metric from an Amazon SQS queue, it would raise when 100 messages are visible in the queue in two of the last three seconds.

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
});
```

An alternative way to create an alarm is using the metric's createAlarm() method, which takes essentially the same properties as the `Alarm` constructor; you just don't need to pass in the metric, since it's already known.

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
});
```

Get a value from a context variable

You can specify a context variable either as part of an AWS CDK CLI command, or in `cdk.json`.

To create a command line context variable, use the **--context** (**-c**) option, as shown in the following example.

```
cdk synth -c bucket_name=mygroovybucket
```

To specify the same context variable and value in the `cdk.json` file, use the following code.

```
{
  "context": {
    "bucket_name": "myotherbucket"
  }
}
```

To get the value of a context variable in your app, use the `TryGetContext` method in the context of a construct (that is, when `this`, or `self` in Python, is an instance of some construct). The example gets the context value **bucket_name**. If the requested value is not defined, `TryGetContext` returns `undefined` (`None` in Python; `null` in Java and C#) rather than raising an exception.

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");
```

Outside the context of a construct, you can access the context variable from the app object, like this.

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");
```

For more details on working with context variables, see the section called “Context” (p. 159).

Continuous integration and delivery (CI/CD) using CDK Pipelines

CDK Pipelines is a construct library module for painless continuous delivery of AWS CDK applications. Whenever you check your AWS CDK app's source code in to AWS CodeCommit, GitHub, or CodeStar, CDK Pipelines can automatically build, test, and deploy your new version.

CDK Pipelines are self-updating: if you add application stages or stacks, the pipeline automatically reconfigures itself to deploy those new stages and/or stacks.

> **Note**
> CDK Pipelines supports two APIs: the original API that was made available in the Developer Preview, and a modern one that incorporates feedback from CDK customers received during the preview phase. The examples in this topic use the modern API. For details on the differences between the two supported APIs, see CDK Pipelines original API.

Bootstrap your AWS environments

Before you can use CDK Pipelines, you must bootstrap the AWS environment(s) to which you will deploy your stacks. An environment (p. 92) is an account/region pair to which you want to deploy a CDK stack. A CDK Pipeline involves at least two environments: the environment where the pipeline is provisioned, and the environment where you want to deploy the application's stacks (or its stages, which are groups of related stacks). These environments can be the same, though best practices recommend you isolate stages from each other in different AWS accounts or regions.

> **Note**
> See the section called “Bootstrapping” (p. 174) for more information on the kinds of resources created by bootstrapping and how to customize the bootstrap stack.

You may have already bootstrapped one or more environments so you can deploy assets and Lambda functions using the AWS CDK. Continuous deployment with CDK Pipelines requires that the CDK Toolkit stack include additional resources, so the bootstrap stack has been extended to include an additional Amazon S3 bucket, an Amazon ECR repository, and IAM roles to give the various parts of a pipeline the permissions they need. This new style of CDK Toolkit stack will eventually become the default, but at this writing, you must opt in. The AWS CDK Toolkit will upgrade your existing bootstrap stack or create a new one, as necessary.

To bootstrap an environment that can provision an AWS CDK pipeline, set the environment variable `CDK_NEW_BOOTSTRAP` before invoking `cdk bootstrap`, as shown below. Invoking the AWS CDK Toolkit via the `npx` command installs it if necessary, and will use the version of the Toolkit installed in the current project if one exists.

--cloudformation-execution-policies specifies the ARN of a policy under which future CDK Pipelines deployments will execute. The default `AdministratorAccess` policy ensures that your pipeline can deploy every type of AWS resource. If you use this policy, make sure you trust all the code and dependencies that make up your AWS CDK app.

Most organizations mandate stricter controls on what kinds of resources can be deployed by automation. Check with the appropriate department within your organization to determine the policy your pipeline should use.

You may omit the **--profile** option if your default AWS profile contains the necessary credentials or to instead use the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` to provide your AWS account credentials.

macOS/Linux

```
export CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess \
    aws://ACCOUNT-ID/REGION
```

Windows

```
set CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess ^
    aws://ACCOUNT-ID/REGION
```

To bootstrap additional environments into which AWS CDK applications will be deployed by the pipeline, use the commands below instead. The **--trust** option indicates which other account should have permissions to deploy AWS CDK applications into this environment; specify the pipeline's AWS account ID.

Again, you may omit the **--profile** option if your default AWS profile contains the necessary credentials or if you are using the `AWS_*` environment variables to provide your AWS account credentials.

macOS/Linux

```
export CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess \
    --trust PIPELINE-ACCOUNT-ID \
    aws://ACCOUNT-ID/REGION
```

Windows

```
set CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap aws://ACCOUNT-NUMBER/REGION --profile ADMIN-PROFILE ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess ^
    --trust PIPELINE-ACCOUNT-ID ^
    aws://ACCOUNT-ID/REGION
```

Tip
Use administrative credentials only to bootstrap and to provision the initial pipeline. Afterward, use the pipeline itself, not your local machine, to deploy changes.

If you are upgrading a legacy-bootstrapped environment, the old Amazon S3 bucket is orphaned when the new bucket is created. Delete it manually using the Amazon S3 console.

Initialize project

Create a new, empty GitHub project and clone it to your workstation in the `my-pipeline` directory. (Our code examples in this topic use GitHub; you can also use CodeStar or AWS CodeCommit.)

```
git clone GITHUB-CLONE-URL my-pipeline
cd my-pipeline
```

> **Note**
> You may use a name other than `my-pipeline` for your app's main directory, but since the AWS CDK Toolkit bases some file and class names on the name of the main directory, you'll need to tweak these later in this topic.

After cloning, initialize the project as usual.

TypeScript

```
cdk init app --language typescript
```

JavaScript

```
cdk init app --language javascript
```

Python

```
cdk init app --language python
```

After the app has been created, also enter the following two commands to activate the app's Python virtual environment and install the AWS CDK core dependencies.

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
cdk init app --language java
```

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

C#

```
cdk init app --language csharp
```

If you are using Visual Studio, open the solution file in the `src` directory.

Install the CDK Pipelines module along with any others you'll be using.

> **Tip**
> Be sure to commit your `cdk.json` and `cdk.context.json` files in source control. The context information (such as feature flags and cached values retrieved from your AWS account) are part of your project's state. The values may be different in another environment, which can cause instability (unexpected changes) in your results.

TypeScript

```
npm install @aws-cdk/pipelines @aws-cdk/aws-lambda
```

JavaScript

```
npm install @aws-cdk/pipelines @aws-cdk/aws-lambda
```

Python

```
python -m pip install aws_cdk.pipelines aws_cdk.aws_lambda
```

Add the project's dependencies to `requirements.txt` so they can be installed in the CI/CD environment. It is convenient to use `pip freeze` for this.

macOS/Linux

```
python -m pip freeze | grep -v '^[-#]' > requirements.txt
```

Windows

```
python -m pip freeze | findstr /R /B /V "[-#]" > requirements.txt
```

Java

Edit your project's `pom.xml` and add a `<dependency>` element for the `pipeline` module and the others you'll need. Follow the template below for each module, placing each inside the existing `<dependencies>` container.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>cdk-pipelines</artifactId>
    <version>${cdk.version}</version>
</dependency>
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>lambda</artifactId>
    <version>${cdk.version}</version>
</dependency>
```

After updating `pom.xml`, issue `mvn package` to install the new modules.

C#

In Visual Studio, choose **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio and add the following packages.

```
Amazon.CDK.Pipelines
Amazon.CDK.AWS.Lambda
```

Finally, add the `@aws-cdk/core:newStyleStackSynthesis` feature flag (p. 164) to the new project's `cdk.json` file. The file will already contain some context values; add this new one inside the `context` object if it's not already there.

```
{
  ...
  "context": {
    ...
    "@aws-cdk/core:newStyleStackSynthesis": true
  }
}
```

In a future release of the AWS CDK, "new style" stack synthesis will become the default, but for now we need to opt in using the feature flag.

Define a pipeline

Your CDK Pipelines application will include at least two stacks: one that represents the pipeline itself, and one or more stacks that represent the application deployed through it. Stacks can also be grouped into *stages*, which you can use to deploy copies of infrastructure stacks to different environments. For now, we'll consider the pipeline, and later delve into the application it will deploy.

The construct `CodePipeline` is the construct that represents a CDK Pipeline that uses AWS CodePipeline as its deployment engine. When you instantiate `CodePipeline` in a stack, you define the source location for the pipeline (e.g. a GitHub repository) and the commands to build the app. For example, the following defines a pipeline whose source is stored in a GitHub repository, and includes a build step for a TypeScript CDK application. Fill in the information about your GitHub repo where indicated.

> **Note**
> By default, the pipeline authenticates to GitHub using a personal access token stored in Secrets Manager under the name `github-token`.

You'll also need to update the instantiation of the pipeline stack to specify the AWS account and region.

TypeScript

In `lib/my-pipeline-stack.ts` (may vary if your project folder isn't named `my-pipeline`):

```
import * as cdk from '@aws-cdk/core';
import { CodePipeline, CodePipelineSource, ShellStep } from '@aws-cdk/pipelines';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: cdk.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']
      })
    });
  }
}
```

In `bin/my-pipeline.ts` (may vary if your project folder isn't named `my-pipeline`):

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from '@aws-cdk/core';
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

In `lib/my-pipeline-stack.js` (may vary if your project folder isn't named `my-pipeline`):

```
const cdk = require('@aws-cdk/core');
const { CodePipeline, CodePipelineSource, ShellStep } = require('@aws-cdk/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 }
```

In `bin/my-pipeline.js` (may vary if your project folder isn't named `my-pipeline`):

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

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

In `my-pipeline/my-pipeline-stack.py` (may vary if your project folder isn't named `my-pipeline`):

```
from aws_cdk import core as cdk
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: cdk.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"]
                        )
                    )
```

In `app.py`:

```
#!/usr/bin/env python3
from aws_cdk import core 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

In `src/main/java/com/myorg/MyPipelineStack.java` (may vary if your project folder isn't named `my-pipeline`):

```
package com.myorg;

import java.util.Arrays;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.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();
    }
}
```

In `src/main/java/com/myorg/MyPipelineApp.java` (may vary if your project folder isn't named `my-pipeline`):

```
package com.myorg;
```

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

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

        new MyPipelineStack(app, "PipelineStack", StackProps.builder()
            .env(new Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build());

        app.synth();
    }
}
```

C#

In `src/MyPipeline/MyPipelineStack.cs` (may vary if your project folder isn't named `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" }
                })
            });
        }
    }
}
```

In `src/MyPipeline/Program.cs` (may vary if your project folder isn't named `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();
        }
    }
}
```

You must deploy a pipeline manually once. After that, the pipeline will keep itself up to date from the source code repository, so make sure the code in the repo is the code you want deployed. Check in your changes and push to GitHub, then deploy:

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

Tip
Now that you've done the initial deployment, your local AWS account no longer needs administrative access, because all changes to your app will be deployed via the pipeline. All you need to be able to do is push to GitHub.

Application stages

To define a multi-stack AWS application that can be added to the pipeline all at once, define a subclass of `Stage` (not to be confused with `CdkStage` in the CDK Pipelines module).

The stage contains the stacks that make up your application. If there are dependencies between the stacks, the stacks are automatically added to the pipeline in the right order. Stacks that don't depend on each other are deployed in parallel. You can add a dependency relationship between stacks by calling `stack1.addDependency(stack2)`.

Stages accept a default `env` argument, which becomes the default environment for the stacks inside it. (Stacks can still have their own environment specified.).

An application is added to the pipeline by calling `addStage()` with instances of `Stage`. A stage can be instantiated and added to the pipeline multiple times to define different stages of your DTAP or multi-region application pipeline:

We will create a stack containing a simple Lambda function and place that stack in a stage. Then we will add the stage to the pipeline so it can be deployed.

TypeScript

Create the new file `lib/my-pipeline-lambda-stack.ts` to hold our application stack containing a Lambda function.

```
import * as cdk from '@aws-cdk/core';
import { Function, InlineCode, Runtime } from '@aws-cdk/aws-lambda';

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

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_12_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
```

```
}
```

Create the new file `lib/my-pipeline-app-stage.ts` to hold our stage.

```
import * as cdk from '@aws-cdk/core';
import { MyLambdaStack } from './my-pipeline-lambda-stack';

export class MyPipelineAppStage extends cdk.Stage {

    constructor(scope: cdk.Construct, id: string, props?: cdk.StageProps) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}
```

Edit `lib/my-pipeline-stack.ts` to add the stage to our pipeline.

```
import * as cdk from '@aws-cdk/core';
import { CodePipeline, CodePipelineSource, ShellStep } from '@aws-cdk/pipelines';
import { MyPipelineAppStage } from './my-pipeline-app-stage';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: cdk.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

Create the new file `lib/my-pipeline-lambda-stack.js` to hold our application stack containing a Lambda function.

```
const cdk = require('@aws-cdk/core');
const { Function, InlineCode, Runtime } = require('@aws-cdk/aws-lambda');

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

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_12_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}

module.exports = { MyLambdaStack }
```

Create the new file `lib/my-pipeline-app-stage.js` to hold our stage.

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

Edit `lib/my-pipeline-stack.ts` to add the stage to our pipeline.

```
const cdk = require('@aws-cdk/core');
const { CodePipeline, CodePipelineSource, ShellStep } = require('@aws-cdk/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

Create the new file `my_pipeline/my_pipeline_lambda_stack.py` to hold our application stack containing a Lambda function.

```
from aws_cdk import core as cdk
from aws_cdk.aws_lambda import Function, InlieCode, Runtime

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

        Function(self, "LambdaFunction",
            runtime=Runtime.NODEJS_12_X,
            handler="index.handler",
            code=InlineCode("exports.handler = _ => 'Hello, CDK';")
        )
```

Create the new file `my_pipeline/my_pipeline_app_stage.py` to hold our stage.

```
from aws_cdk import core as cdk
from my_pipeline.my_pipeline_lambda_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage):
    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        lambdaStack = MyLambdaStack(self, "LambdaStack")
```

Edit `my_pipeline/my_pipeline_stack.py` to add the stage to our pipeline.

```
from aws_cdk import core as cdk
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: cdk.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

Create the new file `src/main/java/com.myorg/MyPipelineLambdaStack.java` to hold our application stack containing a Lambda function.

```
package com.myorg;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.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_12_X)
          .handler("index.handler")
          .code(new InlineCode("exports.handler = _ => 'Hello, CDK';"))
          .build();

    }
```

```
}
```

Create the new file `src/main/java/com.myorg/MyPipelineAppStage.java` to hold our stage.

```
package com.myorg;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.Stage;
import software.amazon.awscdk.core.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");
    }

}
```

Edit `src/main/java/com.myorg/MyPipelineStack.java` to add the stage to our pipeline.

```
package com.myorg;

import java.util.Arrays;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Environment;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.core.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#

Create the new file `src/MyPipeline/MyPipelineLambdaStack.cs` to hold our application stack containing a Lambda function.

```
using Amazon.CDK;
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_12_X,
                Handler = "index.handler",
                Code = new InlineCode("exports.handler = _ => 'Hello, CDK';")
            });
        }
    }
}
```

Create the new file `src/MyPipeline/MyPipelineAppStage.cs` to hold our stage.

```
using Amazon.CDK;

namespace MyPipeline
{
    class MyPipelineAppStage : Stage
    {
        public MyPipelineAppStage(Construct scope, string id, StageProps props=null) :
 base(scope, id, props)
        {
            Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
        }
    }
}
```

Edit `src/MyPipeline/MyPipelineStack.cs` to add the stage to our 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" }
                })
            });

            pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
            {
                Env = new Environment
                {
                    Account = "111111111111", Region = "eu-west-1"
                }
            }));
        }
    }
}
```

Every application stage added by `addStage()` results in the addition of a corresponding pipeline stage, represented by a StageDeployment instance returned by the `addStage()` call. You can add pre-deployment or post-deployment actions to the stage by calling its `addPre()` or `addPost()` method.

TypeScript

```
// import { ManualApprovalStep } from '@aws-cdk/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/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"));
```

You can add stages to a Wave to deploy them in parallel, for example when deploying a stage to multiple accounts or regions.

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"
    }
}));
```

Testing deployments

You can add steps to a CDK Pipeline to validate the deployments you are performing. Using the CDK Pipeline library's `ShellStep`, you can try to access a just-deployed Amazon API Gateway backed by a Lambda function, for example, or issue an AWS CLI command to check some setting of a deployed resource.

In its simplest form, adding validation actions looks like this:

TypeScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['curl -Ssf https://my.webservice.com/'],
}));
```

JavaScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['curl -Ssf https://my.webservice.com/'],
}));
```

Python

```
# stage was returned by pipeline.add_stage

stage.add_post(ShellStep("validate",
  commands=['curl -Ssf https://my.webservice.com/']
))
```

Java

```
// stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create("validate")
        .commands(Arrays.asList("curl -Ssf https://my.webservice.com/"))
        .build());
```

C#

```
// stage was returned by pipeline.addStage

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Commands = new string[] { "curl -Ssf https://my.webservice.com/" }
}));
```

Because many AWS CloudFormation deployments result in the generation of resources with unpredictable names, CDK Pipelines provide a way to read AWS CloudFormation outputs after a deployment. This makes it possible to pass (for example) the generated URL of a load balancer to a test action.

To use outputs, expose the `CfnOutput` object you're interested in and pass it in a step's `envFromCfnOutputs` property to make it available as an environment variable within that step.

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(new HashMap<String, CfnOutput>() {{
         put("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" }
}));
```

You can write simple validation tests right in the `ShellStep`, but this approach becomes unwieldy when the test is more than a few lines. For more complex tests, you can bring additional files (such as complete shell scripts, or programs in other languages) into the `ShellStep` via the `inputs` property. The inputs can be any step that has an output, including a source (such as a GitHub repo) or another `ShellStep`.

Bringing in files from the source repository is appropriate if the files are directly usable in the test (for example, if they are themselves executable). In this example, we declare our GitHub repo as `source` (rather than instantiating it inline as part of the `CodePipeline`), then pass this fileset to both the pipeline and the validation test.

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=["curl -Ssf https://my.webservice.com/"],
))
```

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" }
}));
```

Getting the additional files from the synth step is appropriate if your tests need to be compiled, which is done as part of synthesis.

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" }
}));
```

Security notes

Any form of continuous delivery has inherent security risks. Under the AWS Shared Responsibility Model, you are responsible for the security of your information in the AWS cloud. The CDK Pipelines library gives you a head start by incorporating secure defaults and modeling best practices, but by its very nature a library that needs a high level of access to fulfill its intended purpose cannot assure complete security. There are many attack vectors outside of AWS and your organization.

In particular, keep in mind the following.

- Be mindful of the software you depend on. Vet all third-party software you run in your pipeline, as it has the ability to change the infrastructure that gets deployed.
- Use dependency locking to prevent accidental upgrades. CDK Pipelines respects `package-lock.json` and `yarn.lock` to ensure your dependencies are the ones you expect.
- Credentials for production environments should be short-lived. After bootstrapping and initial provisioning, there is no need for developers to have account credentials at all; changes can be deployed through the pipeline. Eliminate the possibility of credentials leaking by not needing them in the first place!

Troubleshooting

The following issues are commonly encountered while getting started with CDK Pipelines.

Pipeline: Internal Failure

```
CREATE_FAILED  | AWS::CodePipeline::Pipeline | Pipeline/Pipeline
```

```
Internal Failure
```

Check your GitHub access token. It might be missing, or might not have the permissions to access the repository.

Key: Policy contains a statement with one or more invalid principals

```
CREATE_FAILED | AWS::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.
```

One of the target environments has not been bootstrapped with the new bootstrap stack. Make sure all your target environments are bootstrapped.

Stack is in ROLLBACK_COMPLETE state and can not be updated.

```
Stack STACK_NAME is in ROLLBACK_COMPLETE state and can not be updated. (Service:
AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request
ID: ...)
```

The stack failed its previous deployment and is in a non-retryable state. Delete the stack from the AWS CloudFormation console and retry the deployment.

AWS CDK tools

This section contains information about the AWS CDK tools listed below.

Topics

AWS CDK Toolkit (`cdk` command)

The AWS CDK Toolkit, the CLI command `cdk`, is the primary tool for interacting with your AWS CDK app. It executes your app, interrogates the application model you defined, and produces and deploys the AWS CloudFormation templates generated by the AWS CDK. It also provides other features useful for creating and working with AWS CDK projects. This topic contains information about common use cases of the CDK Toolkit.

The AWS CDK Toolkit is installed with the Node Package Manager. In most cases, we recommend installing it globally.

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

Tip
If you regularly work with multiple versions of the AWS CDK, you may want to install a matching version of the AWS CDK Toolkit in individual CDK projects. To do this, omit `-g` from the `npm install` command. Then use `npx cdk` to invoke it; this will run the local version if one exists, falling back to a global version if not.

Toolkit commands

All CDK Toolkit commands start with `cdk`, which is followed by a subcommand (`list`, `synthesize`, `deploy`, etc.). Some subcommands have a shorter version (`ls`, `synth`, etc.) that is equivalent. Options and arguments follow the subcommand in any order. The available commands are summarized here.

Command	Function
`cdk list` (`ls`)	Lists the stacks in the app
`cdk synthesize` (`synth`)	Synthesizes and prints the CloudFormation template for the specified stack(s)
`cdk bootstrap`	Deploys the CDK Toolkit staging stack; see the section called "Bootstrapping" (p. 174)
`cdk deploy`	Deploys the specified stack(s)
`cdk destroy`	Destroys the specified stack(s)
`cdk diff`	Compares the specified stack with the deployed stack or a local CloudFormation template
`cdk metadata`	Displays metadata about the specified stack

Command	Function
`cdk init`	Creates a new CDK project in the current directory from a specified template
`cdk context`	Manages cached context values
`cdk docs` (`doc`)	Opens the CDK API reference in your browser
`cdk doctor`	Checks your CDK project for potential problems

For the options available for each command, see the section called "Toolkit reference" (p. 290) or the section called "Built-in help" (p. 280).

Specifying options and their values

Command line options begin with two hyphens (`--`). Some frequently-used options have single-letter synonyms that begin with a single hyphen (for example, `--app` has a synonym `-a`). The order of options in an AWS CDK Toolkit command is not important.

All options accept a value, which must follow the option name. The value may be separated from the name by whitespace or by an equals sign `=`. The following two options are equivalent

```
--toolkit-stack-name MyBootstrapStack
--toolkit-stack-name=MyBootstrapStack
```

Some options are flags (Booleans). You may specify `true` or `false` as their value. If you do not provide a value, the value is taken to be `true`. You may also prefix the option name with `no-` to imply `false`.

```
# sets staging flag to true
--staging
--staging=true
--staging true

# sets staging flag to false
--no-staging
--staging=false
--staging false
```

A few flags, namely `--context`, `--parameters`, `--plugin`, `--tags`, and `--trust`, may be specified more than once to specify multiple values. These are noted as having `[array]` type in the CDK Toolkit help.

Built-in help

The AWS CDK Toolkit has integrated help. You can see general help about the utility and a list of the provided subcommands by issuing:

```
cdk --help
```

To see help for a particular subcommand, for example `deploy`, specify it before the `--help` flag.

```
cdk deploy --help
```

Issue `cdk version` to display the version of the AWS CDK Toolkit. Provide this information when requesting support.

Version reporting

To gain insight into how the AWS CDK is used, the constructs used by AWS CDK applications are collected and reported by using a resource identified as `AWS::CDK::Metadata`. This resource is added to AWS CloudFormation templates, and can easily be reviewed. This information can also be used by AWS to identify stacks using a construct with known security or reliability issues, and to contact their users with important information.

> **Note**
> Prior to version 1.93.0, the AWS CDK reported the names and versions of the modules loaded during synthesis, rather than the constructs used in the stack.

By default, the AWS CDK reports the use of constructs in the following NPM modules that are used in the stack:

- AWS CDK core module
- AWS Construct Library modules
- AWS Solutions Constructs module
- AWS Render Farm Deployment Kit module

The `AWS::CDK::Metadata` resource looks something like the following.

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Analytics: "v2:deflate64:H4sIAND9SGAAAzXKSw5AMBAA0L1b2PdzBYnEAdio3RglglY60zQi7u6TWL/
XKmNUlxeQSOKwaPTBqrNhwEWU3hGHiCzK0dWWfAxoL/Fd8mvk+QkS/0X6BdjnCdgmOOQKWz
+AqqLDt2Y3YMnLYWwAAAA="
```

The `Analytics` property is a gzipped, base64-encoded, prefix-encoded list of the constructs present in the stack.

To opt out of version reporting, use one of the following methods:

- Use the **cdk** command with the **--no-version-reporting** argument to opt out for a single command.

  ```
  cdk --no-version-reporting synth
  ```

  Remember, the AWS CDK Toolkit synthesizes fresh templates before deploying, so you should also add `--no-version-reporting` to `cdk deploy` commands.
- Set **versionReporting** to **false** in `./cdk.json` or `~/.cdk.json`. This opts out unless you opt in by specifying `--version-reporting` on an individual command.

  ```
  {
    "app": "...",
    "versionReporting": false
  }
  ```

Specifying credentials and region

The CDK Toolkit needs to know your AWS account credentials and the AWS region into which you are deploying, not only for deployment operations but also to retrieve context values during synthesis. Together, your account and region make up the *environment*.

Important
We strongly recommend against using your main AWS account for day-to-day tasks. Instead, create a user in IAM and use its credentials with the CDK.

Credentials and region may be specified using environment variables or in configuration files. These are the same variables and files used by other AWS tools such as the AWS CLI and the various AWS SDKs. The CDK Toolkit looks for this information in the following order.

- The `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` environment variables. Always specify all three variables, not just one or two.
- A specific profile defined in the standard AWS `config` and `credentials` files, and specified using the `--profile` option on `cdk` commands.
- The `[default]` section of the standard AWS `config` and `credentials` files.

Note
The standard AWS `config` and `credentials` files are located at `~/.aws/config` and `~/.aws/credentials` (macOS/Linux) or `%USERPROFILE%\.aws\config` and `%USERPROFILE%\.aws\credentials` (Windows).

The environment specified in your AWS CDK app using the stack's `env` property is used during synthesis to generate an environment-specific AWS CloudFormation template and during deployment to override the account or region specified by one of the above methods. For more information, see the section called “Environments” (p. 92).

If you have the AWS CLI installed, the easiest way to configure your account credentials and a default region is to issue the following command:

```
aws configure
```

Provide your AWS access key ID, secret access key, and default region when prompted. These values are written to the `[default]` section of the `config` and `credentials` files.

If you don't have the AWS CLI installed, you can manually create or edit the `config` and `credentials` files to contain default credentials and a default region, in the following format.

- In `~/.aws/config` or `%USERPROFILE%\.aws\config`

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

- In `~/.aws/credentials` or `%USERPROFILE%\.aws\credentials`

```
[default]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
```

Besides specifying AWS credentials and a region in the `[default]` section, you can also add one or more `[profile` *`NAME`*`]` sections, where *`NAME`* is the name of the profile.

- In `~/.aws/config` or `%USERPROFILE%\.aws\config`

```
[profile test]
region=us-east-1

[profile prod]
```

```
region=us-west-1
```

- In `~/.aws/credentials` or `%USERPROFILE%\.aws\credentials`

```
[profile test]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY

[profile test]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
```

Always add named profiles to both the `config` and `credentials` files. The AWS CDK Toolkit does not fall back to using the region in the `[default]` section when the specified named profile is not found in the `config` file, as some other AWS tools do.

Important
Do not name a profile `default`: that is, do not use a `[profile default]` section in either `config` or `credentials`.

Note
Although the AWS CDK uses credentials from the same sources files as other AWS tools and SDKs, including the AWS Command Line Interface, it may behave slightly differently from these tools. See Setting credentials for complete details on setting up credentials for the AWS SDK for JavaScript, which the AWS CDK uses under the hood.

You may optionally use the `--role-arn` (or `-r`) option to specify the ARN of an IAM role that should be used for deployment. This role must be assumable by the AWS account being used.

Specifying the app command

Many features of the CDK Toolkit require one or more AWS CloudFormation templates be synthesized, which in turn requires running your application. Since the AWS CDK supports programs written in a variety of languages, it uses a configuration option to specify the exact command necessary to run your app. This option can be specified in two ways.

First, and most commonly, it can be specified using the `app` key inside the file `cdk.json`, which is in the main directory of your AWS CDK project. The CDK Toolkit provides an appropriate command when creating a new project with `cdk init`. Here is the `cdk.json` from a fresh TypeScript project, for instance.

```
{
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

The CDK Toolkit looks for `cdk.json` in the current working directory when attempting to run your app, so you might keep a shell open in your project's main directory for issuing CDK Toolkit commands.

The CDK Toolkit also looks for the app key in `~/.cdk.json` (that is, in your home directory) if it can't find it in `./cdk.json`. Adding the app command here can be useful if you usually work with CDK code in the same language, as it does not require you to be in the app's main directory when you run a `cdk` command.

If you are in some other directory, or if you want to run your app via a command other than the one in `cdk.json`, you can use the `--app` (or `-a`) option to specify it.

```
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

Specifying stacks

Many CDK Toolkit commands (for example, `cdk deploy`) work on stacks defined in your app. If your app contains only one stack, the CDK Toolkit assumes you mean that one if you don't specify a stack explicitly.

Otherwise, you must specify the stack or stacks you want to work with. You can do this by specifying the desired stacks by ID individually on the command line. Recall that the ID is the value specified by the second argument when you instantiate the stack.

```
cdk synth PipelineStack LambdaStack
```

You may also use wildcards to specify IDs that match a pattern.

- `?` matches any single character
- `*` matches any number of characters (`*` alone matches all stacks)
- `**` matches everything in a hierarchy

You may also use the **--all** option to specify all stacks.

If your app uses CDK Pipelines (p. 255), the CDK Toolkit understands your stacks and stages as a hierarchy, and the **--all** option and the `*` wildcard only match top-level stacks. To match all the stacks, use `**`. Also use `**` to indicate all the stacks under a particular hierarchy.

When using wildcards, enclose the pattern in quotes, or escape the wildcards with `\`. If you don't, your shell may try to expand the pattern to the names of files in the current directory. At best, this won't do what you expect; at worst, you could deploy stacks you didn't intend to. This isn't strictly necessary on Windows because `cmd.exe` does not expand wildcards, but is good practice nonetheless.

```
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
The order in which you specify the stacks is not necessarily the order in which they will be processed. The AWS CDK Toolkit takes into account dependencies between stacks when deciding the order in which to process them. For example, if one stack uses a value produced by another (such as the ARN of a resource defined in the second stack), the second stack is synthesized before the first one because of this dependency. You can add dependencies between stacks manually using the stack's `addDependency()` method.

Bootstrapping your AWS environment

Deploying stacks that contain assets (p. 138), synthesize to large templates, or use CDK Pipelines (p. 255) require special dedicated AWS CDK resources to be provisioned. The `cdk bootstrap` command creates the necessary resources for you. You only need to bootstrap if you are deploying a stack that requires these dedicated resources. See the section called “Bootstrapping” (p. 174) for details.

```
cdk bootstrap
```

If issued with no arguments, as shown here, the `cdk bootstrap` command synthesizes the current app and bootstraps the environments its stacks will be deployed to. If the app contains environment-agnostic

stacks, which do not explicitly specify an environment so they can be deployed anywhere, the default account and region are bootstrapped, or the environment specified using `--profile`.

Outside of an app, you must explicitly specify the environment to be bootstrapped. You may also do so to bootstrap an environment that's not specified in your app or local AWS profile. Credentials must be configured (e.g. in `~/.aws/credentials`) for the specified account and region. You may specify a profile that contains the required credentials.

```
cdk bootstrap ACCOUNT-NUMBER/REGION # e.g.
cdk bootstrap 1111111111/us-east-1
cdk bootstrap --profile test 1111111111/us-east-1
```

Important
Each environment (account/region combination) to which you deploy such a stack must be bootstrapped separately.

You may incur AWS charges for what the AWS CDK stores in the bootstrapped resources. Additionally, if you use `-bootstrap-customer-key`, a Customer Master Key (CMK) will be created, which also incurs charges per environment.

Note
Older versions of the modern template created a Customer Master Key by default. To avoid charges, re-bootstrap using `--no-bootstrap-customer-key`.

The CDK Toolkit supports two bootstrap templates: the modern template and the legacy template. The legacy template is the default, but the modern template is required by CDK Pipelines. For more information, see the section called “Bootstrapping” (p. 174).

Important
The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list, which by default will extend permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack (p. 178) with policies and trusted accounts you are comfortable with.

Creating a new app

To create a new app, create a directory for it, then, inside the directory, issue `cdk init`.

```
mkdir my-cdk-app
cd my-cdk-app
cdk init TEMPLATE --language LANGUAGE
```

The supported languages (*LANGUAGE*) are:

Code	Language
`typescript`	TypeScript
`javascript`	JavaScript
`python`	Python
`java`	Java
`csharp`	C#

TEMPLATE is an optional template. If the desired template is *app*, the default, you may omit it. The available templates are:

Template	Description
`app` (default)	Creates an empty AWS CDK app.
`sample-app`	Creates an AWS CDK app with a stack containing an Amazon SQS queue and an Amazon SNS topic.

The templates use the name of the project folder to generate names for files and classes inside your new app.

Listing stacks

To see a list of the IDs of the stacks in your AWS CDK application, enter one of the following equivalent commands:

```
cdk list
cdk ls
```

If your application contains CDK Pipelines (p. 255) stacks, the CDK Toolkit displays stack names as paths according to their location in the pipeline hierarchy (e.g., `PipelineStack`, `PipelineStack/Prod`, `PipelineStack/Prod/MyService`, etc).

If your app contains many stacks, you can specify full or partial stack IDs of the stacks to be listed; see the section called "Specifying stacks" (p. 284).

Add the `--long` flag to see more information about the stacks, including the stack names and their environments (AWS account and region).

Synthesizing stacks

The `cdk synthesize` command (almost always abbreviated `synth`) synthesizes a stack defined in your app into a CloudFormation template.

```
cdk synth         # if app contains only one stack
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*"     # all stacks in app
```

Note
The CDK Toolkit actually runs your app and synthesizes fresh templates before most operations (e.g. when deploying or comparing stacks). These templates are stored by default in the `cdk.out` directory. The `cdk synth` command simply prints the generated templates for the specified stack(s).

See `cdk synth --help` for all available options. A few of the most-frequently-used options are covered below.

Specifying context values

Use the `--context` or `-c` option to pass runtime context (p. 159) values to your CDK app.

```
# 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
```

When deploying multiple stacks, the specified context values are normally passed to all of them. If you wish, you may specify different values for each stack by prefixing the stack name to the context value.

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

Specifying display format

By default, the synthesized template is displayed in YAML format. Add the `--json` flag to display it in JSON format instead.

```
cdk synth --json MyStack
```

Specifying output directory

Add the `--output` (`-o`) option to write the synthesized templates to a directory other than `cdk.out`.

```
cdk synth --output=~/templates
```

Deploying stacks

The `cdk deploy` subcommand deploys the specified stack(s) to your AWS account.

```
cdk deploy        # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"    # all stacks in app
```

> **Note**
> The CDK Toolkit runs your app and synthesizes fresh AWS CloudFormation templates before deploying anything. Therefore, most command line options you can use with `cdk synth` (for example, `--context`) can also be used with `cdk deploy`.

See `cdk deploy --help` for all available options. A few of the most-frequently-used options are covered below.

Specifying AWS CloudFormation parameters

The AWS CDK Toolkit supports specifying AWS CloudFormation parameters (p. 128) at deployment. You may provide these on the command line following the `--parameters` flag.

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

To define multiple parameters, use multiple `--parameters` flags.

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

If you are deploying multiple stacks, you can specify a different value of each parameter for each stack by prefixing the name of the parameter with the stack name and a colon. Otherwise, the same value is passed to all stacks.

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

By default, the AWS CDK retains values of parameters from previous deployments and uses them in later deployments if they are not specified explicitly. Use the `--no-previous-parameters` flag to require all parameters to be specified.

Specifying outputs file

If your stack declares AWS CloudFormation outputs, these are normally displayed on the screen at the conclusion of deployment. To write them to a file in JSON format, use the `--outputs-file` flag.

```
cdk deploy --outputs-file outputs.json MyStack
```

Security-related changes

To protect you against unintended changes that affect your security posture, the AWS CDK Toolkit prompts you to approve security-related changes before deploying them. You can specify the level of change that requires approval:

```
cdk deploy --require-approval LEVEL
```

`LEVEL` can be one of the following:

Term	Meaning
`never`	Approval is never required
`any-change`	Requires approval on any IAM or security-group-related change
`broadening` (default)	Requires approval when IAM statements or traffic rules are added; removals don't require approval

The setting can also be configured in the `cdk.json` file.

```
{
  "app": "...",
  "requireApproval": "never"
}
```

Comparing stacks

The `cdk diff` command compares the current version of a stack defined in your app with the already-deployed version, or with a saved AWS CloudFormation template, and displays a list of changes .

```
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
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3F:
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF93626:
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56CD69A:
 {"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
```

To compare your app's stack(s) with the existing deployment:

```
cdk diff MyStack
```

To compare your app's stack(s) with a saved CloudFormation template:

```
cdk diff --template ~/stacks/MyStack.old MyStack
```

Toolkit reference

This section provides a reference for the AWS CDK Toolkit derived from its help, first a general reference with the options available with all commands, then (in collapsible sections) specific references with options available only with specific subcommands.

```
Usage: cdk -a <cdk-app> COMMAND

Commands:

  cdk list [STACKS..]              Lists all stacks in the app      [aliases: ls]

  cdk synthesize [STACKS..]        Synthesizes and prints the CloudFormation
                                   template for this stack       [aliases: synth]

  cdk bootstrap [ENVIRONMENTS..]   Deploys the CDK toolkit stack into an AWS
                                   environment

  cdk deploy [STACKS..]            Deploys the stack(s) named STACKS into your
                                   AWS account

  cdk destroy [STACKS..]           Destroy the stack(s) named STACKS

  cdk diff [STACKS..]              Compares the specified stack with the deployed
                                   stack or a local template file, and returns
                                   with status 1 if any difference is found

  cdk metadata [STACK]             Returns all metadata associated with this
                                   stack

  cdk init [TEMPLATE]              Create a new, empty CDK project from a
                                   template.

  cdk context                      Manage cached context values

  cdk docs                         Opens the reference documentation in a browser
                                                                  [aliases: doc]

  cdk doctor                       Check your set-up for potential problems

Options:

  -a, --app                 REQUIRED: command-line for executing your app or a
                            cloud assembly directory (e.g. "node bin/my-app.js")
                                                                      [string]

  -c, --context             Add contextual string parameter (KEY=VALUE)  [array]

  -p, --plugin              Name or path of a node package that extend the CDK
```

```
                          features. Can be specified multiple times    [array]

    --trace               Print trace for stack warnings             [boolean]

    --strict              Do not construct stacks with warnings      [boolean]

    --lookups             Perform context lookups (synthesis fails if this is
                          disabled and context lookups need to be performed)
                                                  [boolean] [default: true]

    --ignore-errors       Ignores synthesis errors, which will likely produce
                          an invalid output         [boolean] [default: false]

-j, --json                Use JSON output instead of YAML when templates are
                          printed to STDOUT         [boolean] [default: false]

-v, --verbose             Show debug logs (specify multiple times to increase
                          verbosity)                  [count] [default: false]

    --debug               Enable emission of additional debugging information,
                          such as creation stack traces of tokens
                                                  [boolean] [default: false]

    --profile             Use the indicated AWS profile as the default
                          environment                                 [string]

    --proxy               Use the indicated proxy. Will read from HTTPS_PROXY
                          environment variable if not specified       [string]

    --ca-bundle-path      Path to CA certificate to use when validating HTTPS
                          requests. Will read from AWS_CA_BUNDLE environment
                          variable if not specified                   [string]

-i, --ec2creds            Force trying to fetch EC2 instance credentials.
                          Default: guess EC2 instance status         [boolean]

    --version-reporting   Include the "AWS::CDK::Metadata" resource in
                          synthesized templates (enabled by default) [boolean]

    --path-metadata       Include "aws:cdk:path" CloudFormation metadata for
                          each resource (enabled by default)
                                                  [boolean] [default: true]

    --asset-metadata      Include "aws:asset:*" CloudFormation metadata for
                          resources that uses assets (enabled by default)
                                                  [boolean] [default: true]

-r, --role-arn            ARN of Role to use when invoking CloudFormation
                                                                      [string]

    --toolkit-stack-name  The name of the CDK toolkit stack           [string]

    --staging             Copy assets to the output directory (use
                          --no-staging to disable, needed for local debugging
                          the source files with SAM CLI)
                                                  [boolean] [default: true]

-o, --output              Emits the synthesized cloud assembly into a
                          directory (default: cdk.out)                [string]

    --no-color            Removes colors and other style from console output
                                                  [boolean] [default: false]

    --version             Show version number                        [boolean]

-h, --help                Show help                                  [boolean]
```

```
If your app has a single stack, there is no need to specify the stack name

If one of cdk.json or ~/.cdk.json exists, options specified there will be used
as defaults. Settings in cdk.json take precedence.
```

cdk list (ls)

```
cdk list [STACKS..]

Lists all stacks in the app

Options:

  -l, --long                Display environment information for each stack
                                                    [boolean] [default: false]
```

cdk synthesize (synth)

```
cdk synthesize [STACKS..]

Synthesizes and prints the CloudFormation template for this stack

Options:

  -e, --exclusively         Only synthesize requested stacks, don't include
                            dependencies                              [boolean]

      --validation          After synthesis, validate stacks with the
                            "validateOnSynth" attribute set (can also be
                            controlled with CDK_VALIDATION)
                                                     [boolean] [default: true]

  -q, --quiet               Do not output CloudFormation Template to stdout
                                                    [boolean] [default: false]
```

cdk bootstrap

```
cdk bootstrap [ENVIRONMENTS..]

Deploys the CDK toolkit stack into an AWS environment

Options:

  -b, --bootstrap-bucket-name,              The name of the CDK toolkit bucket;
  --toolkit-bucket-name                     bucket will be created and must not
                                            exist                      [string]

      --bootstrap-kms-key-id                AWS KMS master key ID used for the
                                            SSE-KMS encryption         [string]

      --bootstrap-customer-key              Create a Customer Master Key (CMK)
                                            for the bootstrap bucket (you will
                                            be charged but can customize
                                            permissions, modern bootstrapping
                                            only)                      [boolean]

      --qualifier                           Unique string to distinguish
                                            multiple bootstrap stacks   [string]
```

```
      --public-access-block-configuration   Block public access configuration
                                            on CDK toolkit bucket (enabled by
                                            default)                  [boolean]

  -t, --tags                                Tags to add for the stack
                                            (KEY=VALUE)    [array] [default: []]

      --execute                             Whether to execute ChangeSet
                                            (--no-execute will NOT execute the
                                            ChangeSet) [boolean] [default: true]

      --trust                               The AWS account IDs that should be
                                            trusted to perform deployments into
                                            this environment (may be repeated,
                                            modern bootstrapping only)
                                                          [array] [default: []]

      --trust-for-lookup                    The AWS account IDs that should be
                                            trusted to look up values in this
                                            environment (may be repeated, modern
                                            bootstrapping only)
                                                          [array] [default: []]

      --cloudformation-execution-policies   The Managed Policy ARNs that should
                                            be attached to the role performing
                                            deployments into this environment
                                            (may be repeated, modern
                                            bootstrapping only)
                                                          [array] [default: []]

  -f, --force                               Always bootstrap even if it would
                                            downgrade template version
                                                     [boolean] [default: false]

      --termination-protection              Toggle CloudFormation termination
                                            protection on the bootstrap stacks
                                                                    [boolean]

      --show-template                       Instead of actual bootstrapping,
                                            print the current CLI's
                                            bootstrapping template to stdout for
                                            customization
                                                     [boolean] [default: false]

      --template                            Use the template from the given file
                                            instead of the built-in one (use
                                            --show-template to obtain an
                                            example)                   [string]
```

cdk deploy

```
cdk deploy [STACKS..]

Deploys the stack(s) named STACKS into your AWS account

Options:

      --all                 Deploy all available stacks
                                                 [boolean] [default: false]

  -E, --build-exclude       Do not rebuild asset with the given ID. Can be
                            specified multiple times      [array] [default: []]

  -e, --exclusively         Only deploy requested stacks, don't include
```

```
                            dependencies                                [boolean]

      --require-approval    What security-sensitive changes need manual
                            approval
                        [string] [choices: "never", "any-change", "broadening"]

      --ci                  Force CI detection       [boolean] [default: false]

      --notification-arns   ARNs of SNS topics that CloudFormation will notify
                            with stack related events                   [array]

  -t, --tags                Tags to add to the stack (KEY=VALUE), overrides
                            tags from Cloud Assembly (deprecated)       [array]

      --execute             Whether to execute ChangeSet (--no-execute will NOT
                            execute the ChangeSet)    [boolean] [default: true]

      --change-set-name     Name of the CloudFormation change set to create
                                                                       [string]

  -f, --force               Always deploy stack even if templates are identical
                                                  [boolean] [default: false]

      --parameters          Additional parameters passed to CloudFormation at
                            deploy time (STACK:KEY=VALUE) [array] [default: {}]

  -O, --outputs-file        Path to file where stack outputs will be written as
                            JSON                                        [string]

      --previous-parameters  Use previous values for existing parameters (you
                            must specify all parameters on every deployment if
                            this is disabled)         [boolean] [default: true]

      --progress            Display mode for stack activity events
                                          [string] [choices: "bar", "events"]

      --rollback            Rollback stack to stable state on failure (iterate
                            more rapidly with --no-rollback or -R)
                                                  [boolean] [default: true]
```

cdk destroy

```
cdk destroy [STACKS..]

Destroy the stack(s) named STACKS

Options:

      --all                 Destroy all available stacks
                                                  [boolean] [default: false]

  -e, --exclusively         Only destroy requested stacks, don't include
                            dependees                                   [boolean]

  -f, --force               Do not ask for confirmation before destroying the
                            stacks                                      [boolean]
```

cdk diff

```
cdk diff [STACKS..]

Compares the specified stack with the deployed stack or a local template file,
```

```
and returns with status 1 if any difference is found

Options:

  -e, --exclusively         Only diff requested stacks, don't include
                            dependencies                                [boolean]

      --context-lines       Number of context lines to include in arbitrary JSON
                            diff rendering                [number] [default: 3]

      --template            The path to the CloudFormation template to compare
                            with                                         [string]

      --security-only       Only diff for broadened security changes
                                                          [boolean] [default: false]

      --fail                Fail with exit code 1 in case of diff
                                                          [boolean] [default: false]
```

cdk init

```
cdk init [TEMPLATE]

Create a new, empty CDK project from a template.

Options:

  -l, --language            The language to be used for the new project (default
                            can be configured in ~/.cdk.json)
    [string] [choices: "csharp", "fsharp", "go", "java", "javascript", "python",
                                                                   "typescript"]

      --list                List the available templates               [boolean]

      --generate-only       If true, only generates project files, without
                            executing additional operations such as setting up a
                            git repo, installing dependencies or compiling the
                            project                   [boolean] [default: false]
```

cdk context

```
cdk context

Manage cached context values

Options:

  -e, --reset               The context key (or its index) to reset     [string]

      --clear               Clear all context                          [boolean]
```

AWS Toolkit for Visual Studio Code

The AWS Toolkit for Visual Studio Code is an open source plug-in for Visual Studio Code that makes it easier to create, debug, and deploy applications on AWS. The toolkit provides an integrated experience for developing AWS CDK applications, including the AWS CDK Explorer feature to list your AWS CDK projects and browse the various components of the CDK application. Install the AWS Toolkit and learn more about using the AWS CDK Explorer.

SAM CLI

This topic describes how to use the AWS SAM CLI with the AWS CDK to test a Lambda function locally. For further information, see Invoking Functions Locally. To install the SAM CLI, see Installing the AWS SAM CLI.

> **Note**
> Full AWS CDK integration with the AWS SAM CLI is currently in public preview. This integration allows you to locally test and build serverless application defined using the CDK. For more information, see AWS Cloud Development Kit (CDK) in the AWS SAM Developer Guide.
> The instructions here apply to the current (non-preview) version of the AWS SAM CLI.

1. The first step is to create a AWS CDK application and add the Lambda package.

```
mkdir cdk-sam-example
cd cdk-sam-example
cdk init app --language typescript
npm install @aws-cdk/aws-lambda
```

2. Add a Lambda reference to `lib/cdk-sam-example-stack.ts`:

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

3. Replace the comment in `lib/cdk-sam-example-stack.ts` with the following Lambda function:

```
new lambda.Function(this, 'MyFunction', {
  runtime: lambda.Runtime.PYTHON_3_7,
  handler: 'app.lambda_handler',
  code:    lambda.Code.asset('./my_function'),
});
```

4. Create the directory `my_function`

```
mkdir my_function
```

5. Create the file `app.py` in `my_function` with the following content:

```
def lambda_handler(event, context):
  return "This is a Lambda Function defined through CDK"
```

6. Run your AWS CDK app and create a AWS CloudFormation template

```
cdk synth --no-staging > template.yaml
```

7. Find the logical ID for your Lambda function in `template.yaml`. It will look like `MyFunction`*`12345678`*, where *`12345678`* represents an 8-character unique ID that the AWS CDK generates for all resources. The line right after it should look like:

```
Type: AWS::Lambda::Function
```

8. Run the function by executing:

```
sam local invoke MyFunction12345678 --no-event
```

The output should look something like the following.

```
2019-04-01 12:22:41 Found credentials in shared credentials file: ~/.aws/credentials
```

```
2019-04-01 12:22:41 Invoking app.lambda_handler (python3.7)

Fetching lambci/lambda:python3.7 Docker container image......
2019-04-01 12:22:43 Mounting D:\cdk-sam-example\.cdk.staging
\a57f59883918e662ab3c46b964d2faa5 as /var/task:ro,delegated inside runtime container
START RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72 Version: $LATEST
END RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72
REPORT RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72     Duration: 3.70 ms       Billed
 Duration: 100 ms Memory Size: 128 MB     Max Memory Used: 22 MB

"This is a Lambda Function defined through CDK"
```

Testing constructs

With the AWS CDK, your infrastructure can be as testable as any other code you write. This article illustrates one approach to testing AWS CDK apps written in TypeScript using the Jest test framework. Currently, TypeScript is the only supported language for testing AWS CDK infrastructure, though we intend to eventually make this capability available in all languages supported by the AWS CDK.

There are three categories of tests you can write for AWS CDK apps.

- **Snapshot tests** test the synthesized AWS CloudFormation template against a previously-stored baseline template. This way, when you're refactoring your app, you can be sure that the refactored code works exactly the same way as the original. If the changes were intentional, you can accept a new baseline for future tests.
- **Fine-grained assertions** test specific aspects of the generated AWS CloudFormation template, such as "this resource has this property with this value." These tests help when you're developing new features, since any code you add will cause your snapshot test to fail even if existing features still work. When this happens, your fine-grained tests will reassure you that the existing functionality is unaffected.
- **Validation tests** help you "fail fast" by making sure your AWS CDK constructs raise errors when you pass them invalid data. The ability to do this type of testing is a big advantage of developing your infrastructure in a general-purpose programming language.

Getting started

As an example, we'll create a dead letter queue construct. A dead letter queue holds messages from another queue that have failed delivery for some time. This usually indicates failure of the message processor, which we want to know about, so our dead letter queue has an alarm that fires when a message arrives. The user of the construct can hook up actions such as notifying an Amazon SNS topic to this alarm.

Creating the construct

Start by creating an empty construct library project using the AWS CDK Toolkit and installing the construct libraries we'll need:

```
mkdir dead-letter-queue && cd dead-letter-queue
cdk init --language=typescript lib
npm install @aws-cdk/aws-sqs @aws-cdk/aws-cloudwatch
```

Place the following code in `lib/index.ts`:

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

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(),
    });
  }
}
```

Installing the testing framework

Since we're using the Jest framework, our next setup step is to install Jest. We'll also need the AWS CDK assert module, which includes helpers for writing tests for CDK libraries, including `assert` and `expect`.

```
npm install --save-dev jest @types/jest @aws-cdk/assert
```

Updating `package.json`

Finally, edit the project's `package.json` to tell NPM how to run Jest, and to tell Jest what kinds of files to collect. The necessary changes are as follows.

- Add a new `test` key to the `scripts` section
- Add Jest and its types to the `devDependencies` section
- Add a new `jest` top-level key with a `moduleFileExtensions` declaration

These changes are shown in outline below. Place the new text where indicated in `package.json`. The "..." placeholders indicate existing parts of the file that should not be changed.

```
{
  ...
 "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "@types/jest": "^24.0.18",
    "jest": "^24.9.0",
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

Snapshot tests

Add a snapshot test by placing the following code in `test/dead-letter-queue.test.ts`.

```
import { SynthUtils } from '@aws-cdk/assert';
import { Stack } from '@aws-cdk/core';

import * as dlq from '../lib/index';
```

```
test('dlq creates an alarm', () => {
  const stack = new Stack();
  new dlq.DeadLetterQueue(stack, 'DLQ');
  expect(SynthUtils.toCloudFormation(stack)).toMatchSnapshot();
});
```

To build the project and run the test, issue these commands.

```
npm run build && npm test
```

The output from Jest indicates that it has run the test and recorded a snapshot.

```
PASS  test/dead-letter-queue.test.js
 # dlq creates an alarm (55ms)
 › 1 snapshot written.
Snapshot Summary
› 1 snapshot written
```

Jest stores the snapshots in a directory named `__snapshots__` inside the project. In this directory is a copy of the AWS CloudFormation template generated by the dead letter queue construct. The beginning looks something like this.

```
exports[`dlq creates an alarm 1`] = `
Object {
  "Resources": Object {
    "DLQ581697C4": Object {
      "Type": "AWS::SQS::Queue",
    },
    "DLQAlarm008FBE3A": Object {
     "Properties": Object {
        "AlarmDescription": "There are messages in the Dead Letter Queue",
        "ComparisonOperator": "GreaterThanOrEqualToThreshold",
...
```

Testing the test

To make sure the test works, change the construct so that it generates different AWS CloudFormation output, then build and test again. For example, add a `period` property of 1 minute to override the default of 5 minutes.

```
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
    alarmDescription: 'There are messages in the Dead Letter Queue',
    evaluationPeriods: 1,
    threshold: 1,
    metric: this.metricApproximateNumberOfMessagesVisible(),
    period: Duration.minutes(1),
});
```

Build the project and run the tests again.

```
npm run build && npm test
```

```
FAIL test/dead-letter-queue.test.js
# dlq creates an alarm (58ms)

# dlq creates an alarm
```

```
expect(received).toMatchSnapshot()

Snapshot name: `dlq creates an alarm 1`

- Snapshot
+ Received

@@ -19,11 +19,11 @@
                },
              ],
              "EvaluationPeriods": 1,
              "MetricName": "ApproximateNumberOfMessagesVisible",
              "Namespace": "AWS/SQS",
     -        "Period": 300,
     +        "Period": 60,
              "Statistic": "Maximum",
              "Threshold": 1,
            },
            "Type": "AWS::CloudWatch::Alarm",
          },

 › 1 snapshot failed.
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect your code changes or run `npm test -- -u`
 to update them.
```

Accepting the new snapshot

Jest has told us that the `Period` attribute of the synthesized AWS CloudFormation template has changed from 300 to 60. To accept the new snapshot, issue:

```
npm test -- -u
```

Now we can run the test again and see that it passes.

Limitations

Snapshot tests are easy to create and are a powerful backstop when refactoring. They can serve as an early warning sign that more testing is needed. Snapshot tests can even be useful for test-driven development: modify the snapshot to reflect the result you're aiming for, and adjust the code until the test passes.

The chief limitation of snapshot tests is that they test the *entire* template. Consider that our dead letter queue uses the default retention period. To give ourselves as much time as possible to recover the undelivered messages, for example, we might set the queue's retention time to the maximum—14 days—by changing the code as follows.

```
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string) {
    super(scope, id, {
      // Maximum retention period
      retentionPeriod: Duration.days(14)
    });
```

When we run the test again, it breaks. The name we've given the test hints that we are interested mainly in testing whether the alarm is created, but the snapshot test also tests whether the queue is created

with default options—along with literally everything else about the synthesized template. This problem is magnified when a project contains many constructs, each with a snapshot test.

Fine-grained assertions

To avoid needing to review every snapshot whenever you make a change, use the custom assertions in the `@aws-cdk/assert/jest` module to write fine-grained tests that verify only part of the construct's behavior. For example, the test we called "dlq creates an alarm" in our example really should assert only that an alarm is created with the appropriate metric.

The AWS::CloudWatch::Alarm resource specification reveals that we're interested in the properties Namespace, MetricName and Dimensions. We'll use the `expect(stack).toHaveResource(...)` assertion, which is in the `@aws-cdk/assert/jest` module, to make sure these properties have the appropriate values.

Replace the code in `test/dead-letter-queue.test.ts` with the following.

```
import { Stack } from '@aws-cdk/core';
import '@aws-cdk/assert/jest';

import * as dlq from '../lib/index';

test('dlq creates an alarm', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ');

  expect(stack).toHaveResource('AWS::CloudWatch::Alarm', {
    MetricName: "ApproximateNumberOfMessagesVisible",
    Namespace: "AWS/SQS",
    Dimensions: [
      {
        Name: "QueueName",
        Value: { "Fn::GetAtt": [ "DLQ581697C4", "QueueName" ] }
      }
    ],
  });
});

test('dlq has maximum retention period', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ');

  expect(stack).toHaveResource('AWS::SQS::Queue', {
    MessageRetentionPeriod: 1209600
  });
});
```

There are now two tests. The first checks that the dead letter queue creates an alarm on its `ApproximateNumberOfMessagesVisible` metric. The second verifies the message retention period.

Again, build the project and run the tests.

```
npm run build && npm test
```

Note
Since we've replaced the snapshot test, the first time we run the new tests, Jest reminds us that we have a snapshot that is not used by any test. Issue `npm test -- -u` to tell Jest to clean it up.

Validation tests

Suppose we want to make the dead letter queue's retention period configurable. Of course, we also want to make sure that the value provided by the user of the construct is within an allowable range. We can write a test to make sure that the validation logic works: pass in invalid values and see what happens.

First, create a `props` interface for the construct.

```
export interface DeadLetterQueueProps {
    /**
     * The amount of days messages will live in the dead letter queue
     *
     * Cannot exceed 14 days.
     *
     * @default 14
     */
    retentionDays?: number;
}

export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string, props: DeadLetterQueueProps = {}) {
    if (props.retentionDays !== undefined && props.retentionDays > 14) {
      throw new Error('retentionDays may not exceed 14 days');
    }

    super(scope, id, {
        // Given retention period or maximum
        retentionPeriod: Duration.days(props.retentionDays || 14)
    });
    // ...
  }
}
```

To test that the new feature actually does what we expect, we write two tests:

- One that makes sure the configured value ends up in the template
- One that supplies an incorrect value to the construct and checks it raises the expected error

Add the following to `test/dead-letter-queue.test.ts`.

```
test('retention period can be configured', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ', {
    retentionDays: 7
  });

  expect(stack).toHaveResource('AWS::SQS::Queue', {
    MessageRetentionPeriod: 604800
  });
});

test('configurable retention period cannot exceed 14 days', () => {
  const stack = new Stack();

  expect(() => {
    new dlq.DeadLetterQueue(stack, 'DLQ', {
      retentionDays: 15
```

```
    });
  }).toThrowError(/retentionDays may not exceed 14 days/);
});
```

Run the tests to confirm the construct behaves as expected.

```
npm run build && npm test
```

```
PASS  test/dead-letter-queue.test.js
  # dlq creates an alarm (62ms)
  # dlq has maximum retention period (14ms)
  # retention period can be configured (18ms)
  # configurable retention period cannot exceed 14 days (1ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
```

Tips for tests

Remember, your tests will live just as long as the code they test, and be read and modified just as often, so it pays to take a moment to consider how best to write them. Don't copy and paste setup lines or common assertions, for example; refactor this logic into helper functions. Use good names that reflect what each test actually tests.

Don't assert too much in one test. Preferably, a test should test one and only one behavior. If you accidentally break that behavior, exactly one test should fail, and the name of the test should tell you exactly what failed. This is more an ideal to be striven for, however; sometimes you will unavoidably (or inadvertently) write tests that test more than one behavior. Snapshot tests are, for reasons we've already described, especially prone to this problem, so use them sparingly.

Security for the AWS Cloud Development Kit (AWS CDK)

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The Shared Responsibility Model describes this as Security of the Cloud and Security in the Cloud.

Security of the Cloud – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Topics

Identity and access management for the AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) is an Amazon Web Services (AWS) service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use resources in AWS services. IAM is an AWS service that you can use with no additional charge.

To use the AWS CDK to access AWS, you need an AWS account and AWS credentials. To increase the security of your AWS account, we recommend that you use an *IAM user* to provide access credentials instead of using your AWS account credentials.

For details about working with IAM, see AWS Identity and Access Management.

For an overview of IAM users and why they are important for the security of your account, see AWS Security Credentials in the Amazon Web Services General Reference.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Compliance validation for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

The security and compliance of AWS services is assessed by third-party auditors as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using the AWS CDK to access an AWS service is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of an AWS service is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security-focused and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience for the AWS Cloud Development Kit (AWS CDK)

The Amazon Web Services (AWS) global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Infrastructure security for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Troubleshooting common AWS CDK issues

This topic describes how to troubleshoot the following issues with the AWS CDK.

After updating the AWS CDK, code that used to work fine now results in errors

Errors in code that used to work is typically a symptom of having mismatched versions of AWS Construct Library modules. Make sure all library modules are the same version and up-to-date.

An error message commonly seen in this situation is `unable to determine cloud assembly output directory. Assets must be defined indirectly within a "Stage" or an "App" scope`.

The modules that make up the AWS Construct Library are a matched set. They are released together and are intended to be used together. Interfaces between modules are considered private; we may change them when necessary to implement new features in the library.

We also update the libraries that are used by the AWS Construct Library from time to time, and different versions of the library modules may have incompatible dependencies. Synchronizing the versions of the library modules will also address this issue.

JSII is an important AWS CDK dependency, especially if you are using the AWS CDK in a language other than TypeScript or JavaScript. You do not ordinarily have to concern yourself with the JSII versions, since it is a declared dependency of all AWS CDK modules. If a compatible version is not installed, however, you can see unexpected type-relatd errors, such as `'undefined' is not a valid TargetType`. Making sure all AWS CDK modules are the same version will resolve JSII compatibility issues, since they will all depend on the same JSII version.

Below, you'll find details on managing the versions of your installed AWS Construct Library modules in TypeScript, JavaScript, Python, Java, and C#.

TypeScript/JavaScript

Install your project's AWS Construct Library modules locally (the default). Use `npm` to install the modules and keep them up to date.

To see what needs to be updated:

```
npm outdated
```

To actually update the modules to the latest version:

```
npm update
```

If you are working with a specific older version of the AWS Construct Library, rather than the latest, first uninstall all of your project's `@aws-cdk` modules, then reinstall the specific version you want to use. For example, to install version 1.9.0 of the Amazon S3 module, use:

```
npm uninstall @aws-cdk/aws-s3
npm install @aws-cdk/aws-s3@1.9.0
```

Repeat these commands for each module your project uses.

You can edit your `package.json` file to lock the AWS Construct Library modules to a specific version, so `npm update` won't update them. You can also specify a version using ~ or ^ to allow modules to be updated to versions that are API-compatible with the current version, such as `^1.0.0` to accept any update API-compatible with version 1.x. Use the same version specification for all AWS Construct Library modules within a project, including these special characters. Otherwise, Node.js may not resolve all these specifications to the same concrete version, resulting in a mismatch.

Python

Use a virtual environment to manage your project's AWS Construct Library modules. For your convenience, `cdk init` creates a virtual environment for new Python projects in the project's `.venv` directory.

Add the AWS Construct Library modules your project uses to its `requirements.txt` file. Use the = syntax to specify an exact version, or the ~= syntax to constrain updates to versions without breaking API changes. For example, the following specifies the latest version of the listed modules that are API-compatible with version 1.x:

```
aws-cdk.core~=1.0
aws-cdk.aws-s3~=1.0
```

If you wanted to accept only bug-fix updates to, for example, version 1.9.0, you could instead specify `~=1.9.0`. Use the same version specification for all AWS Construct Library modules within a single project.

Use `pip` to install and update the modules.

To see what needs to be updated:

```
pip list --local --outdated
```

To actually update the modules to the latest compatible version:

```
pip install --upgrade -r requirements.txt
```

If your project requires a specific older version of the AWS Construct Library, rather than the latest, first uninstall all of your project's `aws-cdk` modules. Edit `requirements.txt` to specify the exact versions of the modules you want to use using =, then install from `requirements.txt`.

```
pip install -r requirements.txt
```

Java

Add your project's AWS Construct Library modules as dependencies in your project's `pom.xml`. You may specify an exact version, or use Maven's range syntax to specify a range of allowable versions.

For example, to specify an exact version of a dependency:

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>1.23.0</version>
</dependency>
```

To specify that any 1.x.x version is acceptable (note use of right parenthesis to indicate that the end of the range excludes version 2.0.0):

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>[1.0.0,2.0.0)</version>
</dependency>
```

Maven automatically downloads and installs the latest versions that allow all requirements to be fulfilled when you build your application.

If you prefer to pin dependencies to a specific version, you can issue `mvn versions:use-latest-versions` to rewrite the version specifications in `pom.xml` to the latest available versions when you decide to upgrade.

C#

Use the Visual Studio NuGet GUI (**Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**) to install the desired version of your application's AWS Construct Library modules.

- The **Installed** panel shows you what modules are currently installed; you can install any available version of any module from this page.
- The **Updates** panel shows you modules for which updates are available, and lets you update some or all of them.

(back to list (p. 308))

After updating the AWS CDK, the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library

The version of the AWS CDK Toolkit (which provides the `cdk` command) must be at least equal to the version of the AWS Construct Library. The Toolkit is intended to be backward compatible within the same major version; the latest 1.x version of the toolkit can be used with any 1.x release of the library. For this reason, we recommend you install this component globally and keep it up-to-date.

```
npm update -g aws-cdk
```

If, for some reason, you need to work with multiple versions of the AWS CDK Toolkit, you can install a specific version of the toolkit locally in your project folder.

If you are using a language other than TypeScript or JavaScript, first create a `node_modules` folder in your project directory. Then, regardless of language, use `npm` to install the AWS CDK Toolkit, omitting the `-g` flag and specifying the desired version. For example:

```
npm install aws-cdk@1.50.0
```

To run a locally-installed AWS CDK Toolkit, use the command `npx cdk` rather than just `cdk`. For example:

```
npx cdk deploy MyStack
```

`npx cdk` runs the local version of the AWS CDK Toolkit if one exists, and falls back to the global version when a project doesn't have a local installation. You may find it convenient to set up a shell alias or batch file to make sure `cdk` is always invoked this way. For example, Linux users might add the following statement to their `.bash_profile` file.

```
alias cdk=npx cdk
```

(back to list (p. 308))

When deploying my AWS CDK stack, I receive a `NoSuchBucket` error

Your AWS environment does not have a staging bucket, which the AWS CDK uses to hold resources during deployment. Stacks require this bucket if they contain the section called “Assets” (p. 138) or synthesize to AWS CloudFormation templates larger than 50 kilobytes. You can create the staging bucket with the following command:

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

To avoid generating unexpected AWS charges, the AWS CDK does not automatically create a staging bucket. You must bootstrap your environment explicitly.

By default, the staging bucket is created in the region(s) used by stacks in the current AWS CDK application, or the region specified in your local AWS profile (set by `aws configure`), using that profile's account. You can specify a different account and region on the command line as follows. (You must specify the account and region if you are not in an app's directory.)

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

You must bootstrap in every region where you will deploy stacks that require a staging bucket.

For more information, see the section called “Bootstrapping” (p. 174)

(back to list (p. 308))

When deploying my AWS CDK stack, I receive a `forbidden: null` message

You are deploying a stack that requires the use of a staging bucket, but are using an IAM role or account that lacks permission to write to it. (The staging bucket is used when deploying stacks that contain assets or that synthesize an AWS CloudFormation template larger than 50K.) Use an account or role that has permission to perform the action `s3:*` against the resource `arn:aws:s3:::cdktoolkit-stagingbucket-*`.

(back to list (p. 308))

When synthesizing an AWS CDK stack, I get the message `--app is required either in command-line, in cdk.json or in ~/.cdk.json`

This message usually means that you aren't in the main directory of your AWS CDK project when you issue `cdk synth`. The file `cdk.json` in this directory, created by the `cdk init` command, contains

the command line needed to run (and thereby synthesize) your AWS CDK app. For a TypeScript app, for example, the default `cdk.json` looks something like this:

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

We recommend issuing `cdk` commands only in your project's main directory, so the AWS CDK toolkit can find `cdk.json` there and successfully run your app.

If this isn't practical for some reason, the AWS CDK Toolkit looks for the app's command line in two other locations:

- in `cdk.json` in your home directory
- on the `cdk synth` command itself using the `-a` option

For example, you might synthesize a stack from a TypeScript app as follows.

```
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

(back to list (p. 308))

When synthesizing an AWS CDK stack, I receive an error because the AWS CloudFormation template contains too many resources

The AWS CDK generates and deploys AWS CloudFormation templates. AWS CloudFormation has a hard limit on the number of resources a stack can contain. With the AWS CDK, you can run up against this limit more quickly than you might expect.

> **Note**
> The AWS CloudFormation resource limit is 500 at this writing. See AWS CloudFormation quotas for the current resource limit.

The AWS Construct Library's higher-level, intent-based constructs automatically provision any auxiliary resources that are needed for logging, key management, authorization, and other purposes. For example, granting one resource access to another generates any IAM objects needed for the relevant services to communicate.

In our experience, real-world use of intent-based constructs results in 1–5 AWS CloudFormation resources per construct, though this can vary. For serverless applications, 5–8 AWS resources per API endpoint is typical.

Patterns, which represent a higher level of abstraction, let you define even more AWS resources with even less code. The AWS CDK code in the section called “ECS” (p. 215), for example, generates more than fifty AWS CloudFormation resources while defining only three constructs!

Exceeding the AWS CloudFormation resource limit is an error during AWS CloudFormation synthesis. The AWS CDK issues a warning if your stack exceeds 80% of the limit. You can use a different limit by setting the `maxResources` property on your stack, or disable validation by setting `maxResources` to 0.

> **Tip**
> You can get an exact count of the resources in your synthesized output using the following utility script. (Since every AWS CDK developer needs Node.js, the script is written in 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");
```

As your stack's resource count approaches the limit, consider re-architecting to reduce the number of resources your stack contains: for example, by combining some Lambda functions, or by breaking your stack into multiple stacks. The CDK supports references between stacks (p. 101), so it is straightforward to separate your app's functionality into different stacks in whatever way makes the most sense to you.

> **Note**
> AWS CloudFormation experts often suggest the use of nested stacks as a solution to the resource limit. The AWS CDK supports this approach via the `NestedStack` (p. 91) construct.

(back to list (p. 308))

I specified three (or more) Availability Zones for my EC2 Auto-Scaling Group or Virtual Private Cloud, but it was only deployed in two

To get the number of Availability Zones you requested, specify the account and region in the stack's `env` property. If you do not specify both, the AWS CDK, by default, synthesizes the stack as environment-agnostic, such that it can be deployed to any region. You can then deploy the stack to a specific region using AWS CloudFormation. Because some regions have only two availability zones, an environment-agnostic template never uses more than two.

> **Note**
> In the past, regions have occasionally launched with only one availability zone. Environment-agnostic AWS CDK stacks cannot be deployed to such regions. At this writing, however, all AWS regions have at least two AZs.

You can change this behavior by overriding your stack's `availablilityZones` (Python: `availability_zones`) property to explicitly specify the zones you want to use.

For more information about specifying a stack's account and region at synthesis time, while retaining the flexibility to deploy to any region, see the section called “Environments” (p. 92).

(back to list (p. 308))

My S3 bucket, DynamoDB table, or other resource is not deleted when I issue `cdk destroy`

By default, resources that can contain user data have a `removalPolicy` (Python: `removal_policy`) property of `RETAIN`, and the resource is not deleted when the stack is destroyed. Instead, the resource is orphaned from the stack. You must then delete the resource manually after the stack is destroyed. Until you do, redeploying the stack fails, because the name of the new resource being created during deployment conflicts with the name of the orphaned resource.

If you set a resource's removal policy to `DESTROY`, that resource will be deleted when the stack is destroyed.

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,
    });
  }
}
```

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
    });
  }
}

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)
```

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).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 cannot delete a non-empty Amazon S3 bucket. If you set an Amazon S3 bucket's removal policy to `DESTROY`, and it contains data, attempting to destroy the stack will fail because the bucket cannot be deleted. You can have the AWS CDK delete the objects in the bucket before attempting to destroy it by setting the bucket's `autoDeleteObjects` prop to `true`.

(back to list (p. 308))

OpenPGP keys for the AWS CDK and JSII

This topic contains the OpenPGP keys for the AWS CDK and JSII.

AWS CDK OpenPGP key

Key ID:	0x0566A784E17F3870
Type:	RSA
Size:	4096/4096
Created:	2018-06-19
Expires:	2022-06-18
User ID:	AWS CDK Team <aws-cdk@amazon.com>
Key fingerprint:	E88B E3B6 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

Select the "Copy" icon to copy the following OpenPGP key:

```
-----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 key

Key ID:	0x1C7ACE4CB2A1B93A
Type:	RSA
Size:	4096/4096
Created:	2018-08-06
Expires:	2022-08-05
User ID:	AWS JSII Team <aws-jsii@amazon.com>
Key fingerprint:	85EF 6522 4CE2 1E8C 72DB 28EC 1C7A CE4C B2A1 B93A

Select the "Copy" icon to copy the following OpenPGP key:

```
-----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 Developer Guide history

See Releases for information about AWS CDK releases. The AWS CDK is updated approximately once a week. Maintenance versions may be released between weekly releases to address critical issues. Each release includes a matched AWS CDK Toolkit (CDK CLI), AWS Construct Library, and API Reference. Updates to this Guide generally do not synchronize with AWS CDK releases.

Note
The table below represents significant documentation milestones. We fix errors and improve content on an ongoing basis.

update-history-change	update-history-description	update-history-date
Go developer preview (p. 318)	Add information about Go language support (in developer preview) including a "Working with the CDK in Go" topic.	April 19, 2021
Import or migrate AWS CloudFormation template (p. 318)	Add description of the new `cloudformation-include.CfnInclude` class, a powerful way to import and migrate AWS CloudFormation templates, or to vend them as AWS CDK constructs.	October 15, 2020
Add construct tree section (p. 318)	Information about the construct tree and its relation to the constructs you define in your AWS CDK app.	October 5, 2020
Add Bootstrapping topic (p. 318)	A complete explanation of why and how to bootstrap AWS environments for the CDK, including a comprehensive discussion of how to customize the process.	September 8, 2020
Add CDK Pipelines how-to (p. 318)	CDK Pipelines let you easily automate the deployment of your AWS CDK apps from source control whenever they're updated.	July 17, 2020
Improve CDK Toolkit topic (p. 318)	Include more information and examples around performing common tasks with the CLI (and the relevant flags) rather than just including a copy of the help.	July 9, 2020
Improve CodePipeline example (p. 318)	Update pipeline stack to build in proper language and add more material dealing with the CodeCommit repository.	July 6, 2020

Improve Getting Started (p. 318)	Remove extraneous material from Getting Started, use a more conversational tone, incorporate current best practices. Break out Hello World into its own topic.	June 17, 2020
Update stability index (p. 318)	Incorporate the latest definitions of the stability levels for AWS Construct Library modules.	June 11, 2020
CDK Toolkit versioning (p. 318)	Add information about cloud assembly versioning and compatibility of the CDK Toolkit (CLI) with the AWS Construct Library	April 22, 2020
Translating from TypeScript (p. 318)	Updated "CDK in Other Languages" topic to also include JavaScript, Java, and C# and renamed it "Translating from TypeScript."	April 10, 2020
Parameters topic (p. 318)	Add Concepts topic on using parameters with the AWS CDK.	April 8, 2020
JavaScript code snippets (p. 318)	Reinstate JavaScript code snippets throughout (automated translation via Babel).	April 3, 2020
Working with the CDK (p. 318)	Add "Working with the CDK" articles for the five supported languages. Various other improvements and fixes.	February 4, 2020
Java code snippets (p. 318)	Add Java code snippets throughout. Designate Java and C# bindings stable.	November 25, 2019
C# code snippets (p. 318)	Add C# code snippets throughout.	November 19, 2019
Python code snippets (p. 318)	Add Python code snippets throughout. Add Troubleshooting and Testing topics.	November 14, 2019
Troubleshooting topic (p. 318)	Add Troubleshooting topic to AWS CDK Developer Guide.	October 30, 2019
Improve Environments topic (p. 318)	Add Troubleshooting topic to AWS CDK Developer Guide.	October 10, 2019
ECS Patterns improvements (p. 318)	Updates to reflect improvements to ECS Patterns module.	September 17, 2019
New tagging API (p. 318)	Update tagging topic to use new API.	August 13, 2019
General availability (p. 318)	The AWS CDK Developer Guide is released.	July 11, 2019