AWS Cloud Development Kit (AWS CDK)

Developer Guide

Version latest

AWS Cloud Development Kit (AWS CDK): Developer Guide

Copyright © 2019 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

This documentation is for the developer preview release (public beta) of the AWS Cloud Development Kit (AWS CDK). Releases might lack important features and might have future breaking changes.

What Is the AWS Cloud Development Kit?

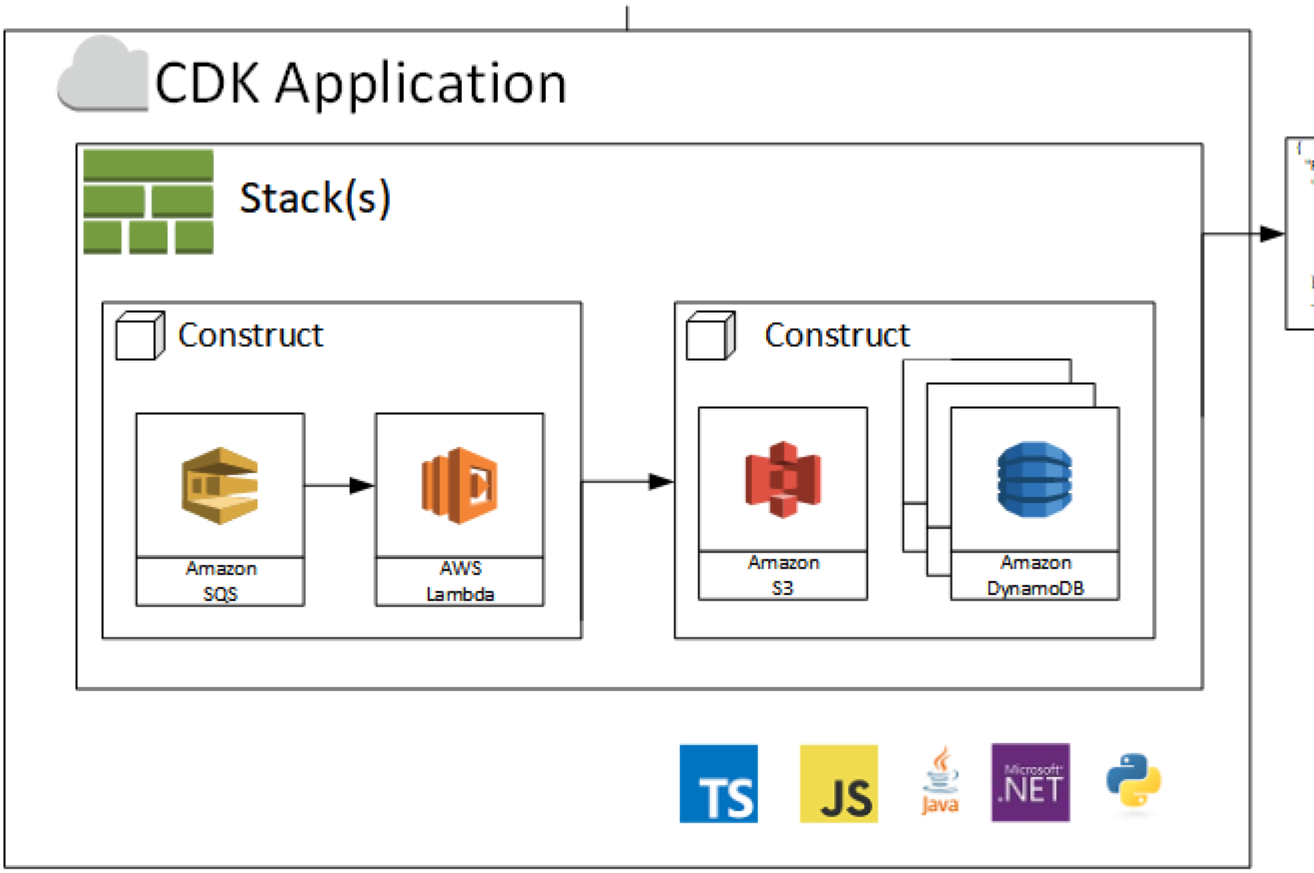
Welcome to the *AWS Cloud Development Kit (AWS CDK) Developer Guide*. This document provides information about the AWS CDK, which is a software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.

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 using one of the supported programming languages: C#/.NET, Java, JavaScript, Python, or TypeScript. This document does not supply reference information for the AWS CDK. You can find that information in the AWS CDK Reference.

Developers can use one of the supported programming languages to define reusable cloud components known as Constructs (p. 18). You compose these together into Stacks (p. 22) and Apps (p. 22).



Why Use the AWS CDK?

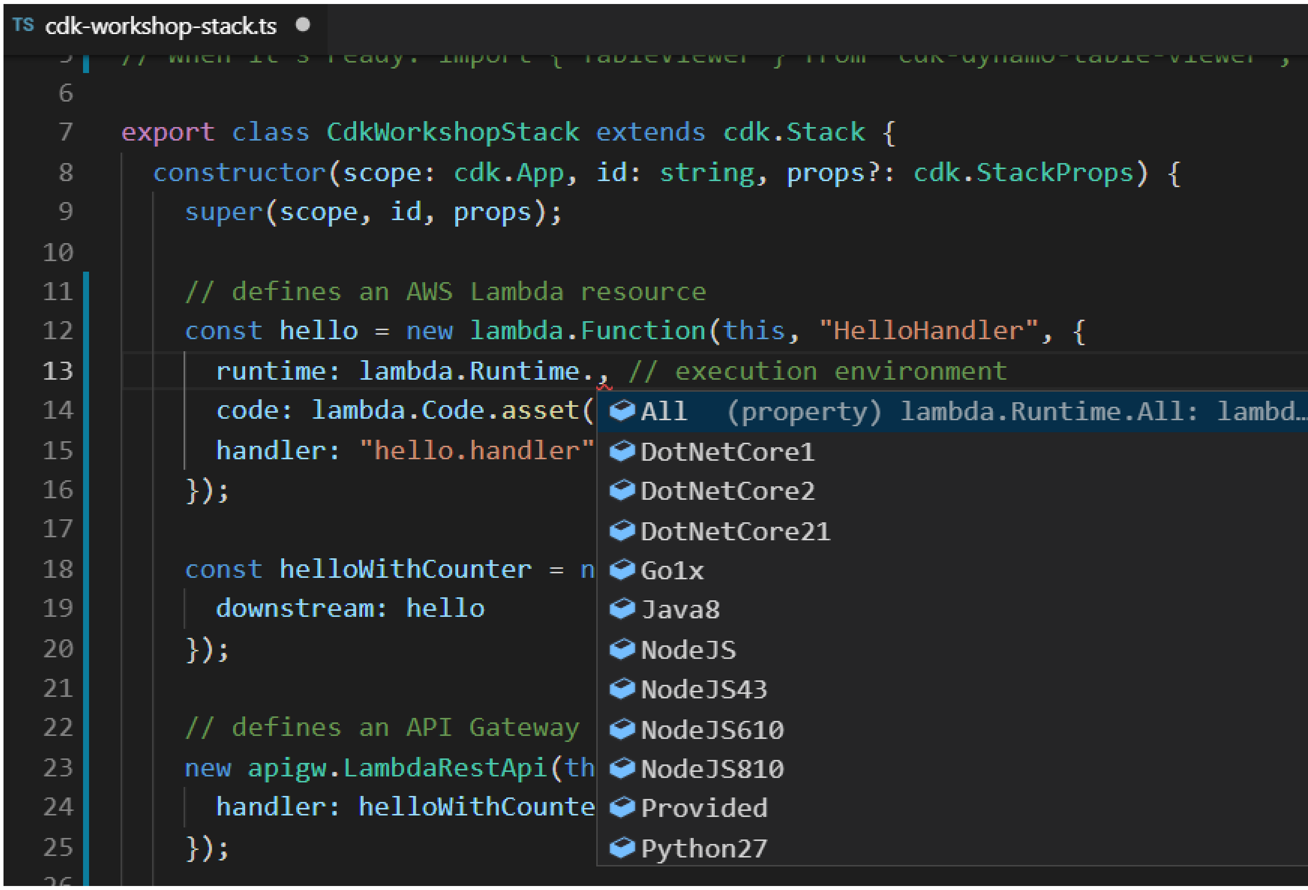
Let's look at the power of the AWS CDK. Here is some TypeScript code in an AWS CDK project to create an AWS Fargate service (this is the code we use in the Creating an AWS Fargate Service Using the AWS CDK (p. 45)).

```
export class MyEcsConstructStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.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.LoadBalancedFargateService(this, 'MyFargateService', {
      cluster: cluster,  // Required
      cpu: '512', // Default is 256
      desiredCount: 6,  // Default is 1
      image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"), // Required
      memoryMiB: '2048',  // Default is 512
      publicLoadBalancer: true  // Default is false
    });
```

This produces an AWS CloudFormation template of over 600 lines. We'll show the first 25 lines and Outputs of a **cdk synth** command.

```
Resources:
  MyVpcF9F0CA6F:
    Type: AWS::EC2::VPC
    Properties:
      CidrBlock: 10.0.0.0/16
      EnableDnsHostnames: true
      EnableDnsSupport: true
      InstanceTenancy: default
      Tags:
        - Key: Name
          Value: MyEcsConstruct/MyVpc
    Metadata:
      aws:cdk:path: MyEcsConstruct/MyVpc/Resource
  MyVpcPublicSubnet1SubnetF6608456:
    Type: AWS::EC2::Subnet
    Properties:
      CidrBlock: 10.0.0.0/19
      VpcId:
        Ref: MyVpcF9F0CA6F
      AvailabilityZone: us-west-2a
      MapPublicIpOnLaunch: true
      Tags:
        - Key: Name
          Value: MyEcsConstruct/MyVpc/PublicSubnet1
        - Key: aws-cdk:subnet-name
...
  MyFargateServiceLoadBalancerDNS704F6391:
    Value:
      Fn::GetAtt:
        - MyFargateServiceLBDE830E97
        - DNSName
```

Another advantage of IAC (infrastructure as code) is that you get code completion within your IDE.



Developing with the AWS CDK

Unless otherwise indicated, the code examples in this guide are in TypeScript. To aid you in porting a TypeScript example to a supported programming language, see Multi-Language Support in the AWS CDK (p. 33). The AWS CDK also includes examples in the supported programming languages. See AWS CDK Examples (p. 47) for a list of the examples.

The AWS CDK Toolchain (p. 58) 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. 35) 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, however 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 Simple Monthly 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 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:

- Reference
- AWS CDK Demo at re:Invent 2018
- AWS CDK Workshop
- AWS CDK Examples
- AWS Developer Blog
- Gitter Channel
- Stack Overflow
- GitHub Repository
  - Issues
  - Examples
  - Documentation Source
  - License
  - Releases
    - AWS CDK OpenPGP Key (p. 64)
    - JSII OpenPGP Key (p. 65)
- AWS CDK Sample for Cloud9
- AWS CloudFormation Concepts
- AWS Glossary

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 queuing).

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 describes how to install and configure the AWS CDK and create your first AWS CDK app.

Prerequisites

AWS CDK command line tools

- Node.js (>= 8.11.x)
- You must specify both your credentials and an AWS Region to use the AWS CDK Toolkit;, as described in Specifying Your Credentials and Region (p. 7).

TypeScript

TypeScript >= 2.7

JavaScript

none

Java

- Maven 3.5.4 and Java 8
- Set the `JAVA_HOME` environment variable to the path to where you have installed the JDK on your machine

C#

.NET standard 2.0 compatible implementation:

- .NET Core >= 2.0
- .NET Framework >= 4.6.1
- Mono >= 5.4

Python

- Python >= 3.6

Installing the AWS CDK

Install the AWS CDK using the following command.

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

Run the following command to see the version number of the AWS CDK.

```
cdk --version
```

Updating Your Language Dependencies

If you get an error message that your language framework is out of date, use one of the following commands to update the components that the AWS CDK needs to support the language.

TypeScript

```
npx npm-check-updates -u
```

JavaScript

```
npx npm-check-updates -u
```

Java

```
mvn versions:use-latest-versions
```

C#

```
nuget update
```

Python

```
pip install --upgrade aws-cdk.cdk
```

You might have to call this multiple times to update all dependencies.

Using the env Property to Specify Account and Region

You can use the `env` property on a stack to specify the account and region used when deploying a stack, as shown in the following example, where *`REGION`* is the region and *`ACCOUNT`* is the account ID.

```
new MyStack(app, { env: { region: 'REGION', account: 'ACCOUNT' } });
```

Note
The AWS CDK team recommends that you explicitly set your account and region using the `env` property on a stack when you deploy stacks to production.

Since you can create any number of stacks in any of your accounts in any region that supports all of the stack's resources, the AWS CDK team recommends that you create your production stacks in one AWS CDK app, and deploy them as necessary. For example, if you own three accounts, with account IDs *`ONE`*, *`TWO`*, and *`THREE`* and want to be able to deploy each one in **us-west-2** and **us-east-1**, you might declare them as:

```
new MyStack(app, 'Stack-One-W', { env: { account: 'ONE', region: 'us-west-2' }});
new MyStack(app, 'Stack-One-E', { env: { account: 'ONE', region: 'us-east-1' }});
new MyStack(app, 'Stack-Two-W', { env: { account: 'TWO', region: 'us-west-2' }});
new MyStack(app, 'Stack-Two-E', { env: { account: 'TWO', region: 'us-east-1' }});
new MyStack(app, 'Stack-Three-W', { env: { account: 'THREE', region: 'us-west-2' }});
new MyStack(app, 'Stack-Three-E', { env: { account: 'THREE', region: 'us-east-1' }});
```

And deploy the stack for account *TWO* in **us-east-1** with:

```
cdk deploy Stack-Two-E
```

Note
If the existing credentials do not have permission to create resources within the account you specify, the AWS CDK returns an AWS CloudFormation error when you attempt to deploy the stack.

Specifying Your Credentials and Region

You must specify your credentials and an AWS Region to use the AWS CDK Toolkit. The CDK looks for credentials and region in the following order:

- Using the **--profile** option to **cdk** commands.
- Using environment variables.
- Using the default profile as set by the AWS Command Line Interface (AWS CLI).

Using the --profile Option to Specify Credentials and Region

Use the **--profile** ***PROFILE*** option to a **cdk** command to use a specific profile when executing the command.

For example, if the `~/.aws/config` (Linux or Mac) or `%USERPROFILE%\.aws\config` (Windows) file contains the following profile:

```
[profile test]
aws_access_key_id=AKIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
region=us-west-2
```

You can deploy your app using the **test** profile with the following command.

```
cdk deploy --profile test
```

Note
The profile must contain the access key, secret access key, and region.

See Named Profiles in the AWS CLI documentation for details.

Using Environment Variables to Specify Credentials and a Region

Use environment variables to specify your credentials and region.

- `AWS_ACCESS_KEY_ID` – Specifies your access key.
- `AWS_SECRET_ACCESS_KEY` – Specifies your secret access key.
- `AWS_DEFAULT_REGION` – Specifies your default Region.

For example, to set the region to **us-east-2** on Linux or macOS:

```
export AWS_DEFAULT_REGION=us-east-2
```

To do the same on Windows:

```
set AWS_DEFAULT_REGION=us-east-2
```

See Environment Variables in the *AWS Command Line Interface User Guide* for details.

Using the AWS CLI to Specify Credentials and a Region

Use the AWS Command Line Interface **aws configure** command to specify your default credentials and a region.

Hello World Tutorial

The typical workflow for creating a new app is:

1. Create the app directory.
2. Initialize the app.
3. Add code to the app.
4. Compile the app, if necessary.
5. To deploy the resources defined in the app.
6. Test the app.
7. If there are any issues, loop through modify, compile (if necessary), deploy, and test again.

And of course, keep your code under version control.

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

Creating the App Directory

Create a directory for your app with an empty Git repository.

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

Initializing the App

Initialize an app, where *LANGUAGE* is one of the supported programming languages: **csharp** (C#), **java** (Java), **python** (Python), or **typescript** (TypeScript) and *TEMPLATE* is an optional template that creates an app with different resources than the default app that **cdk init** creates for the language.

```
cdk init --language LANGUAGE [TEMPLATE]
```

The following table describes the templates provided by the supported languages.

Language	Template	Description
C#	*none*	Creates a CDK app with two stacks. Both stacks containing an SQS queue, an SNS topic, an IAM policy document so the topic could send messages to the queue, and a user-created construct. Each stack calls the user-created construct to create five S3 buckets, create a new user, and give the user access to the S3 buckets.
Java	*none*	Same as C#
Python	*none*	Creates an empty CDK app.
	sample-app	Same as C#, but four buckets
TypeScript	*none*	Creates an empty CDK app.
	sample-app	Creates a CDK app with stack containing an SQS queue, an SNS topic, and an IAM policy document so the topic could send messages to the queue.

TypeScript

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

JavaScript

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

Java

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

Once the **init** command finishes, we need to modify the template's output.

- Open `src/main/java/com/myorg/HelloApp.java`.
- Change the two stacks to one:

```
new HelloStack(app, "HelloCdkStack");
```

C#

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

Once the **init** command finishes, we need to modify the template's output.

- Open `src/HelloCdk/Program.cs`.
- Change the two stacks to one:

```
new HelloStack(app, "HelloCdkStack", new StackProps());
```

- Open `src/HelloCdk/HelloStack.cs`.
- Delete all of the using statements except the first.

```
using Amazon.CDK;
```

- Delete everything from the constructor.

Python

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

Once the **init** command finishes, your prompt should show **(.env)**, indicating you are running under virtualenv. If not, you must perform one or two more tasks, depending upon your operating system.

On Linux/MacOS:

```
python3 -m venv .env
source .env/bin/activate
```

On Windows:

```
.env\Scripts\activate.bat
```

Once you've got your virtualenv running, run the following command to install the required dependencies.

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

Change the instantiation of **HelloCdkStack** in `app.py` to the following.

```
HelloCdkStack(app, "HelloCdkStack")
```

Compiling the App

Compile your program, as follows.

TypeScript

```
npm run build
```

JavaScript

Nothing to compile.

Java

```
mvn compile
```

C#

```
dotnet build src
```

Python

Nothing to compile.

Note
If you are using Java and get annoyed by the **[INFO]** log messages, you can suppress them by including the **-q** option to your **mvn** commands. This includes changing `cdk.json to:`

```
"app": "mvn -q exec:java"
```

Listing the Stacks in the App

List the stacks in the app.

```
cdk ls
```

The result is just the name of the stack.

```
HelloCdkStack
```

Adding an Amazon S3 Bucket

At this point, what can you do with this app? Nothing, because the stack is empty, so there's nothing to deploy. Let's define an Amazon S3 bucket.

Install the `@aws-cdk/aws-s3` package.

TypeScript

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

JavaScript

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

Java

If necessary, add the following to `pom.xml`, where *`CDK-VERSION`* is the version of the AWS CDK.

```
<dependency>
  <groupId>software.amazon.awscdk</groupId>
  <artifactId>s3</artifactId>
  <version>CDK-VERSION</version>
</dependency>
```

C#

Run the following command in the src/HelloCdk directory.

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

Python

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

You might have to execute this command multiple times to resolve dependencies.

Next, define an Amazon S3 bucket in the stack. Amazon S3 buckets are represented by the Bucket class.

TypeScript

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

```
import cdk = require('@aws-cdk/cdk');
import s3 = require('@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 `index.js`:

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

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

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

Java

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

```
package com.myorg;

import software.amazon.awscdk.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketProps;

public class HelloStack extends Stack {
    public HelloStack(final Construct parent, final String id) {
        this(parent, id, null);
```

```
    }

    public HelloStack(final Construct parent, final String id, final StackProps props)
 {
        super(parent, id, props);

        new Bucket(this, "MyFirstBucket", BucketProps.builder()
            .withVersioned(true)
            .build());
    }
}
```

C#

Update `HelloStack.cs` to include a Amazon S3 bucket with versioning enabled.

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

namespace HelloCdk
{
    public class HelloStack : Stack
    {
        public HelloStack(Construct parent, string id, IStackProps props) :
 base(parent, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps
            {
                Versioned = true
            });
        }
    }
}
```

Python

Replace the import statement in `hello_cdk_stack.py` in the `hello_cdk` directory with the following code.

```
from aws_cdk import (
    aws_s3 as s3,
    cdk
)
```

Replace the comment with the following code.

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

Notice a few things:

- Bucket is a construct. This means it's initialization signature has `scope`, `id`, and `props` and it is a child of the stack.
- `MyFirstBucket` is the **id** of the bucket construct, not the physical name of the Amazon S3 bucket. The logical ID is used to uniquely identify resources in your stack across deployments. To specify a physical name for your bucket, set the bucketName property when you define your bucket.
- Because the bucket's versioned property is `true`, versioning is enabled on the bucket.

Compile your program, as follows.

TypeScript

```
npm run build
```

JavaScript

Nothing to compile.

Java

```
mvn compile
```

C#

```
dotnet build src
```

Python

Nothing to compile.

Synthesizing an AWS CloudFormation Template

Synthesize an AWS CloudFormation template for the app, as follows. If you get an error like "--app is required...", it's because you are running the command from within the `hello_cdk` sub-directory. Navigate to the parent directory and try again.

```
cdk synth
```

This command executes the AWS CDK app and synthesizes an AWS CloudFormation template for the `HelloCdkStack` stack. You should see something similar to the following, where *`VERSION`* is the version of the AWS CDK.

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    Metadata:
      aws:cdk:path: HelloCdkStack/MyFirstBucket/Resource
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Modules: "@aws-cdk/aws-codepipeline-api=VERSION,@aws-cdk/aws-events=VERSION,@aws-c\
        dk/aws-iam=VERSION,@aws-cdk/aws-kms=VERSION,@aws-cdk/aws-s3=VERSION,@aws-c\
        dk/aws-s3-notifications=VERSION,@aws-cdk/cdk=VERSION,@aws-cdk/cx-api=VERSION\
        .0,hello-cdk=0.1.0"
```

You can see that the stack contains an `AWS::S3::Bucket` resource with the versioning configuration we want.

Note

The toolkit automatically added the **AWS::CDK::Metadata** resource to your template. The AWS CDK uses metadata to gain insight into how the AWS CDK is used. One possible benefit is that the CDK team could notify users if a construct is going to be deprecated. For details, including how to opt out (p. 60) of version reporting, see Version Reporting (p. 60) .

Deploying the Stack

Deploy the app, as follows.

```
cdk deploy
```

The **deploy** command synthesizes an AWS CloudFormation template from the app, and then invokes the AWS CloudFormation create/update API to deploy it into your AWS account. If your code includes changes to your existing infrastructure, the command displays information about those changes and requires you to confirm them before it deploys the changes. The command displays information as it completes various steps in the process. There is no mechanism to detect or react to any specific step in the process.

Modifying the App

Configure the bucket to use AWS Key Management Service (AWS KMS) managed encryption.

TypeScript

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

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

JavaScript

Update `index.js`.

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

Java

Update `src/main/java/com/myorg/HelloStack.java`.

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

```
new Bucket(this, "MyFirstBucket", BucketProps.builder()
    .withVersioned(true)
    .withEncryption(BucketEncryption.KmsManaged)
    .build());
```

C#

Update `HelloStack.cs`.

```
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true,
```

```
    Encryption = BucketEncryption.KmsManaged
});
```

Python

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

Compile your program, as follows.

TypeScript

```
npm run build
```

JavaScript

Nothing to compile.

Java

```
mvn compile
```

C#

```
dotnet build src
```

Python

Nothing to compile.

Preparing for Deployment

Before you deploy the updated app, evaluate the difference between the AWS CDK app and the deployed app.

```
cdk diff
```

The toolkit queries your AWS account for the current AWS CloudFormation template for the `hello-cdk` stack, and compares the result with the template synthesized from the app. The Resources section of the output should look like the following.

```
Stack HelloCdkStack
Resources
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 |- [+] BucketEncryption
     |- {"ServerSideEncryptionConfiguration":[{"ServerSideEncryptionByDefault":
{"SSEAlgorithm":"aws:kms"}}]}
```

As you can see, the diff indicates that the `ServerSideEncryptionConfiguration` property of the bucket is now set to enable server-side encryption.

You can also see that the bucket isn't going to be replaced, but will be updated instead (**Updating MyFirstBucket...**).

Deploy the changes.

```
cdk deploy
```

Enter **y** to approve the changes and deploy the updated stack. The AWS CDK Toolkit updates the bucket configuration to enable server-side AWS KMS encryption for the bucket. The final output is the ARN of the stack, where *REGION* is your default region, *ACCOUNT-ID* is your account ID, and *ID* is a unique identifier for the bucket or stack.

```
HelloCdkStack: deploying...
HelloCdkStack: creating CloudFormation changeset...
 0/2 | 10:55:30 AM | UPDATE_IN_PROGRESS   | AWS::S3::Bucket    | MyFirstBucket
 (MyFirstBucketID)
 1/2 | 10:55:50 AM | UPDATE_COMPLETE      | AWS::S3::Bucket    | MyFirstBucket
 (MyFirstBucketID)

HelloCdkStack

Stack ARN:
arn:aws:cloudformation:REGION:ACCOUNT-ID:stack/HelloCdkStack/ID
```

Destroying the App's Resources

Destroy the app's resources to avoid incurring any costs from the resources created in this tutorial, as follows.

```
cdk destroy
```

Enter **y** to approve the changes and delete any stack resources. In some cases this command fails, such as when a resource isn't empty and must be empty before it can be destroyed. See Delete Stack Fails in the *AWS CloudFormation User Guide* for details.

Concepts

This topic describes some of the concepts (the why and how) behind the AWS CDK. It also discusses the advantages of using the AWS Construct Library instead of a low-level AWS CloudFormation Resource.

AWS CDK apps are composed of building blocks known as Constructs (p. 18), which are composed together to form stacks .

Constructs

You can think of constructs as *cloud components*. They can represent architectures of any complexity. They can represent a single resource, such as an Amazon Simple Storage Service (Amazon S3) bucket or an Amazon Simple Notification Service (Amazon SNS) topic. They can represent reusable components, such as a static website, a part of a specific application, or complex, multistack applications that span multiple accounts and AWS Regions. Constructs can also include other constructs. Everything in the AWS CDK is a construct.

This composition of constructs means that you can create sharable constructs. For example, if construct A and construct B use construct C and you make changes to construct C, then and both construct A and construct B get those changes.

The AWS CloudFormation Resource Library

The AWS CDK provides a class library of constructs called the **AWS CloudFormation Resource Library**. This library consists of constructs that represent all the resources available on AWS.

Each module in the AWS Construct Library includes two types of constructs for each resource: low-level constructs known as an AWS CloudFormation Resource constructs and high-level constructs known as an AWS Construct Library constructs.

The AWS CDK creates the low-level resources from the AWS CloudFormation Resource Specification on a regular basis. Low-level constructs are named **Cfn**`Xyz`, where `Xyz` represents the name of the resource. These constructs provide direct, one-to-one access to how a resource is synthesized in the AWS CloudFormation template produced by your AWS CDK app. Using low-level resources requires you to explicitly configure all resource properties, IAM policies, and have a deep understanding of the details.

High-level resource constructs are authored by AWS and offer an intent-based API for using AWS services. They provide the same functionality as the low-level resources, but encode much of the details, boilerplate, and glue logic required to use AWS. High-level resources offer convenient defaults and additional knowledge about the inner workings of the AWS resources they represent.

Similarly to the AWS SDKs and AWS CloudFormation, the AWS Construct Library is organized into modules, one for each AWS service. For example, the `@aws-cdk/aws-ec2` module includes resources for Amazon EC2 instances and networking. The `aws-sns` module includes resources such as `Topic` and `Subscription`. See the Reference for descriptions of the AWS CDK packages and constructs.

AWS Construct Library members are found in the `@aws-cdk/aws-NAMESPACE` packages, where `NAMESPACE` is the short name for the associated service, such as `SQS` for the AWS Construct Library for the Amazon Simple Queue Service (Amazon SQS) service.

Construct Structure

Constructs are represented as normal classes in your code and are defined by instantiating an object of that class.

When constructs are initialized, they are always defined within the *scope* of another construct, and always have an *id* that must be unique within the same scope.

For example, here's how you would define an Amazon SNS `topic` in your stack with default configuration.

```
new sns.Topic(this, 'MyTopic');
```

The first argument to every construct is always the scope in which it's created, and is almost always `this`, because most constructs are defined within the current scope.

Scopes enable constructs to be composed together to form higher-level abstractions. This is done by enabling the framework to group them together into logical units, allocate globally unique identifiers, and allow them to consult context information, such as the AWS Region in which it's going to be deployed and which availability Zones are available for your account.

In most cases, the construct initializer has a third `props` argument that can be used to define the construct's initial configuration. For example:

```
new MyConstruct(this, 'Foo', {
  favoriteColor: 'green',
  timeout: 300
});
```

Use the `construct.node` property to get the following information about the construct.

construct.node.scope

Gets the scope in which the construct was defined.

construct.node.id

Gets the `id` of the construct.

construct.node.uniqueId

Gets the app-wide unique, safe ID of the construct. This ID encodes the construct's path into a human-readable portion and a hash of the full path to ensure global uniqueness.

construct.node.path

Gets the full path of this construct from the root of the scope (the `App`).

Construct IDs

Every construct in a AWS CDK app must have an `id` that's unique within the scope in which the construct is defined. The AWS CDK uses IDs to find constructs in the construct hierarchy. It also uses IDs to allocate logical IDs so that AWS CloudFormation can keep track of the generated resources.

When a construct is created, its ID is specified as the second initializer argument.

```
const c1 = new MyConstruct(this, 'OneConstruct');
const c2 = new MyConstruct(this, 'TwoConstruct');
assert(c1.node.id === 'OneConstruct');
assert(c2.node.id === 'TwoConstruct');
```

Notice that the ID of a construct doesn't directly map to the physical name of the resource when it's created. To give a physical name to a bucket or table, specify the physical name using the appropriate property, such as `bucketName` or `tableName`, as shown in the following example.

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

We recommend that you avoid specifying physical names. Instead, let AWS CloudFormation generate names for you. Use attributes, such as `bucket.bucketName`, to discover the generated names.

When you synthesize a AWS CDK app into an AWS CloudFormation template, the AWS CloudFormation logical ID for each resource in the template is allocated according to the path of that resource in the scope hierarchy.

Construct Properties

Customize constructs by passing a property object as the third parameter (*props*). Every construct has its own set of properties, defined as an interface. You can pass a property object to your construct in two ways: inline, or instantiated as a separate property object.

```
// Inline (recommended)
new sqs.Queue(this, 'MyQueue', {
  visibilityTimeout: 300
});

// Instantiate separate property object
const props: QueueProps = {
  visibilityTimeout: 300
};

new Queue(this, 'MyQueue', props);
```

Construct Metadata

Attach metadata to a construct using the `addMetadata` method. Metadata is an AWS CDK-level annotation, and as such, does not appear in the deployed resources. Metadata entries automatically include the stack trace from which the metadata entry was added to allow tracing back to your code, even if the entry was defined by a lower-level library that you don't own.

Use the `addWarning()` method to emit a message when you you synthesis a stack; use the `addError()` method to not only emit a message when you you synthesis a stack, but to also block the deployment of a stack.

The following example blocks the deployment of `myStack` if it is not in `us-west-2`:

```
if (myStack.region !== 'us-west-2') {
    myStack.node.addError('myStack is not in us-west-2');
}
```

Tagging Constructs

You can add a tag to any construct to identify the resources you create. Tags can be applied to any construct. Tags are inherited, and are based on scope. If you tag construct `A`, and construct `A` contains construct `B`, construct `B` inherits the tag.

There are two tag operations.

Tag

    Adds (or applies) a tag to a set of resources, or to all but a set of resources.

RemoveTag

Removes a tag from a set of resources, or from all but a set of resources.

The following example adds the tag key-value pair *StackType-TheBest* to any resource created within the **theBestStack** stack labeled *MarketingSystem*.

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

const app = new cdk.App();
const theBestStack = new cdk.Stack(app, 'MarketingSystem');
theBestStack.node.applyAspect(new cdk.Tag('StackType', 'TheBest'));

// To remove the tag:
theBestStack.node.applyAspect(new cdk.RemoveTag('TheBest'));

// To remove the tag from all EXCEPT the subnets:
theBestStack.node.applyAspect(new cdk.RemoveTag('TheBest'), {exludeResourceTypes:
 ['AWS::EC2::Subnet']}));
```

The tag operations include some properties to fine-tune how tags are applied to or removed from the resources that the construct creates.

applyToLaunchedInstances

Use this Boolean property to set `PropagateAtLaunch` for any Auto Scaling group resource the construct creates. The default is `true`.

includeResourceTypes

Use this array of strings to apply a tag only to those AWS CloudFormation resource types. The default is an empty array, which means the tag applies to all AWS CloudFormation resource types.

excludeResourceTypes

Use this array of strings to exclude a tag from those AWS CloudFormation resource types. The default is an empty array, which means the tag applies to all AWS CloudFormation resource types. This property takes precedence over the `includeResourceTypes` property.

priority

Set this integer value to control the precedence of tags. The default is 0 (zero) for `Tag` and 1 for `RemoveTag`. Higher values take precedence over lower values.

Apps, Stacks, and Environments and Authentication

The main artifact of a AWS CDK program known as a *CDK app*. This is an executable program that you can use to synthesize deployment artifacts that supporting tools, such as the AWS CDK Toolkit, can deploy, as described in AWS CDK Command Line Interface (cdk) (p. 58).

Stacks are AWS CDK constructs that you can deploy into an AWS environment. The combination of AWS Region and account becomes the stack's *environment*. Most production apps consist of multiple stacks of resources that are deployed as a single transaction using a resource provisioning service such as AWS CloudFormation. Any resources added directly or indirectly as children of a stack are included in the stack's template when it is synthesized by your AWS CDK program.

Let's look at apps and stacks from the bottom up.

Stacks

Define an application stack by extending the Stack class, as shown in the following example.

```
import cdk = require("@aws-cdk/cdk");
import s3 = require("@aws-cdk/aws-s3");

interface MyStackProps extends cdk.StackProps {
  enc: boolean;
}

export class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props: MyStackProps) {
    super(scope, id, props);

    if (props.enc) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KmsManaged
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket");
    }
  }
}
```

Next we'll show you how to use one or more stacks to create a AWS CDK app.

Apps

An app is a collection of stack objects, as shown in the following example.

```
import cdk = require("@aws-cdk/cdk");
import { MyStack } from "../lib/MyApp-stack";

const app = new cdk.App();

new MyStack(app, "MyWestCdkStack", {
  env: {
    region: "us-west-2"
  },
  enc: false
});

new MyStack(app, "MyEastCdkStack", {
  env: {
    region: "us-east-1"
  },
  enc: true
});
```

Use the **cdk ls** command to list the stacks in this executable, as shown in the following example.

```
MyEastCdkStack
MyWestCdkStack
```

Use the **cdk deploy** command to deploy an individual stack.

```
cdk deploy MyWestCdkStack
```

Environments and Authentication

When you create a Stack instance, you can supply the target deployment environment for the stack using the `env` property, as described in Specifying Your Credentials and Region (p. 7).

We recommend that you use the default environment for development stacks, and explicitly specify accounts and regions using the `env` property for production stacks.

You can always find the Region within which your stack is deployed by using the `region` property of the stack, as follows.

```
this.region
```

Resources

The AWS CDK creates the low-level resources from the AWS CloudFormation Resource Specification on a regular basis. The classes are available under the `CfnXxx` classes of each AWS library. Their API matches 1:1 with how you would use these resources in AWS CloudFormation.

When defining AWS CloudFormation resources, the `props` argument of the class initializer matches 1:1 to the resource's properties in AWS CloudFormation.

For example, to define an AWS::SQS::Queue resource encrypted with an AWS managed key, you can directly specify the KmsMasterKeyId property.

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

new sqs.CfnQueue(this, 'MyQueueResource', {
    kmsMasterKeyId: 'alias/aws/sqs'
});
```

For reference, if you use the Queue construct, you can define managed queue encryption as follows.

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

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

Resource Object Properties

Use resource object properties to get a runtime attribute of an AWS CloudFormation resource.

The following example configures the dead letter queue of an AWS Lambda function to use the Amazon Resource Name (ARN) of an Amazon SQS queue resource.

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

const dlq = new sqs.CfnQueue(this, { name: 'DLQ' });

new lambda.CfnFunction(this, {
    deadLetterConfig: {
        targetArn: dlq.queueArn
    }
});
```

The cdk.CfnReference attribute represents the AWS CloudFormation resource's intrinsic reference (or *return value*). For example, *dlq.ref* also refers to the queue's ARN. When possible, use an explicitly named attribute instead of *ref*.

Resource Options

For resources, the CfnResource.options object includes AWS CloudFormation options, such as `condition`, `updatePolicy`, `createPolicy`, and `metadata`.

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.

Lets 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.

```
import { App, Construct, Stack, StackProps } from '@aws-cdk/cdk';
import s3 = require('@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');
```

Paths

As the constructs in an AWS CDK application form a hierarchy, we refer to the collection of ids from a given construct, then of its parent construct, then grandparent construct, and so on up to the root of the construct tree, which is an instance of the `App` class 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 programatically, as shown in the following example, which gets the path of `myConstruct`. Since ids must be unique within the scope they are created, their paths are always unique within a AWS CDK application.

```
const path: string = 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 unique identifier for each construct in an application. Since the AWS CDK already has paths that are globally unique, the AWS CDK generates these unique identifiers by concatenating the elements of the path, and adds an 8-digit hash. The hash is necessary, as otherwise two distinct paths, such as `A/B/C` and `A/BC` would result in the same identifier. The AWS CDK calls this concatenated path elements and hash the *unique ID* of the construct.

You can access the unique ID of any construct programatically, as shown in the following example, which gets the unique ID of `myConstruct`. Since ids must be unique within the scope they are created, their paths are always unique within a AWS CDK application.

```
const uid: string = myConstruct.node.uniqueId;
```

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.

Run-Time Context

The AWS CDK uses context to retrieve information such as the Availability Zones in your account or Amazon Machine Image (AMI) IDs used to start your instances. To avoid unexpected changes to your deployments, such as when a new Amazon Linux AMI is released, thus changing your Auto Scaling group, the AWS CDK stores context values in `cdk.context.json`. This ensures that the AWS CDK retrieves the same value the next time it synthesises your app. Don't forget to put this file under version control.

Viewing and Managing Context

Use the **cdk context** to see context values stored for your application. The output should be something like the following.

```
Context found in cdk.json:

####################################################################################################
# # # Key                                                # Value
                   #
####################################################################################################
# 1 # availability-zones:account=123456789012:region=us- # [ "us-east-1a", "us-east-1b",
 "us-east-1c",        #
#   # east-1                                             # "us-east-1d", "us-east-1e", "us-
east-1f" ]         #
####################################################################################################
# 2 # ssm:account=123456789012:parameterName=/aws/       # "ami-013be31976ca2c322"
                   #
#   # service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_ #
                   #
#   # 64-gp2:region=us-east-1                            #
                   #
####################################################################################################

Run cdk context --reset KEY_OR_NUMBER to remove a context key. It will be refreshed on the
 next CDK synthesis run.
```

If at some point you want to update to the latest version of the Amazon Linux AMI, do a controlled update of the context value, reset it, and synthesize again.

```
$ cdk context --reset 2
```

```
Context value
ssm:account=123456789012:parameterName=/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-
x86_64-gp2:region=us-east-1
reset. It will be refreshed on the next SDK synthesis run.
```

```
cdk synth
```

```
...
```

To clear all context values, run **cdk context --clear**.

```
cdk context --clear
```

Context Providers

The AWS CDK currently supports the following context providers.

AvailabilityZoneProvider

Use this provider to get the list of all supported Availability Zones in this context, as shown in the following example.

```
// "this" refers to a Construct scope
const zones: string[] = new AvailabilityZoneProvider(this).availabilityZones;

for (let zone of zones) {
  // Do something for each zone!
}
```

HostedZoneProvider

Use this provider to discover existing hosted zones in your account. For example, the following code imports an existing hosted zone into your AWS CDK app so you can add records to it.

```
const zone: HostedZoneRef = new HostedZoneProvider(this, {
  domainName: 'test.com'
}).findAndImport(this, 'HostedZone');
```

SSMParameterProvider

Use this provider to read values from the current Region's AWS Systems Manager parameter store. For example, the following code returns the value of the *my-awesome-parameter* key.

```
const ami: string = new SSMParameterProvider(this, {
  parameterName: 'my-awesome-parameter'
).parameterValue();
```

This is only for reading plain strings, and not recommended for secrets. For reading secure strings from the Systems Manager Parameter Store, see Get a Value from a Systems Manager Parameter Store Variable (p. 51).

VpcNetworkProvider

Use this provider to look up and reference existing VPCs in your accounts. For example, the following code imports a VPC by tag name.

```
const provider = new VpcNetworkProvider(this, {
  tags: {
    "tag:Purpose": 'WebServices'
  }
});

const vpc = Vpc.import(this, 'VPC', provider.vpcProps);
```

Assets

Assets are local files, directories, or Docker images that can be bundled into AWS CDK constructs and apps. A common example is a directory that contains the handler code for an AWS Lambda function, however, assets can represent any artifact that the app needs to operate.

When deploying a AWS CDK app that includes constructs with assets, the AWS CDK Toolkit first prepares and publishes them to Amazon Simple Storage Service (Amazon S3) or Amazon Elastic Container Registry (Amazon ECR), and only then deploys the stacks. The locations of the published assets are passed in as AWS CloudFormation parameters to the relevant stacks.

AWS CloudFormation

The AWS Construct Library (p. 35) includes constructs with APIs for defining AWS infrastructure. For example, you can use the Bucket construct to define Amazon Simple Storage Service (Amazon S3) buckets, and the Topic construct to define Amazon Simple Notification Service (Amazon SNS) topics.

Under the hood, these constructs are implemented using AWS CloudFormation resources, which are available in the `CfnXxx` classes in each library. For example, the Bucket construct uses the CfnBucket resource (as well as other resources, depending on what bucket APIs are used).

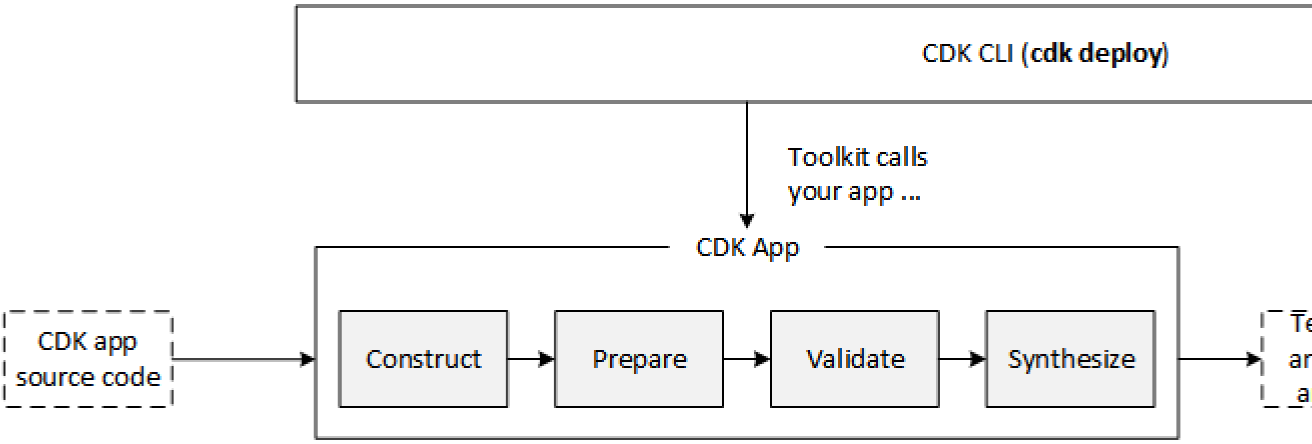
Important
Avoid interacting directly with AWS CloudFormation unless absolutely necessary, such as when a AWS CDK AWS Construct Library does not provide the needed functionality.

Tokens

Tokens represent instrinsic AWS CloudFormation values or values that not are known until deployment.

AWS CDK Lifecycle

The following illustration shows the phases that the AWS CDK goes through when you call **cdk deploy** to create the resources defined by your application.



There are three actors at play to create the resources that your AWS CDK application defines.

- Your AWS CDK app.
- The AWS CDK Toolkit.
- AWS CloudFormation, which the AWS CDK Toolkit calls to deploy your application and create the resources.

After you use the toolkit to deploy a AWS CDK application, the application goes through the following phases.

Construction

Your code instantiates all desired application constructs and links them together.

Preparation

All constructs that have implemented the `prepare` method participate in a final round of modifications, to set up any final state they want to. The preparation phase happens automatically and users do not see any feedback from this phase.

Validation

All constructs that have implemented their `validate` method can validate themselves to make sure they've ended up in a state that will correctly deploy. Users see any validation failures that are detected during this phase.

Synthesis

All constructs render themselves to a set of artifacts, representing AWS CloudFormation templates, AWS Lambda application bundles, and other deployment artifacts. Users do not see any feedback from this phase.

Deployment

The toolkit takes the artifacts produced by the synthesis step, uploads them to Amazon S3 or wherever they need to go, and starts an AWS CloudFormation deployment to deploy the application and create the resources.

Note that your AWS CDK app has already finished and exited by the time the AWS CloudFormation deployment starts. This has the following implications.

- Your AWS CDK app cannot 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.
- Your CDK app might have to work with values that cannot be known at the time it executes. For example, if your AWS CDK application defines an Amazon S3 Bucket with an automatically generated name, and you retrieve the `bucket.bucketName` attribute, that value is not the name of the deployed bucket. Instead, the value of the `bucketName` attribute is a symbolic value, looking like *${TOKEN[Bucket.Name.1234]}*. You can pass this value to constructs, or append it to other strings, and the AWS CDK framework will translate that value. You cannot examine the value and make decisions based on the deployed bucket name, because the bucket name not available until AWS CloudFormation is done deploying, and by that time your program is no longer running. Call `cdk.unresolved(value)`, which returns `true` if `value` not known until deployment time.

Writing AWS CDK Constructs

This topic provides some tips for writing idiomatic new constructs for the AWS CDK. These tips apply equally to constructs written for inclusion in the AWS Construct Library, purpose-built constructs to achieve a well-defined goal, or constructs that serve as building blocks for assembling your cloud applications.

General Design Principles

- Favor composition over inheritance. Most of the constructs should directly extend the `Construct` class instead of some other construct. Use inheritance mainly to allow polymorphism. Typically, you define a construct within your scope and expose any of its APIs and properties in the enclosing construct.
- Provide defaults for everything that a reasonable guess can be made for. Ideally, `props` should be optional and `new MyAwesomeConstruct(this, "Foo")` should be enough to set up a reasonable variant of the construct. This doesn't mean that the user should not have the opportunity to customize! Instead, it means that the specific parameter should be optional and set to a reasonable value if it's not supplied. This might involve creating other resources as part of initializing this construct. For example, all resources that require a role allow passing in a `Role` object (specifically, a `RoleRef` object), but if the user doesn't supply one, an appropriate `Role` object is defined in place.
- Use contextual defaulting between properties. The value of one property might affect sensible defaults for other properties. For example: `enableDnsHostnames` and `enableDnsSupport`. `dnsHostnames` require `dnsSupport`, so only throw an error if the user has explicitly disabled DNS Support, but tried to enable DNS ostnames. A user expects things to just work.
- Make the user think about intent, not implementation detail. For example, if establishing an association between two resources (such as a `Topic` and a `Queue`) requires multiple steps (in this case, creating a `Subscription` but also setting appropriate IAM permissions), make both things happen in a single call (to `subscribeQueue()`).
- Don't rename concepts or terminology. For example don't rename the Amazon SQS `dataKeyReusePeriod` to `keyRotation` because it will be hard for people to diagnose problems. They won't be able to search for *sqs dataKeyReuse* and find topics on it. It's permissible to introduce a concept if a counterpart doesn't exist in AWS CloudFormation, especially if it directly maps onto the mental model that users already have about a service.
- Optimize for the common case. For example, `AutoScalingGroup` accepts a `VPC` and deploys in the private subnet by default because that's the common case, but has an option to `placementOptions` for special cases.
- If a class can have multiple modes or behaviors, prefer values over polymorphism. Try switching behavior on property values first. Switch to multiple classes with a shared base class or interface only if there's value to be had from having multiple classes (type safety, maybe one mode has different featuresor required parameters).

Implementation Details

- Every construct consists of an exported class (`MyConstruct`) and an exported interface (`MyConstructProps`) that defines the parameters for these classes. The `props` argument is the third to the construct (after the mandatory `scope` and `id` arguments), and the entire parameter should be optional if all of the properties on the `props` object are optional.

- Most of the logic happens in the constructor. The constructor builds up the state of the construct (what children it has, which ones are always there and which are optional, and so on).
- Validate as early as possible. Throw an `Error` in the constructor if the parameters don't make sense. Override the `validate()` method to validate mutations that can occur after construction time. Validation has the following hierarchy:
  - Best – Incorrect code won't compile, because of type safety guarantees.
  - Good – Runtime check everything the type checker can't enforce and fail early (error in the constructor).
  - Okay – Validate everything that can't be checked at construction time at synth time (error in `validate()`).
  - Not great – Fail with an error in AWS CloudFormation (bad because the AWS CloudFormation deploy operation can take a long time, and the error can take several minutes to surface).
  - Very bad – Fail with a timeout during AWS CloudFormation deployment. (It can take up to an hour for resource stabilization to timeout!)
  - Worst – Don't fail the deployment at all, but fail at runtime.
- Avoid unneeded hierarchy in `props`. Try to keep the `props` interface flat to help discoverability.
- Constructs are classes that have a set of invariants they maintain over their lifetime (such as which members are initialized, and relationships between properties as members that are mutated).
- Constructs mostly have write-only scalar properties that are passed in the constructor, but mutating functions for collections (for example, there will be `construct.addElement()` or `construct.onEvent()` functions, but not `construct.setProperty()`). It's perfectly fine to deviate from this convention when it makes sense for your own constructs.
- Don't expose `Tokens` to your consumers. Tokens are an implementation mechanism for one of two purposes: representing AWS CloudFormation intrinsic values, or representing lazily evaluated values. You can use them for implementation purposes, but use more specific types as part of your public API.
- `Tokens` are (mostly) used only in the implementation of an AWS Construct Library to pass lazy values to other constructs. For example, if you have an array that can be mutated during the lifetime of your class, you pass it to an AWS CloudFormation Resourceconstruct like so: `new cdk.Token(() => this.myList)`.
- Be aware that you might not be able to usefully inspect all strings. Any string passed into your construct may contain special markers that represent values that will only be known at deploy time (for example, the ARN of a resource that will be created during deployment). Those are *stringified Tokens* and they look like `"${TOKEN[Token.123]}"`. You will not be able to validate those against a regular expression, for example, as their real values are not known yet. To determine whether your string contains a special marker, use `cdk.unresolved(string)`.
- Indicate units of measurement in property names that don't use a strong type. For example, use **milli**, **sec**, **min**, **hr**, **Bytes**, **KiB**, **MiB**, and **GiB**.
- Be sure to define an `IMyResource` interface for your resources that defines the API area that other constructs will use. Typical capabilities on this interface are querying for a resource ARN and adding resource permissions.
- Accept objects instead of ARNs or names. When accepting other resources as parameters, declare your property as `resource: IMyResource` instead of `resourceArn: string`. This makes snapping objects together feel natural to consumers, and allows you to query or modify the incoming resource as well. For example, the latter is particularly useful if something about IAM permissions needs to be set.
- If your construct wraps a single (or most prominent) other construct, give it an ID of either **"Resource"** or **"Default"**. The main resource that an AWS construct represents should use the ID **"Resource"** For higher-level wrapping resources, you will generally use **"Default"** (resources named **"Default"** will inherit their scope's logical ID, while resources named **"Resource"** will have a distinct logical ID, but the human-readable part of it won't show the **"Resource"** part).

Implementation Language

For construct libraries to be reusable across programming languages, they need to be authored in a language that can compile to a `jsii` assembly.

Author in TypeScript unless you plan to isolate your constructs to a single programming language.

Code Organization

Your package should look like the following.

```
your-package
### package.json
### README.md
### lib
#   ### index.ts
#   ### some-resource.ts
#   ### some-other-resource.ts
### test
      ### integ.everything.lit.ts
   ### test.some-resource.ts
      ### test.some-other-resource.ts
```

- If it represents the canonical AWS Construct Library for this service, name your package is named `@aws-cdk/aws-xxx`. We recommend starting with `cdk-`, but you are otherwise free to choose the name.
- Put code under `lib/`, and tests under `test/`.
- The entry point should be `lib/index.ts` and should only contain `exports` for other files.
- You don't need to put every class in a separate file. Try to think of a reader-friendly organization of your source files.
- To make package-private utility functions, put them in a file that isn't exported from `index.ts`.
- Free-floating functions (functions that are not part of a class definition) cannot be accessed through **jsii** (that is, from languages other than TypeScript and JavaScript). Don't use them for public features of your construct library.
- Document all public APIs with doc comments (`JSdoc` syntax). Document defaults using the **@default** marker in doc comments.

Testing

- Add unit tests for every construct (`test.xxx.ts`), relating the construct's properties to the AWS CloudFormation that is generated. Use the `@aws-cdk/assert` library to make it easier to write assertions on the AWS CloudFormation output.
- Try to test one concern per unit test. Even if you could test more than one feature of the construct per test, it's better to write multiple tests, one for each feature. A test should have one reason to break.
- Add integration tests (`integ.xxx.ts`) that are AWS CDK apps that exercise the features of the construct, then load your shell with credentials and run **npm run integ** to exercise them. You will also have to run this if the AWS CloudFormation output of the construct changes.
- If there are packages that you depend on only for testing, add them to `devDependencies` (instead of regular `dependencies`). You're still not allowed to create dependency cycles this way (from the root, run **scripts/find-cycles.sh** to determine whether you have created any cycles).
- If possible, try to make your integ test literate (`integ.xxx.lit.ts`) and link to it from the `README`.

README

- Header should include maturity level.
- Header should start at `H2`, not `H1`.
- Include some example code for the simple use case near the very top.
- If there are multiple common use cases, provide an example for each one and describe what happens under the hood at a high level (for example, which resources are created).
- Reference docs are not needed.
- Use literate (`.lit.ts`) integration tests in the `README` file.

Construct IDs

All child construct IDs are part of your public contract. Those IDs are used to generate AWS CloudFormation logical names for resources. If they change, AWS CloudFormation will replace the resource. Technically, this means that if you change any ID of a child construct you will have to major-version-bump your library.

Multi-Language Support in the AWS CDK

This section describes the multi-language support in the AWS CDK, including hints for porting TypeScript to one of the supported languages. See Hello World Tutorial (p. 8) for an example of creating a AWS CDK app in a supported language.

The AWS CDK supports C#, Java, JavaScript, and TypeScript. Since the AWS CDK is developed in TypeScript, many code examples are still only available in TypeScript. This section will help you port those TypeScript examples to one of the other programming languages.

Importing a Package

In TypeScript, you import a package as follows (we'll use Amazon S3 for our examples):

```
import s3 = require("@aws-cdk/aws-s3");
```

C#

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

Java

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

JavaScript

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

Python

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

Creating a New Object

In TypeScript, you create a new object as follows. The first argument, `scope`, is always `this`, the second is the `id` of the construct, and the last is a list of properties, often optional.

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

C#

```
new Bucket(this, "MyFirstBucket", new BucketProps
{
  // options
});
```

Java

```
new Bucket(this, "MyFirstBucket", BucketProps.builder()
  // options
}
```

JavaScript

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

Python

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

AWS Construct Library

The AWS Construct Library is a set of modules that expose APIs for defining AWS resources in AWS CDK apps. Each module is based on the AWS service to which the resource belongs. For example, EC2 includes the Vpc construct, which makes it easy to define an Amazon VPC in your AWS CDK app.

The AWS Construct Library modules are described in the AWS CDK Reference.

Versioning

The AWS CDK follows the semantic versioning model. This means that breaking changes are limited to major releases, such as 2.0.

Minor releases, such as 2.4, guarantee that any code written in a previous minor version, such as 2.1, will build, run, and produce the exact same results when built, run, and producing results, as before.

AWS CDK Patterns

The AWS Construct Library includes many common patterns and capabilities that are designed to enable developers to focus on their application-specific architectures and reduce the boilerplate and glue logic needed when working with AWS services.

Grants

AWS Identity and Access Management (IAM) policies are automatically defined based on intent. For example, when subscribing an Amazon Simple Notification Service (Amazon SNS) Topic to an AWS Lambda Function, the function's IAM permission policy is automatically modified to allow the specific topic to invoke the function.

Also, most AWS constructs expose `grant*` methods that allow intent-based permission definitions. For example, the Amazon S3 Bucket construct has a grantRead method. This method accepts an IAM IPrincipal such as a User or a Role, which modifies the policy to allow the principal to read objects from the bucket.

Metrics

Many AWS resources emit Amazon CloudWatch metrics as part of their normal operation. Metrics can be used to set up an Alarm or can be included in a Dashboard.

You can obtain Metric objects for AWS constructs by using `metricXxx()` methods. For example, the metricAllDuration method reports the execution time of an AWS Lambda function.

For more information, see CloudWatch.

Events

Many AWS constructs include `on*` methods, which you can to react to events emitted by the construct. For example, the CodeCommit Repository construct implements the onCommit method. You can use

AWS constructs as targets for various event provider interfaces, such as IEventRuleTarget (for the CloudWatch event rule target), IAlarmAction (for Amazon CloudWatch alarm actions), and so on.

For more information, see CloudWatch and Events.

Referencing Resources

This section contains information about how to reference other resources, either from within the same app, or across apps. The only caveat is that the resource must be in the same account and region.

Many CDK classes have required properties that are 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. This 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; a CloudFront distribution requires a reference to the bucket containing source code.

In the AWS CDK, all AWS Construct Library resources that expect another resource take a property that is of the interface type of that resource. For example, the Amazon ECS service takes a property of type `cluster: ICluster`; the CloudFront distribution takes a property of type `sourceBucket: IBucket`.

Since every resource implements its corresponding interface, you can directly pass any resources you're defining in the same AWS CDK application, as shown in the following example, which creates a new Amazon ECS cluster.

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

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

Passing Resources from a Different Stack

You can refer to resources in a different stack, but in the same account and region. If you need to refer to a resource in a different account or region, see the next section.

```
const account = '123456789012';
const region = 'us-east-1';

const stack1 = new StackThatProvidesABucket(app, 'Stack1');

// stack2 takes a property of type IBucket
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket
});
```

If the resource is in the same account and region, but in a different stack, the AWS CDK creates the relevant information, such as the bucket name, that is necessary to transfer that information from one stack to the other.

Passing Resources from a Different Account or Region

You can refer to a resource in a different account or region by using the resource's unique indentifier, such as:

- `bucketName` for `bucket`
- `functionArn` for `lambda`
- `securityGroupId` for `securityGroup`

The following example shows how to pass a generated bucket name to a Lambda function.

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

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

Turning Unique Identifiers into Objects

If you have the unique identifier for a resource, such as a bucket ARN, but you need to pass it to a AWS CDK construct that expects an object, you can turn the ARN into a AWS CDK object in the current stack by calling a static factory method on the resource's class. The following example shows how to create a bucket from an existing bucket ARN.

```
// Construct a resource (bucket) by its full ARN (can be cross account)
Bucket.fromBucketArn(this, 'Bucket', 'arn:aws:s3:::my-bucket-name');
```

About the Reference

See the AWS CDK Reference for 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.

Examples

This topic contains the following examples:

- Creating a Serverless Application Using the AWS CDK (p. 39) Creates a serverless application using Lambda, API Gateway, and Amazon S3.
- Creating an AWS Fargate Service Using the AWS CDK (p. 45) Creates an Amazon ECS Fargate service from an image on DockerHub.

Creating a Serverless Application Using the AWS CDK

This example walks you through how to create the resources for a simple widget dispensing service. It includes:

- An AWS Lambda function.
- An Amazon API Gateway API to call the Lambda function.
- An Amazon S3 bucket that contains the Lambda function code.

This tutorial contains the following steps.

1. Creates a AWS CDK app
2. Creates a Lambda function that gets a list of widgets with: GET /
3. Creates the service that calls the Lambda function
4. Adds the service to the AWS CDK app
5. Tests 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}**

Create a AWS CDK App

Create the TypeScript app **MyWidgetService** in the current folder.

```
mkdir MyWidgetService
cd MyWidgetService
cdk init --language typescript
```

This creates `my_widget_service.ts` in the `bin` directory, and `my_widget_service-stack.ts` in the `lib` directory.

Build the app and notice that it creates an empty stack.

```
npm run build
cdk synth
```

You should see a stack like the following, where *`CDK-VERSION`* is the version of the AWS CDK.

```
Resources:
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Modules: "@aws-cdk/cdk=CDK-VERSION,@aws-cdk/cx-api=CDK-
VERSION,my_widget_service=0.1.0"
```

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.

Create the `resources` directory at the same level as the `bin` directory.

```
mkdir resources
```

Create the following JavaScript file, `widgets.js`, in the `resources` directory.

```
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 it builds and creates an empty stack. Because we haven't wired the function to the app, the Lambda file doesn't appear in the output.

```
npm run build
```

```
cdk synth
```

Creating a Widget Service

Add the API Gateway, Lambda, and Amazon S3 packages to the app.

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

Create the TypeScript file `widget_service.ts` in the `lib` directory.

```
import cdk = require("@aws-cdk/cdk");
import apigateway = require("@aws-cdk/aws-apigateway");
import lambda = require("@aws-cdk/aws-lambda");
import s3 = require("@aws-cdk/aws-s3");

export class WidgetService extends cdk.Construct {
  constructor(scope: cdk.Construct, id: string) {
    super(scope, id);

    const bucket = new s3.Bucket(this, "WidgetStore");

    const handler = new lambda.Function(this, "WidgetHandler", {
      runtime: lambda.Runtime.NodeJS810, // So we can use async in widget.js
      code: lambda.Code.directory("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 /

    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}
  }
}
```

Save the app and be sure it builds and creates a (still empty) stack.

```
npm run build
cdk synth
```

Add the Service to the App

To add the service to the app, first modify `my_widget_service-stack.ts`. Add the following line of code after the existing `import` statement.

```
import widget_service = require('../lib/widget_service');
```

Replace the comment in the constructor with the following line of code.

```
    new widget_service.WidgetService(this, 'Widgets');
```

Be sure the app builds and creates a stack (we don't show the stack because it's over 250 lines).

```
npm run build
cdk synth
```

Deploy and Test the App

Before you can deploy your first AWS CDK app, you must bootstrap your deployment. This creates some AWS infrastructure that the AWS CDK needs. For details, see the **bootstrap** section of the AWS CDK Toolchain (p. 58)(if you've already bootstrapped a AWS CDK app, you'll get a warning and nothing will change).

```
cdk bootstrap
```

Deploy your 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-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 existing `exports.main` function in `widgets.js` with the following code.

```
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 in `widget_service.ts` by adding the following code at the end of the constructor.

```
const widget = api.root.addResource('{name}');

// Add new widget to bucket with: POST /{name}
const postWidgetIntegration = new apigateway.LambdaIntegration(handler);

// Get a specific widget from bucket with: GET /{name}
const getWidgetIntegration = new apigateway.LambdaIntegration(handler);

// Remove a specific widget from the bucket with: DELETE /{name}
const deleteWidgetIntegration = new apigateway.LambdaIntegration(handler);

widget.addMethod('POST', postWidgetIntegration);    // POST /{name}
widget.addMethod('GET', getWidgetIntegration);       // GET /{name}
widget.addMethod('DELETE', deleteWidgetIntegration); // DELETE /{name}
```

Save, build, and deploy the app.

```
npm run build
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. You have to set the **name** value to the name of a widget, such as **example**.

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.

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language typescript
```

Build the app and confirm that it creates an empty stack.

```
npm run build
cdk synth
```

You should see a stack like the following, where *CDK-VERSION* is the version of the CDK.

```
Resources:
  CDKMetadata:
    Type: 'AWS::CDK::Metadata'
    Properties:
      Modules: @aws-cdk/cdk=CDK-VERSION,@aws-cdk/cx-api=CDK-VERSION,my_ecs_construct=0.1.0
```

Add the Amazon EC2 and Amazon ECS Packages

Install support for Amazon EC2 and Amazon ECS.

```
npm install @aws-cdk/aws-ec2 @aws-cdk/aws-ecs
```

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.

The following tutorial creates a Fargate service running on an ECS cluster fronted by an internet-facing Application ad Balancer.

Add the following `import` statements to `lib/my_ecs_construct-stack.ts`.

```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs');
import ecs_patterns = require('@aws-cdk/aws-ecs-patterns');
```

Replace the comment at the end of the constructor with the following code.

```
    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.LoadBalancedFargateService(this, 'MyFargateService', {
      cluster: cluster,  // Required
      cpu: '512', // Default is 256
      desiredCount: 6,  // Default is 1
      image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-sample"), // Required
      memoryMiB: '2048',  // Default is 512
      publicLoadBalancer: true  // Default is false
    });
```

Save it and make sure it builds and creates a stack.

```
npm run build
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.

AWS CDK Examples

The CDK Examples repo on GitHub includes the following examples.

TypeScript examples

Example	Description
api-cors-lambda-crud-dynamodb	Creating a single API with CORS, and five Lambdas doing CRUD operations over a single DynamoDB

Example	Description
application-load-balancer	Using an AutoScalingGroup with an Application Load Balancer
appsync-graphql-dynamodb	Creating a single GraphQL API with an API Key, and four Resolvers doing CRUD operations over a single DynamoDB
classic-load-balancer	Using an AutoScalingGroup with a Classic Load Balancer
custom-resource	Shows adding a Custom Resource to your CDK app
elasticbeanstalk	Elastic Beanstalk example using L1 with a Blue/Green pipeline (community contributed)
ecs-cluster	Provision an ECS Cluster with custom Autoscaling Group configuration
ecs-load-balanced-service	Starting a container fronted by a load balancer on ECS
ecs-service-with-task-placement	Starting a container ECS with task placement specifications
ecs-service-with-advanced-alb-config	Starting a container fronted by a load balancer on ECS with added load balancer configuration
ecs-service-with-task-networking	Starting an ECS service with task networking, allowing ingress traffic to the task but blocking for the instance
fargate-load-balanced-service	Starting a container fronted by a load balancer on Fargate
fargate-service-with-auto-scaling	Starting an ECS service of FARGATE launch type that auto scales based on average CPU Utilization
lambda-cron	Running a Lambda on a schedule
my-widget-service	Use Lambda to serve up widgets
resource-overrides	Shows how to override generated CloudFormation code
static-site	A static site using CloudFront
stepfunctions-job-poller	A simple StepFunctions workflow
ecs-service-with-logging	Starting a container fronted by a load balancer on Fargate
fargate-service-with-logging	Starting a container fronted by a load balancer on Fargate

Java examples

Example	Description
hello-world	A demo application that uses the CDK in Java
lambda-cron	Running a Lambda on a schedule

Python examples

Example	Description
application-load-balancer	Using an AutoScalingGroup with an Application Load Balancer
classic-load-balancer	Using an AutoScalingGroup with a Classic Load Balancer
custom-resource	Shows adding a Custom Resource to your CDK app
ecs-cluster	Provision an ECS Cluster with custom Autoscaling Group configuration
ecs-load-balanced-service	Starting a container fronted by a load balancer on ECS
ecs-service-with-task-placement	Starting a container ECS with task placement specifications
ecs-service-with-advanced-alb-config	Starting a container fronted by a load balancer on ECS with added load balancer configuration
ecs-service-with-task-networking	Starting an ECS service with task networking, allowing ingress traffic to the task but blocking for the instance
fargate-load-balanced-service	Starting a container fronted by a load balancer on Fargate
fargate-service-with-autoscaling	Starting an ECS service of FARGATE launch type that auto scales based on average CPU Utilization
lambda-cron	Running a Lambda on a schedule
stepfunctions	A simple StepFunctions workflow

JavaScript examples

Example	Description
aws-cdk-changelogs-demo	A full serverless Node.js application stack deployed using CDK. It uses AWS Lambda, AWS Fargate, DynamoDB, Elasticache, S3, and CloudFront.

AWS CDK HowTos

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`.

```
const bucket_name = process.env.MYBUCKET;
```

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 next section.

Use an Existing AWS CloudFormation Template

The AWS CDK provides a mechanism that you can use to incorporate resources from an existing AWS CloudFormation template into your AWS CDK app. For example, suppose you have a template, `my-template.json`, with the following resource, where *`S3Bucket`* is the logical ID of the bucket in your template:

```
{
  "S3Bucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "prop1": "value1"
    }
  }
}
```

You can include this bucket in your AWS CDK app, as shown in the following example (note that you cannot use this method in an AWS Construct Library construct):

```
import cdk = require("@amp;aws-cdk/cdk");
import fs = require("fs");

new cdk.Include(this, "ExistingInfrastructure", {
  template: JSON.parse(fs.readFileSync("my-template.json").toString())
});
```

Then to access an attribute of the resource, such as the bucket's ARN:

```
const bucketArn = cdk.Fn.getAtt("S3Bucket", "Arn");
```

Get a Value from a Systems Manager Parameter Store Variable

You can get the value of an AWS Systems Manager Parameter Store variable, depending on whether you want the latest version of a plain string, a particular version of a plain string, or a particular version of a secret string. It isn't possible to retrieve the latest version of a secure string. To read the latest version of a secret, you have to read the secret from AWS Secrets Manager (see Get a Value from AWS Secrets Manager (p. 51)).

To read a particular version of a Systems Manager Parameter Store plain string value at AWS CloudFormation deployment time, use ParameterStoreString. If you don't supply a `version` value, you get the latest version.

```
import ssm = require('@aws-cdk/aws-ssm');

const parameterString = new ssm.ParameterStoreString(this, 'MyParameter', {
    parameterName: 'my-parameter-name',
    version: 1,
});

const myvalue = parameterString.stringValue;
```

To read a particular version of a Systems Manager Parameter Store `SecureString` value at AWS CloudFormation deployment time, use ParameterStoreSecureString. You must supply a `version` value.

```
import ssm = require('@aws-cdk/aws-ssm');

const secureString = new ssm.ParameterStoreSecureString(this, 'MySecretParameter', {
    parameterName: 'my-secret-parameter-name',
    version: 1,
});

const myvalue = secureString.stringValue;
```

Use the put-parameter CLI command to add a string parameter to the system, such as when testing:

```
aws ssm put-parameter --name "my-parameter-name" --type "String" --value "my-parameter-value"
```

The command returns an ARN you can use for the example.

Get a Value from AWS Secrets Manager

To use values from AWS Secrets Manager in your CDK app, use the fromSecretAttributes method. It represents a value that is retrieved from Secrets Manager and used at AWS CloudFormation deployment time.

```
import sm = require("@aws-cdk/aws-secretsmanager");
```

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

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretArn:
        "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:
      // Key,
    });
```

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.

Work Around Missing AWS CDK Features

This topic describes how to modify the underlying AWS CloudFormation resources in the AWS Construct Library. We also call this technique an "escape hatch" because it allows users to "escape" from the abstraction boundary defined by the AWS construct, and patch the underlying resources.

Important
We don't recommend this method because it breaks the abstraction layer and might produce unexpected results.
If you modify an AWS construct in this way, we can't ensure that your code will be compatible with subsequent releases.

AWS constructs, such as Topic, encapsulate one or more AWS CloudFormation resources behind their APIs. These resources are also represented as `CfnXxx` constructs in each library. For example, the Bucket construct encapsulates the CfnBucket. When a stack that includes an AWS construct is synthesized, the AWS CloudFormation definitions of the underlying resources are included in the resulting template.

Eventually, we expect the APIs provided by AWS constructs to support all of the services and capabilities offered by AWS. But we're aware that the library still has many gaps, both at the service level (some services don't have any constructs yet) and at the resource level (an AWS construct exists, but some features are missing).

Note
If you encounter a missing capability in the AWS Construct Library, whether it's an entire library, a specific resource, or a feature, create an issue on GitHub and let us know.

This section describes the following use cases:

- How to access the low-level AWS CloudFormation resources encapsulated by an AWS construct
- How to specify resource options, such as metadata, and dependencies on resources
- How to add overrides to AWS CloudFormation resources and property definitions
- How to directly define low-level AWS CloudFormation resources without an AWS construct

You can also find more information about how to work directly with the AWS CloudFormation layer in AWS Construct Library (p. 35).

Accessing Low-Level Resources

Use construct.findChild() to access any child of a construct by its construct ID. By convention, the main resource of any AWS construct is named **Resource**.

The following example shows how to access the underlying Amazon Simple Storage Service (Amazon S3) bucket resource, given a Bucket construct.

```
// Create an AWS bucket construct
const bucket = new s3.Bucket(this, 'MyBucket');

// The main construct is named "Resource" and
// its type is s3.CfnBucket; const
const bucketResource = bucket.node.findChild('Resource') as s3.CfnBucket;
```

The `bucketResource` represents the low-level AWS CloudFormation resource of type CfnBucket that's encapsulated by the bucket.

If the child can't be located, construct.findChildren(id) fails. This means that if the underlying AWS Construct Library changes the IDs or structure for some reason, synthesis fails.

You can also use construct.children for more advanced queries. For example, you can look for a child that has a certain AWS CloudFormation resource type.

```
const bucketResource =
    bucket.children.find(c => (c as cdk.Resource).resourceType === 'AWS::S3::Bucket')
      as s3.CfnBucket;
```

Once you have a AWS CloudFormation resource, you are interacting with AWS CloudFormation resource classes, which extend cdk.CfnResource.

Resource Options

Set resource options using cdk.CfnResource properties such as *Metadata* and *DependsOn*.

For example, the following code:

```
const bucketResource = bucket.node.findChild('Resource') as s3.CfnBucket;

bucketResource.options.metadata = { MetadataKey: 'MetadataValue' };
bucketResource.options.updatePolicy = {
    autoScalingRollingUpdate: {
        pauseTime: '390'
    }
};

bucketResource.addDependency(otherBucket.node.findChild('Resource') as cdk.Resource);
```

Synthesizes into the following template.

```
{
    "Type": "AWS::S3::Bucket",
    "DependsOn": [ "Other34654A52" ],
    "UpdatePolicy": {
      "AutoScalingRollingUpdate": {
        "PauseTime": "390"
      }
```

```
    },
    "Metadata": {
      "MetadataKey": "MetadataValue"
  }
}
```

Raw Overrides

Use the cdk.CfnResource.addOverride(path, value) method to define an override that is applied to the resource definition during synthesis, as shown in the following example.

```
// Define an override at the resource definition root, you can even modify the "Type"
// of the resource if needed.
bucketResource.addOverride('Type', 'AWS::S3::SpecialBucket');

// fine an override for a property (both are equivalent operations):
bucketResource.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
bucketResource.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');

// se dot-notation to define overrides in complex structures which will be merged
// with the values set by the higher-level construct
bucketResource.addPropertyOverride('LoggingConfiguration.DestinationBucketName',
 otherBucket.bucketName);

// It's also possible to assign a null value
bucketResource.addPropertyOverride('Foo.Bar', null);
```

This synthesizes to the following.

```
{
    "Type": "AWS::S3::SpecialBucket",
    "Properties": {
        "Foo": {
            "Bar": null
        },
        "VersioningConfiguration": {
            "Status": "NewStatus"
        },
        "LoggingConfiguration": {
            "DestinationBucketName": {
              "Ref": "Other34654A52"
            }
        }
    }
}
```

Use `undefined`, cdk.CfnResource.addDeletionOverride, or cdk.CfnResource.addPropertyDeletionOverride to delete values.

```
const bucket = new s3.Bucket(this, 'MyBucket', {
    versioned: true,
    encryption: s3.BucketEncryption.KmsManaged
});

const bucketResource = bucket.node.findChild('Resource') as s3.CfnBucket;
bucketResource.addPropertyOverride('BucketEncryption.ServerSideEncryptionConfiguration.0.EncryptEveryth
 true);
bucketResource.addPropertyDeletionOverride('BucketEncryption.ServerSideEncryptionConfiguration.0.Server
```

This synthesizes to the following.

```
{
    "MyBucketF68F3FF0": {
        "Type": "AWS::S3::Bucket",
        "Properties": {
            "BucketEncryption": {
                "ServerSideEncryptionConfiguration": [
                    {
                        "EncryptEverythingAndAlways": true
                    }
                ]
            },
            "VersioningConfiguration": {
                "Status": "Enabled"
            }
        }
    }
}
```

Directly Defining AWS CloudFormation Resources

You can also explicitly define AWS CloudFormation resources in your stack. To do this, instantiate one of the `CfnXxx` constructs of the dedicated library.

```
new s3.CfnBucket(this, 'MyBucket', {
    analyticsConfigurations: [
      // ...
    ]
});
```

In the rare case where you want to define a resource that doesn't have a corresponding `CfnXxx` class (such as a new resource that wasn't published yet in the AWS CloudFormation resource specification), you can instantiate the cdk.CfnResource.

```
new cdk.Resource(this, 'MyBucket', {
    type: 'AWS::S3::Bucket',
    properties: {
        AnalyticsConfiguration: [ /* ... */ ] // note the PascalCase here
    }
});
```

Create an App with Multiple Stacks

The following example shows one solution to parameterizing how you create a stack. It creates one stack with a `t2.micro` AMI for development, and three stacks with the more expensive `c5.2xlarge` AMI. The development stack and one of the production stacks use the same account to create the stack in `us-east-1`; the other two production stacks use different account IDs and regions.

The file `MyApp-stack.ts` defines a property set that extends the `cdk.StackProps` class to add one additional property, `enc`, which specifies whether to set encryption on the Amazon S3 bucket in the stack.

```
import cdk = require("@aws-cdk/cdk");
import s3 = require("@aws-cdk/aws-s3");

interface MyStackProps extends cdk.StackProps {
  enc: boolean;
```

```
}

export class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props: MyStackProps) {
    super(scope, id, props);

    if (props.enc) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KmsManaged
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket");
    }
  }
}
```

The file `MyApp.ts` creates two stacks. One with an unencrypted bucket in the `us-west-2` region; the other with an encrypted bucket in the `us-east-1` region.

```
import cdk = require("@aws-cdk/cdk");
import { MyStack } from "../lib/MyApp-stack";

const app = new cdk.App();

new MyStack(app, "MyWestCdkStack", {
  env: {
    region: "us-west-2"
  },
  enc: false
});

new MyStack(app, "MyEastCdkStack", {
  env: {
    region: "us-east-1"
  },
  enc: true
});
```

The following example shows how to deploy a stack with an encrypted bucket to the `us-east-1` region.

```
cdk deploy MyEastCdkStack
```

If you look at the stack using **cdk synth MyEastCdkStack**, you should see a bucket similar to the following:

```
  MyGroovyBucketFD9882AC:
    Type: AWS::S3::Bucket
    Properties:
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
              SSEAlgorithm: aws:kms
```

Set a CloudWatch Alarm

The **aws-cloudwatch** package supports setting CloudWatch alarms on CloudWatch metrics. The syntax is as follows, where *METRIC* is a CloudWatch metric you have created, and the alarm is raised there are more than 100 of the measured metrics in two of the last three seconds:

```
new cloudwatch.Alarm(this, 'Alarm', {
  metric: METRIC,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

The syntax for creating a metric for a AWS CloudFormation Resource is as follows, where the *namespace* value should be something like **AWS/SQS** for an Amazon SQS queue.

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensions: { MyDimension: 'MyDimensionValue' }
});
```

Many AWS CDK packages contain an AWS Construct Library construct with functionality to enable setting an alarm based on an existing metric. For example, you can create an Amazon SQS alarm for the **ApproximateNumberOfMessagesVisible** metric that raises an alarm if the queue has more than 100 messages available for retrieval in two of the last three seconds.

```
const qMetric = queue.metric('ApproximateNumberOfMessagesVisible');

new cloudwatch.Alarm(this, 'Alarm', {
  metric: qMetric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

Get a Value from a Context Variable

You can specify a context variable either as part of a AWS CDK Toolkit command, or in `cdk.json`.

To create a command line context variable, use the **--context** (**-c**) option of a AWS CDK Toolkit command, 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 code like the following, which gets the value of the context variable **bucket_name**.

```
const bucket_name = this.node.tryGetContext("bucket_name");
```

AWS CDK Toolchain

This section contains information about AWS CDK tools.

AWS CDK Command Line Interface (cdk)

The AWS CDK Toolkit, **cdk**, is the main tool you use to interact with your AWS CDK app. It executes the AWS CDK app you wrote and compiled, interrogates the application model you defined, and produces and deploys the AWS CloudFormation templates generated by the AWS CDK.

There are two ways to tell **cdk** what command to use to run your AWS CDK app. The first way is to include an explicit **--app** option whenever you use a **cdk** command.

```
cdk --app 'node bin/main.js' synth
```

The second way is to add the following entry to the `cdk.json` file.

```
{
  "app": "node bin/main.js"
}
```

You can also use the **npx cdk** instead of just **cdk**.

Here are the actions you can take on your AWS CDK app (this is the output of the **cdk --help** command).

```
Usage: cdk -a <cdk-app> COMMAND

Commands:
  cdk list                         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. Invoked without TEMPLATE, the app
                                   template will be used.
  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:
  --app, -a             REQUIRED: command-line for executing your app or a cloud
                        assembly directory (e.g. "node bin/my-app.js")  [string]
  --context, -c         Add contextual string parameter (KEY=VALUE)      [array]
  --plugin, -p          Name or path of a node package that extend the CDK
```

```
                            features. Can be specified multiple times        [array]
  --rename                  Rename stack name if different from the one defined in
                            the cloud executable ([ORIGINAL:]RENAMED)       [string]
  --trace                   Print trace for stack warnings                 [boolean]
  --strict                  Do not construct stacks with warnings          [boolean]
  --ignore-errors           Ignores synthesis errors, which will likely produce an
                            invalid output              [boolean] [default: false]
  --json, -j                Use JSON output instead of YAML
                                                        [boolean] [default: false]
  --verbose, -v             Show debug logs             [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]
  --ec2creds, -i            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 user assets (enabled by default)
                                                         [boolean] [default: true]
  --role-arn, -r            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]
  --output, -o              emits the synthesized cloud assembly into a directory
                            (default: cdk.out)                              [string]
  --ci                      Force CI detection. Use --no-ci to disable CI
                            autodetection.              [boolean] [default: false]
  --tags, -t                tags to add to the stack (KEY=VALUE)             [array]
  --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.
```

If your app has a single stack, you don't have to specify the stack name.

If a `cdk.json` or `~/.cdk.json` file exists, options specified there are used as defaults. Settings in `cdk.json` take precedence.

Bootstrapping the AWS CDK

Before you can use the AWS CDK you must bootstrap the AWS CDK to create the infrastructure that the AWS CDK needs. Currently the **bootstrap** command creates only an Amazon S3 bucket.

You incur any charges for what the AWS CDK stores in the bucket. Because the AWS CDK does not remove any objects from the bucket, the bucket can accumulate objects as you use the AWS CDK. You can get rid of the bucket by deleting the **CDKToolkit** stack from your account.

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 change the level of changes that requires approval by specifying:

```
cdk deploy --require-approval LEVEL
```

Where *LEVEL* can be one of the following:

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"
}
```

Version Reporting

To gain insight into how the AWS CDK is used, the versions of libraries 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 to identify stacks using a package with known serious security or reliability issues, and to contact their users with important information.

The AWS CDK reports the name and version of `npm` modules that are loaded into the application at synthesis time, unless their `package.json` file contains the `"private": true` attribute.

The `AWS::CDK::Metadata` resource looks like the following.

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Modules: "@aws-cdk/core=0.7.2-beta,@aws-cdk/s3=0.7.2-beta,lodash=4.17.10"
```

Opting Out from Version Reporting

To opt out of version reporting, use one of the following methods:

- Use the **cdk** command with the **--no-version-reporting** argument.

  ```
  cdk --no-version-reporting synth
  ```

- Set **versionReporting** to **false** in `./cdk.json` or `~/cdk.json`.

  ```
  {
    "app": "...",
    "versionReporting": false
  }
  ```

SAM CLI

This topic describes how to use the 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.

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 lambda = require('@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.Python37,
  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. Compile your AWS CDK app and create a AWS CloudFormation template

```
npm run build
cdk synth > 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"
```

Troubleshooting the AWS CDK

This section describes how to troubleshoot problems with your AWS CDK app.

Inconsistent Module Versions

If you have inconsistent module versions in your `package.json` file or `node_modules` directory, you might see error messages such as the following:

```
lib/my_ecs_construct-stack.ts:56:49 - error TS2345: Argument of type 'this' is not
 assignable to parameter of type 'Construct'.
  Type 'MyEcsConstructStack' is not assignable to type 'Construct'.
    Types of property 'node' are incompatible.
      Property 'root' is missing in type 'ConstructNode' but required in type
 'ConstructNode'.

56     new ecs_patterns.LoadBalancedFargateService(this, "MyNewFargateService", {
                                                   ~~~~

  node_modules/@aws-cdk/aws-ecs-patterns/node_modules/@aws-cdk/cdk/lib/
construct.d.ts:187:14
    187     readonly root: IConstruct;
                     ~~~~
    'root' is declared here.
```

The solution is to delete the `node_modules` directory and re-install your AWS CDK modules:

```
rm -rf node_modules
npm install @aws-cdk/...
```

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-----
```

Document History for the AWS CDK Developer Guide

This document is based on the following release of the AWS Cloud Development Kit (AWS CDK).

- **API version: 0.34.0**
- **Latest documentation update:** June 11, 2019

See Releases for a list of the AWS CDK releases.

update-history-change	update-history-description	update-history-date
Kindle (p. 66)	The developer guide is now available as a free Kindle download.	May 22, 2019
Identifiers (p. 66)	The Concepts section now has information about Identifiers.	May 20, 2019