# Table of Contents

What is the AWS CDK? ................................................................................................................................. 1
Why use the AWS CDK? ............................................................................................................................... 2
Developing with the AWS CDK .................................................................................................................. 6
Contributing to the AWS CDK .................................................................................................................... 7
Additional documentation and resources ................................................................................................. 7
About Amazon Web Services ....................................................................................................................... 7
Getting started ........................................................................................................................................... 9
Your background ...................................................................................................................................... 9
Key concepts ............................................................................................................................................ 9
Supported programming languages ........................................................................................................... 10
Prerequisites ............................................................................................................................................ 11
Install the AWS CDK ................................................................................................................................ 13
AWS CDK tools ........................................................................................................................................ 13
Next steps ................................................................................................................................................... 13
Your first AWS CDK app ............................................................................................................................ 13
Create the app ........................................................................................................................................... 14
Build the app ............................................................................................................................................ 15
List the stacks in the app ............................................................................................................................ 16
Add an Amazon S3 bucket ......................................................................................................................... 16
Synthesize an AWS CloudFormation template .......................................................................................... 19
Deploying the stack ................................................................................................................................... 19
Modifying the app ....................................................................................................................................... 20
Destroying the app's resources .................................................................................................................. 22
Next steps ................................................................................................................................................... 22
Working with the AWS CDK ....................................................................................................................... 23
In TypeScript .............................................................................................................................................. 23
Prerequisites ............................................................................................................................................... 24
Creating a project ....................................................................................................................................... 24
Managing AWS construct library modules ............................................................................................... 24
AWS CDK idioms in TypeScript .................................................................................................................. 24
Building, synthesizing, and deploying ...................................................................................................... 25
In JavaScript .............................................................................................................................................. 26
Prerequisites ............................................................................................................................................... 26
Creating a project ....................................................................................................................................... 26
Managing AWS construct library modules ............................................................................................... 26
AWS CDK idioms in JavaScript .................................................................................................................. 27
Synthesizing and deploying ....................................................................................................................... 28
Using TypeScript examples with JavaScript .............................................................................................. 28
Migrating to TypeScript ............................................................................................................................ 30
In Python .................................................................................................................................................... 31
Prerequisites ............................................................................................................................................... 31
Creating a project ....................................................................................................................................... 31
Managing AWS construct library modules ............................................................................................... 32
AWS CDK idioms in Python ....................................................................................................................... 33
Synthesizing and deploying ....................................................................................................................... 34
In Java ......................................................................................................................................................... 35
Prerequisites ............................................................................................................................................... 35
Creating a project ....................................................................................................................................... 35
Managing AWS construct library modules ............................................................................................... 36
AWS CDK idioms in Java ........................................................................................................................... 36
Building, synthesizing, and deploying ....................................................................................................... 38
In C# .......................................................................................................................................................... 38
Prerequisites ............................................................................................................................................... 39
Creating a project ....................................................................................................................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating from TypeScript</td>
<td>44</td>
</tr>
<tr>
<td>Importing a module</td>
<td>44</td>
</tr>
<tr>
<td>Instantiating a construct</td>
<td>46</td>
</tr>
<tr>
<td>Accessing members</td>
<td>47</td>
</tr>
<tr>
<td>Enum constants</td>
<td>48</td>
</tr>
<tr>
<td>Object interfaces</td>
<td>48</td>
</tr>
<tr>
<td>Concepts</td>
<td>50</td>
</tr>
<tr>
<td>Constructs</td>
<td>50</td>
</tr>
<tr>
<td>AWS Construct library</td>
<td>51</td>
</tr>
<tr>
<td>Composition</td>
<td>51</td>
</tr>
<tr>
<td>Initialization</td>
<td>51</td>
</tr>
<tr>
<td>Apps</td>
<td>65</td>
</tr>
<tr>
<td>The app construct</td>
<td>66</td>
</tr>
<tr>
<td>App lifecycle</td>
<td>68</td>
</tr>
<tr>
<td>Cloud assemblies</td>
<td>69</td>
</tr>
<tr>
<td>Stacks</td>
<td>70</td>
</tr>
<tr>
<td>Stack API</td>
<td>75</td>
</tr>
<tr>
<td>Nested stacks</td>
<td>76</td>
</tr>
<tr>
<td>Environments</td>
<td>77</td>
</tr>
<tr>
<td>Resources</td>
<td>83</td>
</tr>
<tr>
<td>Resource attributes</td>
<td>84</td>
</tr>
<tr>
<td>Referencing resources</td>
<td>84</td>
</tr>
<tr>
<td>Accessing resources in a different stack</td>
<td>85</td>
</tr>
<tr>
<td>Physical names</td>
<td>87</td>
</tr>
<tr>
<td>Passing unique identifiers</td>
<td>88</td>
</tr>
<tr>
<td>Importing existing external resources</td>
<td>90</td>
</tr>
<tr>
<td>Permission grants</td>
<td>93</td>
</tr>
<tr>
<td>Metrics and alarms</td>
<td>94</td>
</tr>
<tr>
<td>Network traffic</td>
<td>96</td>
</tr>
<tr>
<td>Event handling</td>
<td>98</td>
</tr>
<tr>
<td>Removal policies</td>
<td>99</td>
</tr>
<tr>
<td>Identifiers</td>
<td>101</td>
</tr>
<tr>
<td>Construct IDs</td>
<td>102</td>
</tr>
<tr>
<td>Paths</td>
<td>104</td>
</tr>
<tr>
<td>Unique IDs</td>
<td>104</td>
</tr>
<tr>
<td>Logical IDs</td>
<td>105</td>
</tr>
<tr>
<td>Tokens</td>
<td>105</td>
</tr>
<tr>
<td>Tokens and token encodings</td>
<td>106</td>
</tr>
<tr>
<td>String-encoded tokens</td>
<td>108</td>
</tr>
<tr>
<td>List-encoded tokens</td>
<td>109</td>
</tr>
<tr>
<td>Number-encoded tokens</td>
<td>109</td>
</tr>
<tr>
<td>Lazy values</td>
<td>109</td>
</tr>
<tr>
<td>Converting to JSON</td>
<td>111</td>
</tr>
<tr>
<td>Parameters</td>
<td>112</td>
</tr>
<tr>
<td>Defining parameters</td>
<td>112</td>
</tr>
<tr>
<td>Using parameters</td>
<td>113</td>
</tr>
<tr>
<td>Deploying with parameters</td>
<td>115</td>
</tr>
<tr>
<td>Tagging</td>
<td>115</td>
</tr>
</tbody>
</table>
What is the AWS CDK?

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 in a familiar programming language. The AWS CDK supports TypeScript, JavaScript, Python, Java, and C#/.Net.

Developers can use one of the supported programming languages to define reusable cloud components known as **Constructs** (p. 50). You compose these together into **Stacks** (p. 70) and **Apps** (p. 65).
Why use the AWS CDK?

Let's look at the power of the AWS CDK. Here is some code in an AWS CDK project to create an Amazon ECS service with AWS Fargate launch type (this is the code we use in the section called "ECS" (p. 174)).

TypeScript

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

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });
```
Why use the AWS CDK?

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

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

module.exports = { MyEcsConstructStack }

JavaScript

class MyEcsConstructStack extends core.Stack {
    constructor(scope, id, props) {
        super(scope, id, props);
    
        const vpc = new ec2.Vpc(this, "MyVpc", {
            maxAzs: 3 // Default is all AZs in region
        });

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

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

module.exports = { MyEcsConstructStack }

Python

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

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

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

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,             # Required
cpu=512,                        # Default is 256
desired_count=6,                # Default is 1
task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
Java

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

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

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

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

C#

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

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

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

        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
            { Cluster = cluster,
              Cpu = 512,
              DesiredCount = 6,
              TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
```
Why use the AWS CDK?

```python
{   Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
},
MemoryLimitMiB = 2048,
PublicLoadBalancer = true,
}
}

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

- AWS::EC2::EIP
- AWS::EC2::InternetGateway
- AWS::EC2::NatGateway
- AWS::EC2::Route
- AWS::EC2::RouteTable
- AWS::EC2::SecurityGroup
- AWS::EC2::Subnet
- AWS::EC2::SubnetRouteTableAssociation
- AWS::EC2::VPCGatewayAttachment
- AWS::EC2::VPC
- AWS::ECS::Cluster
- AWS::ECS::Service
- AWS::ECS::TaskDefinition
- AWS::ElasticLoadBalancingV2::Listener
- AWS::ElasticLoadBalancingV2::LoadBalancer
- AWS::ElasticLoadBalancingV2::TargetGroup
- AWS::IAM::Policy
- AWS::IAM::Role
- AWS::Logs::LogGroup

Other advantages of the AWS CDK include:

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

Code snippets and longer examples are available in the AWS CDK's supported programming languages: TypeScript, JavaScript, Python, Java, and C#. See AWS CDK examples (p. 202) for a list of the examples.

The AWS CDK Toolkit (p. 253) 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. 50) includes a module for each AWS service with constructs that offer rich APIs that encapsulate the details of how to create resources for an Amazon or AWS service. The
aim of the AWS Construct Library is to reduce the complexity and glue logic required when integrating various AWS services to achieve your goals on AWS.

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

### Contributing to the AWS CDK

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

### Additional documentation and resources

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

- API Reference
- AWS CDK 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. 286)
  - JSII OpenPGP key (p. 287)
- 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 queueing).

AWS uses a pay-as-you-go service model. You are charged only for the services that you — or your applications — use. Also, to make AWS useful as a platform for prototyping and experimentation, AWS offers a free usage tier, in which services are free below a certain level of usage. For more information about AWS costs and the free usage tier, see Test-Driving AWS in the Free Usage Tier.
To obtain an AWS account, go to aws.amazon.com, and then choose **Create an AWS Account**.
Getting started with the AWS CDK

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

Your background

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

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

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

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

Key concepts

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

An AWS CDK app (p. 65) is an application written in TypeScript, JavaScript, Python, Java, or C# that uses the AWS CDK to define AWS infrastructure. An app defines one or more stacks (p. 70). Stacks (equivalent to AWS CloudFormation stacks) contain constructs (p. 50), each of which defines one or more concrete AWS resources, such as Amazon S3 buckets, Lambda functions, Amazon DynamoDB tables, and so on.

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

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

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

Constructs come in three fundamental flavors:

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

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

L2 modules may also define supporting resources needed by the primary resource. Some services have more than one L2 module in the Construct Library for organizational purposes.

Patterns or L3. Patterns declare multiple resources to create entire AWS architectures for particular use cases. All the plumbing is already hooked up, and configuration is boiled down to a few important parameters. In the AWS Construct Library, patterns are in separate modules from L1 and L2 constructs.

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

Supported programming languages

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

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

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

TypeScript

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

JavaScript

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

Python

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

Java

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

With the concepts out of the way, here's what you need to have on your workstation before you install the AWS CDK and start developing.

All CDK developers need to install Node.js 10.3.0 or later, even those working in languages other than TypeScript or JavaScript. The AWS CDK Toolkit (cdk command-line tool) and the AWS Construct Library are developed in TypeScript and run on Node.js. The bindings for other supported languages use this back end and tool set. We suggest the latest LTS version.

Important
Node.js versions 13.0.0 through 13.6.0 are not compatible with the AWS CDK.

You must provide your credentials and an AWS Region to use AWS CDK, if you have not already done so.

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

If you have the AWS CLI installed, the easiest way to satisfy this requirement is to install the AWS CLI and issue the following command:
aws configure

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

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

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

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

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

```
[default]
aws_access_key_id=AKIAI44QH8DHBEREXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Ut7h3yCo8nvbEXAMPLEKEY
```

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

Other prerequisites depend on your development language and are as follows.

TypeScript

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

JavaScript

- No additional requirements

Python

- Python 3.6 or later including pip and virtualenv

Java

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

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

C#

- A .NET Standard 2.1-compatible implementation is required, such as.
  - .NET Core 3.1 or later
  - .NET Framework 4.6.1 or later
  - Mono 5.4 or later

Install the AWS CDK

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

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

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

```bash
cdk --version
```

AWS CDK tools

We've already been using the AWS CDK Toolkit, also known as the Command Line Interface (CLI). It's 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 CDK templates it generates. It also has deployment, diff, deletion, and troubleshooting capabilities. For more information, see `cdk --help` or the section called “AWS CDK Toolkit” (p. 253).

The [AWS Toolkit for Visual Studio Code](https://github.com/aws/aws-visualstudio-code-cdk) is an open-source plug-in for Visual Studio Code that makes it easier to create, debug, and deploy applications on AWS. The toolkit provides an integrated experience for developing AWS CDK applications, including the AWS CDK Explorer feature to list your AWS CDK projects and browse the various components of the CDK application. Install the plug-in and learn more about using the AWS CDK Explorer.

Next steps

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

- Come on in; the water's fine! Build your first AWS CDK app (p. 13).
- Try the [CDK Workshop](https://github.com/aws/aws-visualstudio-code-cdk) for a more in-depth tour involving a more complex project.
- See the API reference to begin exploring the CDK constructs available for your favorite AWS services.
- Dig deeper into concepts like the section called “Environments” (p. 77), the section called “Assets” (p. 121), the section called “Permissions” (p. 135), the section called “Context” (p. 142), the section called “Parameters” (p. 112), and the section called “Escape hatches” (p. 151).
- Explore Examples of using the AWS CDK.

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

Your first AWS CDK app

You've read *Getting started* (p. 9)? Great! Now let's see how it feels to work with the AWS CDK by building the simplest possible AWS CDK app. In this tutorial you'll learn about the structure of a AWS CDK project, how to work with the AWS Construct Library, and how to use the AWS CDK Toolkit command-line tool.

The standard AWS CDK development workflow is similar to the workflow you're already familiar with as a developer, just with a few extra steps to synthesize your stack to an AWS CloudFormation template and deploy it.
Create the app

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

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

**Important**

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

Now initialize the app using the `cdk init` command, specifying the desired template ("app") and programming language.

```
cdk init TEMPLATE --language LANGUAGE
```

That is:

**TypeScript**

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

**JavaScript**

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

**Python**

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

After the app has been created, also enter the following two commands to activate the app's Python virtual environment and install its dependencies.
source .env/bin/activate
python -m pip install -r requirements.txt

Java

cdk init app --language java

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

C#

cdk init app --language csharp

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

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

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

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

Build the app

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

TypeScript

```bash
npm run build
```

JavaScript

No build step is necessary.

Python

No build step is necessary.

Java

```bash
mvn compile -q
```

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

C#

```bash
dotnet build src
```
List the stacks in the app

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

cdk ls

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

Add an Amazon S3 bucket

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

Install the Amazon S3 package from the AWS Construct Library.

TypeScript

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

JavaScript

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

Python

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

Java

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

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

If you are using a Java IDE, it probably has a simpler way to add this dependency to your project. Resist temptation and edit `pom.xml` by hand.

C#

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

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

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

Next, define an Amazon S3 bucket in the stack using an L2 construct, the Bucket class.
Add an Amazon S3 bucket

**TypeScript**

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

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

export class HelloCdkStack extends core.Stack {
  constructor(scope: core.App, id: string, props?: core.StackProps) {
    super(scope, id, props);
    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}
```

**JavaScript**

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

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

class HelloCdkStack extends core.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);
    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

module.exports = { HelloCdkStack }
```

**Python**

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

```python
from aws_cdk import (  
    aws_s3 as s3,  
    core  
)
```

Replace the comment with the following code.

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

**Java**

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

```java
package com.myorg;

import software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.s3.Bucket;
```
public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

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

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

C#

Update HelloCdkStack.cs to look like this.

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

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

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

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

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

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

It's interesting to take note of how props are represented in the different supported languages.

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

Synthesize an AWS CloudFormation template

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

```bash
cdk synth
```

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

Tip

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

The `cdk synth` command executes your app, which causes the resources defined in it to be translated to an AWS CloudFormation template. The output of `cdk synth` is a YAML-format AWS CloudFormation template, which looks something like this.

```
Resources:
  MyFirstBucketB8884501:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
      UpdateReplacePolicy: Retain
      DeletionPolicy: Retain
      Metadata:
        aws:cdk:path: HelloCdkStack/MyFirstBucket/Resource
      CDKMetadata:
        Type: AWS::CDK::Metadata
        Properties:
```

Even if you aren't very familiar with AWS CloudFormation, you should be able to find the definition for an `AWS::S3::Bucket` and see how the versioning configuration was translated.

Note

Every generated template contains a `AWS::CDK::Metadata` resource by default. The AWS CDK team uses this metadata to gain insight into how the AWS CDK is used, so we can continue to improve it. For details, including how to opt out of version reporting, see Version reporting (p. 254).

The `cdk synth` generates a perfectly valid AWS CloudFormation template. You could take it and deploy it using the AWS CloudFormation console. But the AWS CDK Toolkit also has that feature built-in.

Deploying the stack

To deploy the stack using AWS CloudFormation, issue:
cdk deploy

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

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

If your code changes have security implications, you'll see a summary of these, and be asked to confirm them before deployment proceeds.

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

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

Modifying the app

The AWS CDK can update your deployed resources after you modify your app. Let's make a little change to our bucket. We want to be able to delete the bucket automatically when we delete the stack, so we'll change the `RemovalPolicy`.

**TypeScript**

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

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

**JavaScript**

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

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

**Python**

Update `hello_cdk/hello_cdk_stack.py`

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

**Java**

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

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

Bucket.Builder.create(this, "MyFirstBucket")
```
Now we'll use the `cdk diff` command to see the differences between what's already been deployed, and the code we just changed.

```bash
cdk diff
```

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

```
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
## [~] DeletionPolicy
#   ## [-] Retain
#   ## [+] Delete
## [~] UpdateReplacePolicy
## [-] Retain
## [+] Delete
```

As you can see, the diff indicates that the `DeletionPolicy` property of the bucket is now set to `Delete`, enabling the bucket to be deleted when its stack is deleted. The `UpdateReplacePolicy` is also changed.

Don't be confused by the difference in name. The AWS CDK calls it `RemovalPolicy` because its meaning is slightly different from AWS CloudFormation's `DeletionPolicy`: the AWS CDK default is to retain the bucket when the stack is deleted, while AWS CloudFormation's default is to delete it. See the section called "Removal policies" (p. 99) for further details.

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

Now let's deploy.

```bash
cdk deploy
```

Enter `y` to approve the changes and deploy the updated stack. The Toolkit updates the bucket configuration as you requested.
Destroying the app's resources

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

```
cdk destroy
```

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

**Note**

This wouldn't have worked if we hadn't changed the bucket's `RemovalPolicy` just a minute ago!

If `cdk destroy` fails, it probably means you put something in your Amazon S3 bucket. AWS CloudFormation won't delete buckets with files in them. Delete the files and try again.

Next steps

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

- Try the [CDK Workshop](#) for a more in-depth tour involving a more complex project.
- See the [API reference](#) to begin exploring the CDK constructs available for your favorite AWS services.
- Dig deeper into concepts like the section called “Environments” (p. 77), the section called “Assets” (p. 121), the section called “Permissions” (p. 135), the section called “Context” (p. 142), the section called “Parameters” (p. 112), and the section called “Escape hatches” (p. 151).
- Explore [Examples](#) of using the AWS CDK.

The AWS CDK is an open-source project. Want to [contribute](#)?
Working with the AWS CDK

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

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

AWS CDK prerequisites

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

Tip

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

```
aws configure
```

All AWS CDK applications require Node.js 10.3 or later, even when your app is written in Python, Java, or C#. You may download a compatible version for your platform at nodejs.org. We recommend the current LTS version (at this writing, the latest 12.x release).

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

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

Test the installation by issuing cdk --version.

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

Topics

- Working with the AWS CDK in TypeScript (p. 23)
- Working with the AWS CDK in JavaScript (p. 26)
- Working with the AWS CDK in Python (p. 31)
- Working with the AWS CDK in Java (p. 35)
- Working with the AWS CDK in C# (p. 38)

Working with the AWS CDK in TypeScript

TypeScript is a fully-supported client language for the AWS CDK and is considered stable. Working with the AWS CDK in TypeScript uses familiar tools, including Microsoft's TypeScript compiler (tsc), Node.js and the Node Package Manager (npm). You may also use Yarn if you prefer, though the examples in this Guide use NPM. The modules comprising the AWS Construct Library are distributed via the NPM repository, npmjs.org.
You can use any editor or IDE; many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has excellent support for TypeScript.

**Prerequisites**

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 23).

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

```bash
npm install -g typescript
```

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

**Creating a project**

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

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

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

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

**Managing AWS construct library modules**

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

AWS Construct Library modules are named like `@aws-cdk/SERVICE-NAME`. For example, the command below installs the modules for Amazon S3 and AWS Lambda.

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

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

```bash
npm update
```

**Note**

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

**AWS CDK idioms in TypeScript**

**Props**

All AWS Construct Library classes are instantiated using three arguments: the `scope` in which the construct is being defined (its parent in the construct tree), a `name`, and `props`, a bundle of key/value pairs that the construct uses to configure the AWS resources it creates. Other classes and methods also use the “bundle of attributes” pattern for arguments.
In TypeScript, the shape of props is defined using an interface that tells you the required and optional arguments and their types. Such an interface is defined for each kind of props argument, usually specific to a single construct or method. For example, the Bucket construct (in the @aws-cdk/aws-s3 module) specifies a props argument conforming to the BucketProps interface.

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

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

However, we do occasionally add properties to constructs. If a property we add in a later version happens to have the same name as one you're accepting, passing it up the chain can cause unexpected behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to undefined. For example:

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

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

**Missing values**

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

**Building, synthesizing, and deploying**

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

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

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

The stacks (p. 70) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

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

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

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

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

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

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

For full documentation of the cdk command, see the section called “AWS CDK Toolkit” (p. 253).

Working with the AWS CDK in JavaScript

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

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

Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 23).

JavaScript AWS CDK applications require no additional prerequisites beyond these.

Creating a project

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

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

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

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

Managing AWS construct library modules

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

AWS Construct Library modules are named like @aws-cdk/SERVICE-NAME. For example, the command below installs the modules for Amazon S3 and AWS Lambda.

npm install @aws-cdk/aws-s3 @aws-cdk/aws-lambda
Your project's dependencies are maintained in `package.json`. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project's dependencies:

```bash
npm update
```

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

## AWS CDK idioms in JavaScript

### Props

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

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

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

However, we do occasionally add properties to constructs. If a property we add in a later version happens to have the same name as one you're accepting, passing it up the chain can cause unexpected behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to undefined. For example:

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

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

### Missing values

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

```javascript
// a may be undefined, but if it is not, it may have an attribute b
// c is undefined if a is undefined, OR if a doesn't have an attribute b
let c = a && a.b;
```

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

```javascript
let c = a == null ? a : a.b;
```
A version of the ECMAScript standard currently in development specifies new operators that will simplify the handling of undefined values. Using them can simplify your code, but you will need a new version of Node.js to use them. For more information, see the optional chaining and nullish coalescing proposals.

**Synthesizing and deploying**

The stacks (p. 70) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project’s main directory when you issue them.

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

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdk synth</td>
<td>Synthesizes a AWS CloudFormation template</td>
</tr>
<tr>
<td>cdk deploy Happy Grumpy</td>
<td>Deploys the resources defined by the Happy Grumpy stack.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdk synth &quot;Stack?&quot;</td>
<td>Synthesizes all stacks that start with &quot;Stack&quot;.</td>
</tr>
<tr>
<td>cdk deploy &quot;*Stack&quot;</td>
<td>Deploys all stacks that start with &quot;PipeStack&quot;, &quot;LambdaStack&quot;, etc.</td>
</tr>
</tbody>
</table>

**Tip**

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

For full documentation of the cdk command, see the section called “AWS CDK Toolkit” (p. 253).

**Using TypeScript examples with JavaScript**

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

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

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

**TypeScript**

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

**JavaScript**

```javascript
const cdk = require('@aws-cdk/core');
```
Exports can be assigned to the `module.exports` object.

**TypeScript**

```
export class Stack1 extends cdk.Stack {
    // ...
}

export class Stack2 extends cdk.Stack {
    // ...
}
```

**JavaScript**

```
class Stack1 extends cdk.Stack {
    // ...
}
class Stack2 extends cdk.Stack {
    // ...

module.exports = { Stack1, Stack2 }
```

**Note**

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

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

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

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

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

**TypeScript**

```
var encrypted: boolean = true;

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

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

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

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

When a function or method returns a general-purpose type (such as `object`), but you want to treat that value as a more specific child type to access properties or methods that are not part of the more general type's interface, TypeScript lets you cast the value using `as` followed by a type or interface name. TypeScript will make sure that the object you use as, for example, an argument to a function has the required properties of the right types.

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

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

## Migrating to TypeScript

As their projects get larger and more complex, many JavaScript developers move to TypeScript. TypeScript is a superset of JavaScript—all JavaScript code is valid TypeScript code, so no changes to your code are required—and it is also a supported AWS CDK language. Type annotations and other TypeScript features are optional and can be added to your AWS CDK app as you find value in them. TypeScript also gives you early access to new JavaScript features, such as optional chaining and nullish coalescing, before they're finalized—and without requiring that you upgrade Node.js.

TypeScript's "shape-based" interfaces, which define bundles of required and optional properties (and their types) within an object, allow common mistakes to be caught while you're writing the code, and make it easier for your IDE to provide robust autocomplete and other real-time coding advice.
Coding in TypeScript does involve an additional step: compiling your app with the TypeScript compiler, `tsc`. This step can happen automatically whenever you save your source code, or before you run your app. For typical AWS CDK apps, compilation requires a few seconds at most.

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

## Working with the AWS CDK in Python

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

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

### Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 23).

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

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

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

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

**Note**

It is common for Linux distros to use the executable name `python3` for Python 3.x, and have `python` refer to a Python 2.x installation. You can adjust the command used to run your application by editing `cdk.json` in the project's main directory.

### Creating a project

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

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

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

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

source .env/bin/activate

Note
You may recognize this as the Mac/Linux command to activate a virtual environment. The Python templates include a batch file, source.bat, that allows the same command to be used on Windows. The traditional Windows command, .env\Scripts\activate.bat, works, too.

Then install the app's standard dependencies:

python -m pip install -r requirements.txt

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

Managing AWS construct library modules

Use the Python package installer, pip, to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. pip also installs the dependencies for those modules automatically. To run pip without needing it installed in a special directory, invoke it as:

python -m pip PIP-COMMAND

AWS Construct Library modules are named like aws-cdk.SERVICE-NAME. For example, the command below installs the modules for Amazon S3 and AWS Lambda.

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

Similar names are used for importing AWS Construct Library modules into your Python code (just replace the hyphens with underscores).

import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lam

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

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

aws-cdk.aws-s3==X.YY.ZZ
aws-cdk.aws-lambda==X.YY.ZZ
You can edit requirements.txt to allow upgrades; simply replace the `==` preceding a version number with `~=` to allow upgrades to a higher compatible version, or remove the version requirement entirely to specify the latest available version of the module.

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

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

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

## AWS CDK idioms in Python

### Props

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

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

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

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

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

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

### Missing values

The AWS CDK uses `None` to represent missing or undefined values. When working with `**kwargs`, use the dictionary's `get()` method to provide a default value if a property is not provided. Avoid using `kwargs[...]`, as this raises `KeyError` for missing values.
Some AWS CDK methods (such as `tryGetContext()` to get a runtime context value) may return `None`, which you will need to check explicitly.

### Using interfaces

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

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

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

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

### Type pitfalls

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

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

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

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

### Synthesizing and deploying

The stacks (p. 70) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project’s main directory when you issue them.

- `cdk synth`: Synthesizes a AWS CloudFormation template from one or more of the stacks in your AWS CDK app.
- `cdk deploy`: Deploys the resources defined by one or more of the stacks in your AWS CDK app to AWS.
You can specify the names of multiple stacks to be synthesized or deployed in a single command. If your app defines only one stack, you do not need to specify it.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk synth</code></td>
<td># app defines single stack</td>
</tr>
<tr>
<td><code>cdk deploy Happy Grumpy</code></td>
<td># app defines two or more stacks; two are deployed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk synth &quot;Stack?&quot;</code></td>
<td># Stack1, StackA, etc.</td>
</tr>
<tr>
<td><code>cdk deploy &quot;*Stack&quot;</code></td>
<td># PipeStack, LambdaStack, etc.</td>
</tr>
</tbody>
</table>

**Tip**

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

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 253).

## Working with the AWS CDK in Java

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

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

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

### Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See [AWS CDK Prerequisites](#) (p. 23).

Java AWS CDK applications require Java 8 (v1.8) or later. We recommend [Amazon Corretto](https://aws.amazon.com/corretto/), but you can use any OpenJDK distribution or [Oracle’s JDK](https://www.oracle.com/java/technologies/java-se-jdk8.html). You will also need [Apache Maven](https://mvnrepository.com/) 3.5 or later. You can also use tools such as Gradle, but the application skeletons generated by the AWS CDK Toolkit are Maven projects.

### Creating a project

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

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

cdk init uses the name of the project folder to name various elements of the project, including classes, subfolders, and files.
The resulting project includes a reference to the `software.amazon.awscdk.core` Maven package. It and its dependencies are automatically installed by Maven.

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

## Managing AWS construct library modules

Use Maven to install AWS Construct Library packages, which are in the group `software.amazon.awscdk` and named for their service. For example, the Maven artifact ID for Amazon S3 is `s3`. Its Java package name, for use in import statements, is `software.amazon.awscdk.services.s3`. Search the Maven Central Repository to find the names of all AWS Construct Module libraries.

**Note**

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

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

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

**Tip**

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

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

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

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

## AWS CDK idioms in Java

### Props

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

In Java, props are expressed using the Builder pattern. Each construct type has a corresponding props type; for example, the `Bucket` construct (which represents an Amazon S3 bucket) takes as its props an instance of `BucketProps`. 
The `BucketProps` class (like every AWS Construct Library props class) has an inner class called `Builder`. The `BucketProps.Builder` type offers methods to set the various properties of a `BucketProps` instance. Each method returns the `Builder` instance, so the method calls can be chained to set multiple properties. At the end of the chain, you call `build()` to actually produce the `BucketProps` object.

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

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

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

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

**Generic structures**

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

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

**Note**

The double-brace notation (which technically declares an anonymous inner class) is sometimes considered an anti-pattern. However, its disadvantages are not very relevant to this use case, and it is a reasonably compact way to write what would be object or dictionary literals in other languages.

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

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

**Missing values**

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

### Building, synthesizing, and deploying

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

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

The stacks (p. 70) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project’s main directory when you issue them.

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

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

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

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

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

**Tip**

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

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 253).

### Working with the AWS CDK in C#

.NET is a fully-supported client platform for the AWS CDK and is considered stable. Our .NET code examples are in C#. It is possible to write AWS CDK applications in other .NET languages, such as Visual Basic or F#, but we are unable to provide support for these languages.

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

We suggest using Visual Studio 2019 (any edition) and the Microsoft .NET Framework on Windows to develop AWS CDK apps in C#. You may use other tools (for example, Mono on Mac OS X or Linux), but our ability to provide instruction and support for these environments may be limited.
Prerequisites

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See AWS CDK Prerequisites (p. 23).

C# AWS CDK applications require a .NET Standard 2.1 compatible implementation. Suitable implementations include:

- .NET Core v3.1 or later
- .NET Framework v4.6.1 or later
- Mono v5.4 or later on Mac OS X or Linux; download here

If you have an up-to-date Windows 10 installation, you already have a suitable installation of .NET Framework.

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

Creating a project

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

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

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

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

Managing AWS construct library modules

The .NET ecosystem uses the NuGet package manager. AWS Construct Library modules are named like `Amazon.CDK.AWS.SERVICE-NAME`. For example, the NuGet package name for the Amazon S3 module is `Amazon.CDK.AWS.S3`.

**Note**

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

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

The Visual Studio NuGet GUI

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

**Note**

All AWS Construct Library modules deemed "experimental" (see the section called “Versioning” (p. 158)) are flagged as pre-release in NuGet.
Look in the Updates panel to install new versions of your packages.

**The NuGet console**

The NuGet console is a PowerShell-based interface to NuGet that works in the context of a Visual Studio project. You can open it in Visual Studio by choosing Tools > NuGet Package Manager > Package Manager Console. For more information about using this tool, see Install and Manage Packages with the Package Manager Console in Visual Studio.

**The dotnet command**

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

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

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

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

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

To install a specific version of a package, include the --v flag and the desired version. AWS Construct Library modules that are deemed "experimental" (see the section called "Versioning" (p. 158)) are flagged as pre-release in NuGet, and must be installed using an explicit version number.

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

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

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

**The nuget command**

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

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

**AWS CDK idioms in C#**

**Props**

All AWS Construct Library classes are instantiated using three arguments: the scope in which the construct is being defined (its parent in the construct tree), a name, and props, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.
In C#, props are expressed using a props type. In idiomatic C# fashion, we can use an object initializer to set the various properties. Here we're creating an Amazon S3 bucket using the `Bucket` construct; its corresponding props type is `BucketProps`.

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

**Tip**

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

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

```csharp
// extend BucketProps for use with MimeBucket
class MimeBucketProps : BucketProps {
    public string MimeType { get; set; }
}

// hypothetical bucket that enforces MIME type of objects inside it
class MimeBucket : Bucket {
    public MimeBucket(final Construct scope, final string id, final MimeBucketProps props=null) : base(scope, id, props) {
        // ...
    }
}

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

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

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

**Generic structures**

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

**Missing values**

In C#, missing values in AWS CDK objects such as props are represented by `null`. The null-conditional member access operator `?.` and the null coalescing operator `??` are convenient for working with these values.
// mimeType is null if props is null or if props.MimeType is null
string mimeType = props?.MimeType;

// mimeType defaults to text/plain. either props or props.MimeType can be null
string MimeType = props?.MimeType ?? "text/plain";

## Building, synthesizing, and deploying

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

The stacks (p. 70) defined in your AWS CDK app can be deployed individually or together using the commands below. Generally, you should be in your project's main directory when you issue them.

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

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk synth</code></td>
<td># app defines single stack</td>
</tr>
<tr>
<td><code>cdk deploy Happy Grumpy</code></td>
<td># app defines two or more stacks; two are deployed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk synth &quot;Stack?&quot;</code></td>
<td># Stack1, StackA, etc.</td>
</tr>
<tr>
<td><code>cdk deploy &quot;*Stack&quot;</code></td>
<td># PipeStack, LambdaStack, etc.</td>
</tr>
</tbody>
</table>

**Tip**

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

For full documentation of the `cdk` command, see the section called “AWS CDK Toolkit” (p. 253). 
Translating TypeScript AWS CDK code to other languages

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

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

- the section called “In TypeScript” (p. 23)
- the section called “In JavaScript” (p. 26)
- the section called “In Python” (p. 31)
- the section called “In Java” (p. 35)
- the section called “In C#” (p. 38)

Importing a module

TypeScript/JavaScript

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

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

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

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

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

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

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

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

Python

Like TypeScript, Python supports namespaced module imports and selective imports. Module names in Python look like `aws_cdk.xxx`, where `xxx` represents an AWS service name, such as `s3` for Amazon S3 (we'll use Amazon S3 for our examples).
# Import entire module as s3 into current namespace
import aws_cdk.aws_s3 as s3

# s3 can now be used to access classes it contains
bucket = s3.Bucket(...)

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

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

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

Java

Java's imports work differently from TypeScript's. Each import statement imports either a single class name from a given package, or all classes defined in that package (using *). After importing, classes may be accessed using either the class name by itself or (in case of name conflicts) the qualified class name including its package.

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

```java
// Make all Amazon S3 construct library classes available
import software.amazon.awscdk.services.s3.*;

// Make only Bucket and EventType classes available
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that name does not conflict with another class)
Bucket bucket = new Bucket(...);

// We can always use the qualified name of a class (including its package) even without an import directive
software.amazon.awscdk.services.s3.Bucket bucket =
    new software.amazon.awscdk.services.s3.Bucket(...);
```

C#

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

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

```csharp
// Make all Amazon S3 construct library classes available
using Amazon.CDK.AWS.S3;

// Now we can access any S3 type using its name
var bucket = new Bucket(...);

// Import the S3 namespace under an alias
using s3 = Amazon.CDK.AWS.S3;
```
Instantiating a construct

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

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

TypeScript/JavaScript

```javascript
// Instantiate default Bucket
const bucket = new s3.Bucket(this, 'MyBucket');

// Instantiate Bucket with bucketName and versioned properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'my-bucket',
  versioned: true,
});

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
const bucket = new s3.Bucket(this, 'MyBucket', {
  websiteRedirect: {host: 'aws.amazon.com'}
});
```

Python

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

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

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

```python
# Instantiate default Bucket
bucket = s3.Bucket(self, "MyBucket")

# Instantiate Bucket with bucket_name and versioned properties
bucket = s3.Bucket(self, "MyBucket", bucket_name="my-bucket", versioned=true)

# Instantiate Bucket with website_redirect, which has its own sub-properties
bucket = s3.Bucket(self, "MyBucket", website_redirect=s3.WebsiteRedirect(host_name="aws.amazon.com"))
```
Java

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

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

Props are named the same as in TypeScript, using `camelCase`.

```java
// Instantiate default Bucket
Bucket bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with bucketName and versioned properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
    .bucketName("my-bucket").versioned(true)
    .build();

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
Bucket bucket = Bucket.Builder.create(self, "MyBucket")
    .websiteRedirect(new WebsiteRedirect.Builder()
        .hostName("aws.amazon.com").build())
    .build();
```

C#

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

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

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

```csharp
// Instantiate default Bucket
var bucket = Bucket(self, "MyBucket");

// Instantiate Bucket with BucketName and versioned properties
var bucket = Bucket(self, "MyBucket", new BucketProps {
    BucketName = "my-bucket",
    Versioned = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
var bucket = Bucket(self, "MyBucket", new BucketProps {
    WebsiteRedirect = new WebsiteRedirect {
        HostName = "aws.amazon.com"
    }});
```

Accessing members

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

TypeScript/JavaScript

Names are `camelCase`.  

47
Enum constants

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

```typescript
s3.BucketEncryption.KMS_MANAGED
```

Object interfaces

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

**TypeScript/JavaScript**

*Note*

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

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

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

**Python**

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

```python
import jsii

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

---

bucket.bucketArn

Python

Names are `snake_case`.

```python
bucket.bucket_arn
```

Java

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

```java
bucket.getBucketArn()
```

C#

Names are `PascalCase`.

```c#
bucket.BucketArn
```
from aws_cdk.core import IAspect, IConstruct
import jsii

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

Java

import software.amazon.awscdk.core.IAspect;
import software.amazon.awscdk.core.IConstruct;

public class MyAspect implements IAspect {

    public void visit(IConstruct node) {
        System.out.format("Visited %s", node.getNode().getPath());
    }
}

c#  

using Amazon.CDK;

public class MyAspect : IAspect {

    public void Visit(IConstruct node) {
    }
}
 Constructs

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

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

AWS Construct library

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

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

There are different levels of constructs in this library, beginning with low-level constructs, which we call CFN Resources (or L1, short for "level 1"). These constructs directly represent all of the AWS resources that are available in AWS CloudFormation. CFN Resources are periodically generated from the AWS CloudFormation Resource Specification. They are named CfnXyz, where Xyz is name of the resource. For example, s3.CfnBucket represents the AWS::S3::Bucket CFN Resource. When you use CFN resources, you must explicitly configure all resource properties, which requires a complete understanding of the details of the underlying AWS CloudFormation resource model.

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

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

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

The key pattern for defining higher-level abstractions through constructs is called composition. A high-level construct can be composed from any number of lower-level constructs, and in turn, those could be composed from even lower-level constructs. To enable this pattern, constructs are always defined within the scope of another construct. This scoping pattern results in a hierarchy of constructs known as a construct tree. In the AWS CDK, the root of the tree represents your entire AWS CDK app (p. 65). Within the app, you typically define one or more stacks (p. 70), which are the unit of deployment, analogous to AWS CloudFormation stacks. Within stacks, you define resources, or other constructs that eventually contain resources.

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

Initialization

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

- **Scope** – The construct within which this construct is defined. You should almost always pass this for the scope, because it represents the current scope in which you are defining the construct.
- **id** – An identifier (p. 101) that must be unique within this scope. The identifier serves as a namespace for everything that's encapsulated within the scope's subtree and is used to allocate unique identities such as resource names (p. 87) and AWS CloudFormation logical IDs.
- **Props** – A set of properties or keyword arguments, depending upon the supported language, that define the construct's initial configuration. In most cases, constructs provide sensible defaults, and if all props elements are optional, you can leave out the props parameter completely.

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

Apps and stacks

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

TypeScript

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

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

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

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

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

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

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

Python

from aws_cdk.core import App, Stack
from aws_cdk import aws_s3 as s3

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

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

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

Java

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

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

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

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

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

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

As you can see, you need a scope within which to define your bucket. Since resources eventually need to be deployed as part of a AWS CloudFormation stack into an AWS environment (p. 77), which covers a specific AWS account and AWS region. AWS constructs, such as `s3.Bucket`, must be defined within the scope of a `Stack`.

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

TypeScript

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

JavaScript

```javascript
class HelloCdkStack extends Stack {
    constructor(scope, id, props) {
        super(scope, id, props);
            
            //...
    }
}
```

Python

```python
class HelloCdkStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
```
Using L1 constructs

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

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

**TypeScript**

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

**JavaScript**

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

**Python**

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

**Java**

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

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

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

TypeScript

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

JavaScript

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

Python

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

Java

```java
CfnBucket bucket = CfnBucket.Builder.create(this, "MyBucket")
    .bucketName("MyBucket")
    .corsConfiguration(new
        CfnBucket.CorsConfigurationProperty.Builder()
            .corsRules(Arrays.asList(new
                CfnBucket.CorsRuleProperty()                .allowedOrigins(Arrays.asList("*"))
                .allowedMethods(Arrays.asList("GET")))
```
Using L2 constructs

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

**TypeScript**

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

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

**JavaScript**

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

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

**Python**

```python
from aws_cdk import aws_s3 as s3

# "self" is HelloCdkStack
```
Configuration

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

**TypeScript**

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

**JavaScript**

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

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

**Note**

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

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

For example, almost all AWS constructs have a set of grant (p. 135) methods that you can use to grant AWS Identity and Access Management (IAM) permissions on that construct to a principal. The following example grants the IAM group `data-science` permission to read from the Amazon S3 bucket `raw-data`.

**TypeScript**

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

**JavaScript**

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

**Python**

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

Java

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

C#

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

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

TypeScript

```typescript
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_10_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

JavaScript

```javascript
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_10_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

Python

```python
jobs_queue = sqs.Queue(self, "jobs")
create_job_lambda = lambda_.Function(self, "create-job",
  runtime=lambda_.Runtime.NODEJS_10_X,
  handler="index.handler",
  code=lambda_.Code.from_asset("./create-job-lambda-code"),
  environment=dict(
    QUEUE_URL=jobs_queue.queue_url
  )
)
```

Java

```java
final Queue jobsQueue = new Queue(this, "jobs");
Function createJobLambda = Function.Builder.create(this, "create-job")
  .handler("index.handler")
```
Authoring constructs

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

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

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

**TypeScript**

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

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

**JavaScript**

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

Python

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

Java

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

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

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

    public NotifyingBucket(final Construct scope, final String id, final BucketProps props, final String prefix) {
        super(scope, id);
        Bucket bucket = new Bucket(this, "bucket");
        Topic topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
    }
}

C#

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

public class NotifyingBucket : Construct {
    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props = null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        var topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new NotificationKeyFilter
The `NotifyingBucket` constructor has a signature compatible with the base `Construct` class: `scope`, `id`, and `props`. The last argument, `props`, is optional (gets the default value `{}`) because all `props` are optional. This means that you could define an instance of this construct in your app without `props`, for example:

**TypeScript**
```typescript
new NotifyingBucket(this, 'MyNotifyingBucket');
```

**JavaScript**
```javascript
new NotifyingBucket(this, 'MyNotifyingBucket');
```

**Python**
```python
NotifyingBucket(self, "MyNotifyingBucket")
```

**Java**
```java
new NotifyingBucket(this, "MyNotifyingBucket");
```

**C#**
```csharp
new NotifyingBucket(this, "MyNotifyingBucket");
```

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

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

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

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

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

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

**TypeScript**

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

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

**JavaScript**

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

**Python**

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

**Java**

```java
public class NotifyingBucket extends Bucket {

    public Topic topic = null;

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

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

public NotifyingBucket(final Construct scope, final String id, final BucketProps props, final String prefix) {
    super(scope, id, props);
    Bucket bucket = new Bucket(this, "bucket");
    topic = new Topic(this, "topic");
    if (prefix != null)
        bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
}

C#

public class NotifyingBucket : Construct
{
    public readonly Topic topic;

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

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

TypeScript

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

JavaScript

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

Python

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

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

C#

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

Apps

As described in the section called “Constructs” (p. 50), to provision infrastructure resources, all constructs that represent AWS resources must be defined, directly or indirectly, within the scope of a `Stack` construct.

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

TypeScript

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

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

JavaScript

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

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

Python

```python
class MyFirstStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

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

Java

```java
public class MyFirstStack extends Stack {
    public MyFirstStack(final Construct scope, final String id) {
```
To define the previous stack within the scope of an application, use the `App` construct. The following example app instantiates a `MyFirstStack` and produces the AWS CloudFormation template that the stack defined.

**TypeScript**

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

**JavaScript**

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

**Python**

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

**Java**

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

**C#**

```csharp
var app = new App();
new MyFirstStack(app, "hello-cdk");
app.Synth();
```
The `App` construct doesn't require any initialization arguments, because it's the only construct that can be used as a root for the construct tree. You can now use the `App` instance as a scope for defining a single instance of your stack.

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

TypeScript

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

JavaScript

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

Python

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

MyApp().synth()
```

Java

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

import software.amazon.awscdk.core.App;

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

// Main.java
package com.myorg;

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

C#

```csharp
public class MyApp : App
{
    public MyApp(AppProps props = null) : base(props)
```
These two methods are equivalent.

**App lifecycle**

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

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

1. **Construction (or Initialization)**

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

2. **Preparation**

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

3. **Validation**

   All constructs that have implemented the `validate` method can validate themselves to ensure that they're in a state that will correctly deploy. You will get notified of any validation failures that happen during this phase. Generally, we recommend that you perform validation as soon as possible (usually as soon as you get some input) and throw exceptions as early as possible. Performing validation early improves diagnosability as stack traces will be more accurate, and ensures that your code can continue to execute safely.
4. Synthesis

This is the final stage of the execution of your AWS CDK app. It's triggered by a call to `app.synth()`, and it traverses the construct tree and invokes the `synthesize` method on all constructs. Constructs that implement `synthesize` can participate in synthesis and emit deployment artifacts to the resulting cloud assembly. These constructs include AWS CloudFormation templates, AWS Lambda application bundles, file and Docker image assets, and other deployment artifacts. The section called “Cloud assemblies” (p. 69) describes the output of this phase. In most cases, you won't need to implement the `synthesize` method.

5. Deployment

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

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

- The AWS CDK app can't respond to events that happen during deployment, such as a resource being created or the whole deployment finishing. To run code during the deployment phase, you have to inject it into the AWS CloudFormation template as a custom resource (p. 157). For more information about adding a custom resource to your app, see the AWS CloudFormation module, or the custom-resource example.
- The AWS CDK app might have to work with values that can't be known at the time it runs. For example, if the AWS CDK app defines an Amazon S3 bucket with an automatically generated name, and you retrieve the `bucket.bucketName` (Python: `bucket_name`) attribute, that value is not the name of the deployed bucket. Instead, you get a `Token` value. To determine whether a particular value is available, call `cdk.isToken(value)` (Python: `is_token`). See the section called “Tokens” (p. 105) for details.

---

### Cloud assemblies

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

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

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

To work with the CDK CLI, you need to let it know how to execute an AWS CDK app.

```bash
cdk --app executable cdk-command
```

The `--app` option instructs the CLI to run your AWS CDK app, and its contents depend on the programming language you use. Eventually it should be a program that the operating system can run. You can also create the `cdk.json` file and add information to it so that you need to call only `cdk cdk-command`. For example, for JavaScript apps, the `cdk.json` file might look like the following, where `node bin/my-app.js` executes a Node.js program.

TypeScript

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

**JavaScript**

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

**Python**

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

**Java**

{
  "app": "mvn -q exec:java",
}

**C#**

{
  "app": "dotnet run -p src/project-name/project-name.csproj"
}

**Note**

Use the cdk init command to create a language-specific project, with a cdk.json file containing the correct configuration for the programming language you specify.

The cdk-command part of the AWS CDK CLI command represents what you want the AWS CDK to do with the app.

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

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

## Stacks

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

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

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

For example, the following code defines an AWS CDK app with two stacks.
To list all the stacks in an AWS CDK app, run the `cdk ls` command, which for the previous AWS CDK app would have the following output.

```
stack1
stack2
```

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

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

```
stack1
```
This approach is conceptually different from how AWS CloudFormation templates are normally used, where a template can be deployed multiple times and parameterized through AWS CloudFormation parameters. Although AWS CloudFormation parameters can be defined in the AWS CDK, they are generally discouraged because AWS CloudFormation parameters are resolved only during deployment. This means that you cannot determine their value in your code. For example, to conditionally include a resource in your app based on the value of a parameter, you must set up an AWS CloudFormation condition and tag the resource with this condition. Because the AWS CDK takes an approach where concrete templates are resolved at synthesis time, you can use an if statement to check the value to determine whether a resource should be defined or some behavior should be applied.

**Note**

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

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

**TypeScript**

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

interface EnvProps {
  prod: boolean;
}

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

class MyService extends Construct {
  constructor(scope: Construct, id: string, props?: EnvProps) {
    super(scope, id);

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

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

app.synth();
```

**JavaScript**

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

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}
```
class MyService extends Construct {
    constructor(scope, id, props) {
        super(scope, id);

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

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

Python

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

# imagine these stacks declare a bunch of related resources
class ControlPlane(Stack): pass
class DataPlane(Stack): pass
class Monitoring(Stack): pass
class MyService(Construct):
    def __init__(self, scope: Construct, id: str, *, prod=False):
        super().__init__(scope, id)

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

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

Java

```java
package com.myorg;

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

public class MyApp {

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

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

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

static class MyService extends Construct {
    MyService(Construct scope, String id) {
        this(scope, id, false);
    }
    MyService(Construct scope, String id, boolean prod) {
        super(scope, id);
        // we might use the prod argument to change how the service is configured
        new ControlPlane(this, "cp");
        new DataPlane(this, "data");
        new Monitoring(this, "mon");
    }
}

public static void main(final String argv[]) {
    App app = new App();
    new MyService(app, "beta");
    new MyService(app, "prod", true);
    app.synth();
}

C#

using Amazon.CDK;

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

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

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

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

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

```
$ cdk ls
betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

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

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

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

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

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

**C#**
```
new MyStack(this, "not:a:stack:name", new StackProps
{ StackName = "this-is-stack-name"
});
```

### Stack API

The Stack object provides a rich API, including the following:
• **Stack.of(constuct)** – A static method that returns the **Stack** in which a construct is defined. This is useful if you need to interact with a stack from within a reusable construct. The call fails if a stack cannot be found in scope.

• **stack.stackName** (Python: `stack_name`) – Returns the physical name of the stack. As mentioned previously, all AWS CDK stacks have a physical name that the AWS CDK can resolve during synthesis.

• **stack.region** and **stack.account** – Return the AWS Region and account, respectively, into which this stack will be deployed. These properties return either the account or Region explicitly specified when the stack was defined, or a string-encoded token that resolves to the AWS CloudFormation pseudo-parameters for account and Region to indicate that this stack is environment agnostic. See the section called “Environments” (p. 77) for information about how environments are determined for stacks.

• **stack.addDependency(stack)** (Python: `stack.add_dependency(stack)`) – Can be used to explicitly define dependency order between two stacks. This order is respected by the **cdk deploy** command when deploying multiple stacks at once.

• **stack.tags** – Returns a **TagManager** that you can use to add or remove stack-level tags. This tag manager tags all resources within the stack, and also tags the stack itself when it’s created through AWS CloudFormation.

• **stack.partition**, **stack.urlSuffix** (Python: `url_suffix`), **stack.stackId** (Python: `stack_id`), and **stack.notificationArn** (Python: `notification_arn`) – Return tokens that resolve to the respective AWS CloudFormation pseudo-parameters, such as `{ "Ref": "AWS::Partition" }`. These tokens are associated with the specific stack object so that the AWS CDK framework can identify cross-stack references.

• **stack.availabilityZones** (Python: `availability_zones`) – Returns the set of Availability Zones available in the environment in which this stack is deployed. For environment-agnostic stacks, this always returns an array with two Availability Zones, but for environment-specific stacks, the AWS CDK queries the environment and returns the exact set of Availability Zones available in the region you specified.

• **stack.parseArn(arn)** and **stack.formatArn(comps)** (Python: `parse_arn`, `format_arn`) – Can be used to work with Amazon Resource Names (ARNs).

• **stack.toJsonString(obj)** (Python: `to_json_string`) – Can be used to format an arbitrary object as a JSON string that can be embedded in an AWS CloudFormation template. The object can include tokens, attributes, and references, which are only resolved during deployment.

• **stack.templateOptions** (Python: `template_options`) – Enables you to specify AWS CloudFormation template options, such as Transform, Description, and Metadata, for your stack.

---

**Nested stacks**

The **NestedStack** construct offers a way around the AWS CloudFormation 200-resource limit for stacks. A nested stack counts as only one resource in the stack that contains it, but can itself contain up to 200 resources, including additional nested stacks.

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

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

References between parent stacks and nested stacks are automatically translated to stack parameters and outputs in the generated AWS CloudFormation templates, as with any **cross-stack reference** (p. 85).
Warning
Changes in security posture are not displayed before deployment for nested stacks. This information is displayed only for top-level stacks.

Environments

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

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

Note
In an environment-agnostic stack, any constructs that use availability zones will see two of them. This allows the stack to be deployed to almost any region, since nearly all regions have at least two availability zones. The only exception is Osaka (ap-northeast-3), which has one.

When using cdk deploy to deploy environment-agnostic stacks, the AWS CDK CLI uses the specified AWS CLI profile (or the default profile, if none is specified) to determine where to deploy. The AWS CDK CLI follows a protocol similar to the AWS CLI to determine which AWS credentials to use when performing operations against your AWS account. See the section called "AWS CDK Toolkit" (p. 253) for details.

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

TypeScript

```typescript
const envEU = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

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

JavaScript

```javascript
const envEU = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

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

Python

```python
env_EU = core.Environment(account="8373873873", region="eu-west-1")
env_USA = core.Environment(account="2383838383", region="us-west-2")

MyFirstStack(app, "first-stack-us", env=env_USA)
MyFirstStack(app, "first-stack-eu", env=env_EU)
```

Java

```java
public class MyApp {

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

class App {
    public static void main(final String argv[]) {
        Environment envEU = makeEnv("8373873873", "eu-west-1");
        Environment envUSA = makeEnv("2383838383", "us-west-2");
        new MyFirstStack(app, "first-stack-us", StackProps.builder()
            .env(envUSA).build());
        new MyFirstStack(app, "first-stack-eu", StackProps.builder()
            .env(envEU).build());
        app.synth();
    }
}

C#

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

var envEU = makeEnv(account: "8373873873", region: "eu-west-1");
var envUSA = makeEnv(account: "2383838383", region: "us-west-2");
new MyFirstStack(app, "first-stack-us", new StackProps { Env=envUSA });
new MyFirstStack(app, "first-stack-eu", new StackProps { Env=envEU });

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

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

TypeScript

Access environment variables via Node’s process object.

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

new MyDevStack(app, 'dev', {
    env: {
}})
Environments

JavaScript

Access environment variables via Node's process object.

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

Python

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

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

Java

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

```java
public class MyApp {

  // Helper method to build an environment
  static Environment makeEnv(String account, String region) {
    account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") : account;
    region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

    return Environment.builder()
        .account(account)
        .region(region)
        .build();
  }

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

    Environment envEU = makeEnv(null, null);
    Environment envUSA = makeEnv(null, null);
    new MyDevStack(app, "first-stack-us", StackProps.builder()
        .env(envUSA).build());
    new MyDevStack(app, "first-stack-eu", StackProps.builder()
        .env(envEU).build());

    app.synth();
  }
}
```

C#

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

```csharp
Amazon.CDK.Environment makeEnv(string account=null, string region=null) {
```
aws-cloud-development-kit-guide

Environments

```typescript
return new Amazon.CDK.Environment
{
  Account = account ??
System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
Region = region ??
System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
};
}
new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

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

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

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

**TypeScript**

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

**JavaScript**

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

**Python**

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

**Java**

```java
public class MyApp {
```
// Helper method to build an environment
static Environment makeEnv(String account, String region) {
    account = (account == null) ? System.getenv("CDK_DEPLOY_ACCOUNT") : account;
    region = (region == null) ? System.getenv("CDK_DEPLOY_REGION") : region;
    account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") : account;
    region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

    return Environment.builder()
        .account(account)
        .region(region)
        .build();
}

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

    Environment envEU = makeEnv(null, null);
    Environment envUSA = makeEnv(null, null);

    new MyDevStack(app, "first-stack-us", StackProps.builder()
        .env(envUSA).build());
    new MyDevStack(app, "first-stack-eu", StackProps.builder()
        .env(envEU).build());

    app.synth();
}

C#

Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_ACCOUNT") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_REGION") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });

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

Mac OS X/Linux

#!/usr/bin/env bash
if [[ $# -ge 2 ]]; then
    export CDK_DEPLOY_ACCOUNT=$1
    export CDK_DEPLOY_REGION=$2
    shift; shift
    npx cdk deploy "$@
    exit $?
else
    echo 1>&2 "Provide AWS account and region as first two args."
Save the script as `cdk-deploy-to.sh`, then execute `chmod +x cdk-deploy-to.sh` to make it executable.

**Windows**

```bash
@findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass %~dpn0.ps1 %*
@exit /B %ERRORLEVEL%
if ($args.length -ge 2) {
    $env:CDK_DEPLOY_ACCOUNT, $args = $args
    $env:CDK_DEPLOY_REGION, $args = $args
    npx cdk deploy $args
    exit $lastExitCode
} else {
    [console]::error.writeline("Provide AWS account and region as first two args.")
    [console]::error.writeline("Additional args are passed through to cdk deploy.")
    exit 1
}
```

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

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

**Mac OS X/Linux**

```bash
#!/usr/bin/env bash
# cdk-deploy-to-test.sh
./cdk-deploy-to.sh 123457689 us-east-1 "@"
```

**Windows**

```bash
@echo off
rem cdk-deploy-to-test.bat
cdk-deploy-to 135792469 us-east-1 %*
```

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

**Mac OS X/Linux**

```bash
#!/usr/bin/env bash
# cdk-deploy-to-prod.sh
./cdk-deploy-to.sh 135792468 us-west-1 "@" || exit
./cdk-deploy-to.sh 246813579 eu-west-1 "@"
```

**Windows**

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

## Resources

As described in the section called “Constructs” (p. 50), the AWS CDK provides a rich class library of constructs, called *AWS constructs*, that represent all AWS resources. This section describes some common patterns and best practices for how to use these constructs.

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

**TypeScript**

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

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

**JavaScript**

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

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

**Python**

```python
import aws_cdk.aws_sqs as sqs

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

**Java**

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

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

**C#**

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

new Queue(this, "MyQueue", new QueueProps
{
    Encryption = QueueEncryption.KMS_MANAGED
});
```
Some configuration props are optional, and in many cases have default values. In some cases, all props are optional, and the last argument can be omitted entirely.

### Resource attributes

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

**TypeScript**

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

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

**JavaScript**

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

**Python**

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

**Java**

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

**C#**

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

See the section called "Tokens" (p. 105) for information about how the AWS CDK encodes deploy-time attributes as strings.

### Referencing resources

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

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

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

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

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

**TypeScript**

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

**JavaScript**

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

**Python**

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

**Java**

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

**C#**

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

---

**Accessing resources in a different stack**

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

**TypeScript**

```typescript
const prod = { account: '123456789012', region: 'us-east-1' };`
const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

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

JavaScript

const prod = { account: '123456789012', region: 'us-east-1' };  
const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

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

Python

prod = core.Environment(account="123456789012", region="us-east-1")
stack1 = StackThatProvidesABucket(app, "Stack1", env=prod)

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

Java

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

App app = new App();

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

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

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

C#

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

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

var stack1 = new StackThatProvidesABucket(app, "Stack1", new StackProps { Env = prod });
If the AWS CDK determines that the resource is in the same account and Region, but in a different stack, it automatically synthesizes AWS CloudFormation exports in the producing stack and an Fn::ImportValue in the consuming stack to transfer that information from one stack to the other.

Physical names

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

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

You can specify a physical name when creating constructs that represent resources by using the property <resourceType>Name. The following example creates an Amazon S3 bucket with the physical name my-bucket-name.

TypeScript

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

JavaScript

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

Python

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

Java

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

C#

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

Assigning physical names to resources has some disadvantages in AWS CloudFormation. Most importantly, any changes to deployed resources that require a resource replacement, such as changes to a resource's properties that are immutable after creation, will fail if a resource has a physical name assigned. If you end up in a state like that, the only solution is to delete the AWS CloudFormation stack, then deploy the AWS CDK app again. See the AWS CloudFormation documentation for details.
In some cases, such as when creating an AWS CDK app with cross-environment references, physical names are required for the AWS CDK to function correctly. In those cases, if you don’t want to bother with coming up with a physical name yourself, you can let the AWS CDK name it for you by using the special value `PhysicalName.GENERATE_IF_NEEDED`, as follows.

**TypeScript**

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

**JavaScript**

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

**Python**

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

**Java**

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

**C#**

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

### Passing unique identifiers

Whenever possible, you should pass resources by reference, as described in the previous section. However, there are cases where you have no other choice but to refer to a resource by one of its attributes. For example, when you are using the low-level AWS CloudFormation resources, or need to expose resources to the runtime components of an AWS CDK application, such as when referring to Lambda functions through environment variables.

These identifiers are available as attributes on the resources, such as the following.

**TypeScript**

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

**JavaScript**

```javascript
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```
Passing unique identifiers

Python

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

Java

The Java AWS CDK binding uses getter methods for attributes.

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

C#

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

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

TypeScript

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

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

JavaScript

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

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

Python

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

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

Java

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

Function.Builder.create(this, "MyLambda")
  .environment(new HashMap<String, String>() {{
    put("BUCKET_NAME", bucket.getBucketName());
  }});```
Importing existing external resources

Sometimes you already have a resource in your AWS account and want to use it in your AWS CDK app, for example, a resource that was defined through the console, the AWS SDK, directly with AWS CloudFormation, or in a different AWS CDK application. You can turn the resource’s ARN (or another identifying attribute, or group of attributes) into an AWS CDK object in the current stack by calling a static factory method on the resource's class.

The following example shows how to define a bucket based on an existing bucket with the ARN `arn:aws:s3:::my-bucket-name`, and a Amazon Virtual Private Cloud based on an existing VPC having a specific ID.

TypeScript

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

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

// Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
    vpcId: 'vpc-1234567890abcde',
});
```

JavaScript

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

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

// Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
    vpcId: 'vpc-1234567890abcde' });
```

Python

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

# Construct a resource (bucket) by its full ARN (can be cross account)
```
s3.Bucket.from_arn(self, "MyBucket", "arn:aws:s3:::my-bucket-name")

# Construct a resource by giving attribute(s) (complex resources)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")

Java

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

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

// Construct a resource by giving attribute(s) (complex resources)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
    .vpcId("vpc-1234567890abcdef").build());

C#

// Construct a resource (bucket) just by its name (must be same account)
Bucket.FromBucketName(this, "MyBucket", "my-bucket-name");

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

// Construct a resource by giving attribute(s) (complex resources)
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
    { VpcId = "vpc-1234567890abcdef" });

Because the ec2.Vpc construct is complex, composed of many AWS resources, such as the VPC itself, subnets, security groups, and routing tables, it can be difficult to import those resources using attributes. To address this, the VPC construct contains a fromLookup method (Python: from_lookup) that uses a context method (p. 143) to resolve all the required attributes at synthesis time, and cache the values for future use in cdk.context.json.

You must provide attributes sufficient to uniquely identify a VPC in your AWS account. For example, there can only ever be one default VPC, so specifying that you want to import the VPC marked as the default is sufficient.

TypeScript

e2.Vpc.fromLookup(this, 'DefaultVpc', {
    isDefault: true
});

JavaScript

e2.Vpc.fromLookup(this, 'DefaultVpc', {
    isDefault: true
});

Python

e2.Vpc.from_lookup(self, "DefaultVpc", is_default=True)
Importing existing external resources

Java

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

C#

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

You can use the `tags` property to query by tag. Tags may be added to the VPC at the time of its creation using AWS CloudFormation or the AWS CDK, and they may be edited at any time after creation using the AWS Management Console, the AWS CLI, or an AWS SDK. In addition to any tags you have added yourself, the AWS CDK automatically adds the following tags to all VPCs it creates.

- **Name** – The name of the VPC.
- **aws-cdk:subnet-name** – The name of the subnet.
- **aws-cdk:subnet-type** – The type of the subnet: Public, Private, or Isolated.

TypeScript

```typescript
e2.Vpc.fromLookup(this, 'PublicVpc',
    { tags: { 'aws-cdk:subnet-type': "Public" }});
```

JavaScript

```javascript
e2.Vpc.fromLookup(this, 'PublicVpc',
    { tags: { 'aws-cdk:subnet-type': "Public" }});
```

Python

```python
e2.Vpc.from_lookup(self, "PublicVpc",
    tags={"aws-cdk:subnet-type": "Public"})
```

Java

```java
Vpc.fromLookup(this, "PublicVpc", VpcLookupOptions.builder()
    .tags(new HashMap<String, String> {{ put("aws-cdk:subnet-type", "Public"); }})
    .build());
```

C#

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

Note that `Vpc.fromLookup()` works only in stacks that are defined with an explicit `account` and `region` in their `env` property. If the AWS CDK attempts to look up an Amazon VPC from an environment-agnostic stack (p. 75), the CLI does not know which environment to query to find the VPC.

Although you can use an imported resource anywhere, you cannot modify the imported resource. For example, calling `addToResourcePolicy` (Python: `add_to_resource_policy`) on an imported `s3.Bucket` does nothing.
Permission grants

AWS constructs make least-privilege permissions easy to achieve by offering simple, intent-based APIs to express permission requirements. Many AWS constructs offer grant methods that enable you to easily grant an entity, such as an IAM role or a user, permission to work with the resource without having to manually craft one or more IAM permission statements.

The following example creates the permissions to allow a Lambda function's execution role to read and write objects to a particular Amazon S3 bucket. If the Amazon S3 bucket is encrypted using an AWS KMS key, this method also grants the Lambda function's execution role permissions to decrypt using this key.

TypeScript

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

JavaScript

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

Python

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

Java

```java
if (bucket.grantReadWrite(func).getSuccess()) {
    // ...
}
```

C#

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

The grant methods return an `iam.Grant` object. Use the `success` attribute of the `Grant` object to determine whether the grant was effectively applied (for example, it may not have been applied on imported resources (p. 84)). You can also use the `assertSuccess` (Python: `assert_success`) method of the `Grant` object to enforce that the grant was successfully applied.

If a specific grant method isn't available for the particular use case, you can use a generic grant method to define a new grant with a specified list of actions.

The following example shows how to grant a Lambda function access to the Amazon DynamoDB `CreateBackup` action.

TypeScript

```typescript
table.grant(func, 'dynamodb:CreateBackup');
```
Many resources, such as Lambda functions, require a role to be assumed when executing code. A configuration property enables you to specify an `iam.IRole`. If no role is specified, the function automatically creates a role specifically for this use. You can then use grant methods on the resources to add statements to the role.

The grant methods are built using lower-level APIs for handling with IAM policies. Policies are modeled as `PolicyDocument` objects. Add statements directly to roles (or a construct's attached role) using the `addToRolePolicy` method (Python: `add_to_role_policy`), or to a resource's policy (such as a `Bucket` policy) using the `addToResourcePolicy` (Python: `add_to_resource_policy`) method.

### Metrics and alarms

Many resources emit CloudWatch metrics that can be used to set up monitoring dashboards and alarms. AWS constructs have metric methods that allow easy access to the metrics without having to look up the correct name to use.

The following example shows how to define an alarm when the `ApproximateNumberOfMessagesNotVisible` of an Amazon SQS queue exceeds 100.

**TypeScript**

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

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

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

**JavaScript**

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

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

const metric = queue.metricApproximateNumberOfMessagesNotVisible(
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5)
); metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100
});
var metric = queue.MetricApproximateNumberOfMessagesNotVisible(new cw.MetricOptions
{
  Label = "Messages Visible (Approx)",
  Period = cdk.Duration.Minutes(5),
  // ...
});
metric.CreateAlarm(this, "TooManyMessagesAlarm", new cw.CreateAlarmOptions
{
  ComparisonOperator = cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  Threshold = 100,
  // ..
});

If there is no method for a particular metric, you can use the general metric method to specify the metric name manually.

Metrics can also be added to CloudWatch dashboards. See CloudWatch.

Network traffic

In many cases, you must enable permissions on a network for an application to work, such as when the compute infrastructure needs to access the persistence layer. Resources that establish or listen for connections expose methods that enable traffic flows, including setting security group rules or network ACLs.

IConnectable resources have a connections property that is the gateway to network traffic rules configuration.

You enable data to flow on a given network path by using allow methods. The following example enables HTTPS connections to the web and incoming connections from the Amazon EC2 Auto Scaling group fleet2.

**TypeScript**

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

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

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

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

**JavaScript**

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

const fleet1 = asg.AutoScalingGroup();

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

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

Python

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

fleet1 = asg.AutoScalingGroup( ... )

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

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

Java

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

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

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

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

C#

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

// Allow surfing the (secure) Web
var fleet1 = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
    { /* ... */ });
    { FromPort = 443, ToPort = 443 }));

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

Certain resources have default ports associated with them, for example, the listener of a load balancer
on the public port, and the ports on which the database engine accepts connections for instances of an
Amazon RDS database. In such cases, you can enforce tight network control without having to manually
specify the port by using the allowDefaultPortFrom and allowToDefaultPort methods (Python:
allow_default_port_from, allow_to_default_port).

The following example shows how to enable connections from any IPV4 address, and a connection from
an Auto Scaling group to access a database.

TypeScript

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

Some resources can act as event sources. Use the `addEventNotification` method (Python: `add_event_notification`) to register an event target to a particular event type emitted by the resource. In addition to this, `addXxxNotification` methods offer a simple way to register a handler for common event types.

The following example shows how to trigger a Lambda function when an object is added to an Amazon S3 bucket.

TypeScript

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

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

JavaScript

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

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

Python

```python
import aws_cdk.aws_s3_notifications as s3_nots

import s3nots

import s3

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

JavaScript

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

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

Resources that maintain persistent data, such as databases and Amazon S3 buckets, have a removal policy that indicates whether to delete persistent objects when the AWS CDK stack that contains them is destroyed. The values specifying the removal policy are available through the RemovalPolicy enumeration in the AWS CDK core module.

Note

Resources besides those that store data persistently may also have a removalPolicy that is used for a different purpose. For example, a Lambda function version uses a removalPolicy attribute to determine whether a given version is retained when a new version is deployed. These have different meanings and defaults compared to the removal policy on an Amazon S3 bucket or DynamoDB table.

<table>
<thead>
<tr>
<th>Value</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemovalPolicy.RETAIN</td>
<td>Keep the contents of the resource when destroying the stack (default). The resource is orphaned from the stack and must be deleted manually. If you attempt to re-deploy the stack while the resource still exists, you will receive an error message due to a name conflict.</td>
</tr>
<tr>
<td>RemovalPolicy.DESTROY</td>
<td>The resource will be destroyed along with the stack.</td>
</tr>
</tbody>
</table>

AWS CloudFormation does not remove Amazon S3 buckets that contain files even if their removal policy is set to DESTROY. Attempting to do so is a AWS CloudFormation error. Delete the files from the bucket before destroying the stack. You can automate this using a custom resource; see the third-party construct auto-delete-bucket for an example.

Following is an example of creating an Amazon S3 bucket with RemovalPolicy.DESTROY.

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

Java

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

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

C#

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

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

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

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

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

JavaScript

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

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

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

module.exports = { CdkTestStack }
```

Python

```python
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

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

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

Java

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

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

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

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

**Note**
The AWS CDK's RemovalPolicy translates to AWS CloudFormation's DeletionPolicy, but the default in AWS CDK is to retain the data, which is the opposite of the AWS CloudFormation default.
If you attempt to create an identifier with the same value within the same scope, the AWS CDK throws an exception.

**Construct IDs**

The most common identifier, `id`, is the identifier passed as the second argument when instantiating a construct object. This identifier, like all identifiers, need only be unique within the scope in which it is created, which is the first argument when instantiating a construct object.

**Note**

The `id` of a stack is also the identifier you use to refer to it in the section called “AWS CDK Toolkit” (p. 253).

Let’s look at an example where we have two constructs with the identifier `MyBucket` in our app. However, since they are defined in different scopes, the first in the scope of the stack with the identifier `Stack1`, and the second in the scope of a stack with the identifier `Stack2`, that doesn’t cause any sort of conflict, and they can co-exist in the same app without any issues.

**TypeScript**

```typescript
import { App, Construct, Stack, StackProps } from '@aws-cdk/core';
import * as s3 from '@aws-cdk/aws-s3';

class MyStack extends Stack {
  constructor(scope: Construct, id: string, props: StackProps = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

**JavaScript**

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

class MyStack extends Stack {
  constructor(scope, id, props = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

**Python**

```python
from aws_cdk.core import App, Construct, Stack, StackProps
from aws_cdk import aws_s3 as s3

class MyStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
```
super().__init__(scope, id, **kwargs)
s3.Bucket(self, "MyBucket")

app = App()
MyStack(app, 'Stack1')
MyStack(app, 'Stack2')

Java

// MyStack.java
package com.myorg;

import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.s3.Bucket;

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

    public MyStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.core.App;

public class Main {
    public static void main(String[] args) {
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}

C#

using core = Amazon.CDK;
using s3 = Amazon.CDK.AWS.S3;

public class MyStack : core.Stack {
    public MyStack(core.App scope, string id, core.IStackProps props) : base(scope, id, props)
    {
        new s3.Bucket(this, "MyBucket");
    }
}

class Program {
    static void Main(string[] args) {
        var app = new core.App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
The constructs in an AWS CDK application form a hierarchy rooted in the `App` class. We refer to the collection of IDs from a given construct, its parent construct, its grandparent, and so on to the root of the construct tree, as a *path*.

The AWS CDK typically displays paths in your templates as a string, with the IDs from the levels separated by slashes, starting at the node just below the root `App` instance, which is usually a stack. For example, the paths of the two Amazon S3 bucket resources in the previous code example are `Stack1/MyBucket` and `Stack2/MyBucket`.

You can access the path of any construct programmatically, as shown in the following example, which gets the path of `myConstruct` (or `my_construct`, as Python developers would write it). Since IDs must be unique within the scope they are created, their paths are always unique within a AWS CDK application.

**TypeScript**

```typescript
const path: string = myConstruct.node.path;
```

**JavaScript**

```javascript
const path = myConstruct.node.path;
```

**Python**

```python
path = my_construct.node.path
```

**Java**

```java
String path = myConstruct.getNode().getPath();
```

**C#**

```csharp
string path = myConstruct.Node.Path;
```

### Unique IDs

Since AWS CloudFormation requires that all logical IDs in a template are unique, the AWS CDK must be able to generate a unique identifier for each construct in an application. 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 programmatically, as shown in the following example, which gets the unique ID of `myConstruct` (or `my_construct` in Python conventions). Since ids must be unique within the scope they are created, their paths are always unique within a AWS CDK application.

**TypeScript**

```typescript
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.

Tokens

Tokens represent values that can only be resolved at a later time in the lifecycle of an app (see the section called “App lifecycle” (p. 68)). For example, the name of an Amazon S3 bucket that you define in your AWS CDK app is only allocated by AWS CloudFormation when you deploy your app. If you print the bucket.bucketName attribute, which is a string, you see it contains something like the following.

```
${TOKEN[Bucket.Name.1234]}
```

This is how the AWS CDK encodes a token whose value is not yet known at construction time, but will become available later. The AWS CDK calls these placeholders tokens. In this case, it's a token encoded as a string.

You can pass this string around as if it was the name of the bucket, such as in the following example, where the bucket name is specified as an environment variable to an AWS Lambda function.
Tokens and token encodings

Tokens are objects that implement the `IResolvable` interface, which contains a single `resolve` method. The AWS CDK calls this method during synthesis to produce the final value for the AWS CloudFormation template. Tokens participate in the synthesis process to produce arbitrary values of any type.
Note
You'll hardly ever work directly with the IResolvable interface. You will most likely only see string-encoded versions of tokens.

Other functions typically only accept arguments of basic types, such as string or number. To use tokens in these cases, you can encode them into one of three types using static methods on the core.Token class.

- `Token.asString` to generate a string encoding (or call `.toString()` on the token object)
- `Token.asList` to generate a list encoding
- `Token.asNumber` to generate a numeric encoding

These take an arbitrary value, which can be an IResolvable, and encode them into a primitive value of the indicated type.

Important
Because any one of the previous types can potentially be an encoded token, be careful when you parse or try to read their contents. For example, if you attempt to parse a string to extract a value from it, and the string is an encoded token, your parsing will fail. Similarly, if you attempt to query the length of an array, or perform math operations with a number, you must first verify that they are not encoded tokens.

To check whether a value has an unresolved token in it, call the `Token.isUnresolved` (Python: `is_unresolved`) method.

The following example validates that a string value, which could be a token, is no more than 10 characters long.

TypeScript
```typescript
if (!Token.isUnresolved(name) && name.length > 10) {
  throw new Error(`Maximum length for name is 10 characters`);
}
```

JavaScript
```javascript
if ( !Token.isUnresolved(name) && name.length > 10) {
  throw ( new Error(`Maximum length for name is 10 characters`));
}
```

Python
```python
if not Token.is_unresolved(name) and len(name) > 10:
    raise ValueError("Maximum length for name is 10 characters")
```

Java
```java
if (!Token.isUnresolved(name) && name.length() > 10)
    throw new IllegalArgumentException("Maximum length for name is 10 characters");
```

C#
```csharp
if (!Token.IsUnresolved(name) && name.Length > 10)
    throw new ArgumentException("Maximum length for name is 10 characters");
```
If name is a token, validation isn't performed, and an error could still occur in a later stage in the lifecycle, such as during deployment.

**Note**
You can use token encodings to escape the type system. For example, you could string-encode a token that produces a number value at synthesis time. If you use these functions, it's your responsibility to ensure that your template resolves to a usable state after synthesis.

## String-encoded tokens

String-encoded tokens look like the following.

```text
${TOKEN[Bucket.Name.1234]}
```

They can be passed around like regular strings, and can be concatenated, as shown in the following example.

**TypeScript**

```typescript
const functionName = bucket.bucketName + 'Function';
```

**JavaScript**

```javascript
const functionName = bucket.bucketName + 'Function';
```

**Python**

```python
function_name = bucket.bucket_name + "Function"
```

**Java**

```java
String functionName = bucket.getBucketName().concat("Function");
```

**C#**

```csharp
string functionName = bucket.BucketName + "Function";
```

You can also use string interpolation, if your language supports it, as shown in the following example.

**TypeScript**

```typescript
const functionName = `${bucket.bucketName}Function`; 
```

**JavaScript**

```javascript
const functionName = `${bucket.bucketName}Function`;
```

**Python**

```python
function_name = f"{bucket.bucket_name}Function"
```
List-encoded tokens

List-encoded tokens look like the following

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

The only safe thing to do with these lists is pass them directly to other constructs. Tokens in string list form cannot be concatenated, nor can an element be taken from the token. The only safe way to manipulate them is by using AWS CloudFormation intrinsic functions like `Fn.select`.

Number-encoded tokens

Number-encoded tokens are a set of tiny negative floating-point numbers that look like the following.

```
-1.8881545897087626e+289
```

As with list tokens, you cannot modify the number value, as doing so is likely to break the number token. The only allowed operation is to pass the value around to another construct.

Lazy values

In addition to representing deploy-time values, such as AWS CloudFormation parameters (p. 112), tokens are also commonly used to represent synthesis-time lazy values. These are values for which the final value will be determined before synthesis has completed, just not at the point where the value is constructed. Use tokens to pass a literal string or number value to another construct, while the actual value at synthesis time may depend on some calculation that has yet to occur.

You can construct tokens representing synth-time lazy values using static methods on the `Lazy` class, such as `Lazy.stringValue` (Python: `Lazy.string_value`) and `Lazy.numberValue` (Python: `Lazy.number_value`). These methods accept an object whose `producer` property is a function that accepts a context argument and returns the final value when called.

The following example creates an Auto Scaling group whose capacity is determined after its creation.

TypeScript

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
})
```
Lazy values

```javascript
let actualValue;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return (actualValue);
    }
  })
});

// At some later point
actualValue = 10;
```

```python
class Producer:
  def __init__(self, func):
    self.produce = func

actual_value = None

AutoScalingGroup(self, "Group",
    desired_capacity=Lazy.number_value(Producer(lambda context: actual_value))
)

# At some later point
actual_value = 10
```

```java
double actualValue = 0;

class ProduceActualValue implements INumberProducer {
  @Override
  public Number produce(IResolveContext context) {
    return actualValue;
  }
}

AutoScalingGroup.Builder.create(this, "Group")
  .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point
actualValue = 10;
```

```csharp
public class NumberProducer : INumberProducer
{
  Func<Double> function;

  public NumberProducer(Func<Double> function)
  {
    this.function = function;
  
```
Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method `stack.toJsonString`, as shown in the following example.

**TypeScript**

```typescript
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

**JavaScript**

```javascript
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

**Python**

```python
stack = Stack.of(self)
string = stack.to_json_string(dict(value=bucket.bucket_name))
```

**Java**

```java
Stack stack = Stack.of(this);
String stringVal = stack.toJsonString(new HashMap<String, String>() {
  put("value", bucket.getBucketName());
});
```

**C#**

```csharp
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{["value"] = bucket.BucketName
});
```
Parameters

AWS CloudFormation templates can contain parameters—custom values that are supplied at deployment time and incorporated into the template. Since the AWS CDK synthesizes AWS CloudFormation templates, it too offers support for deployment-time parameters.

Using the AWS CDK, you can both define parameters, which can then be used in the properties of constructs you create, and you can also deploy stacks containing parameters.

When deploying the AWS CloudFormation template using the AWS CDK Toolkit, you provide the parameter values on the command line. If you deploy the template through the AWS CloudFormation console, you are prompted for the parameter values.

In general, we recommend against using AWS CloudFormation parameters with the AWS CDK. Unlike context values (p. 142) or environment variables, the usual way to pass values into your AWS CDK apps without hard-coding them, parameter values are not available at synthesis time, and thus cannot be easily used in other parts of your AWS CDK app, particularly for control flow.

Note
To do control flow with parameters, you can use CfnCondition constructs, although this is awkward compared to native if statements.

Using parameters requires you to be mindful of how the code you’re writing behaves at deployment time, as well as at synthesis time. This makes it harder to understand and reason about your AWS CDK application, in many cases for little benefit.

It is better, again in general, to have your CDK app accept any necessary information from the user and use it directly to declare constructs in your CDK app. An ideal AWS CDK-generated AWS CloudFormation template is concrete, with no values remaining to be specified at deployment time.

There are, however, use cases to which AWS CloudFormation parameters are uniquely suited. If you have separate teams defining and deploying infrastructure, for example, you can use parameters to make the generated templates more widely useful. Additionally, the AWS CDK’s support for AWS CloudFormation parameters lets you use the AWS CDK with AWS services that use AWS CloudFormation templates (such as AWS Service Catalog), which use parameters to configure the template being deployed.

Defining parameters

Use the CfnParameter class to define a parameter. You’ll want to specify at least a type and a description for most parameters, though both are technically optional. The description appears when the user is prompted to enter the parameter’s value in the AWS CloudFormation console.

Note
We recommend defining parameters at the stack level to ensure that their logical ID does not change when you refactor your code.

TypeScript

```typescript
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be stored.”
});
```

JavaScript

```javascript
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
});
```
Using parameters

A `CfnParameter` instance exposes its value to your AWS CDK app via a token (p. 105). Like all tokens, the parameter’s token is resolved at synthesis time, but it resolves to a reference to the parameter defined in the AWS CloudFormation template, which will be resolved at deploy time, rather than to a concrete value.

You can retrieve the token as an instance of the `Token` class, or in string, string list, or numeric encoding, depending on the type of value required by the class or method you want to use the parameter with.

### TypeScript

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Token class instance</td>
</tr>
<tr>
<td>valueAsList</td>
<td>The token represented as a string list</td>
</tr>
<tr>
<td>valueAsNumber</td>
<td>The token represented as a number</td>
</tr>
<tr>
<td>valueAsString</td>
<td>The token represented as a string</td>
</tr>
</tbody>
</table>

### JavaScript

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Token class instance</td>
</tr>
<tr>
<td>valueAsList</td>
<td>The token represented as a string list</td>
</tr>
</tbody>
</table>

description: "The name of the Amazon S3 bucket where uploaded files will be stored.");

Java

```java
CfnParameter uploadBucketName = CfnParameter.Builder.create(this, "uploadBucketName")
    .type("String")
    .description("The name of the Amazon S3 bucket where uploaded files will be stored")
    .build();
```

C#

```csharp
var uploadBucketName = new CfnParameter(this, "uploadBucketName", new CfnParameterProps
{    Type = "String",
    Description = "The name of the Amazon S3 bucket where uploaded files will be stored"
});
```
Using parameters

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueAsNumber</td>
<td>The token represented as a number</td>
</tr>
<tr>
<td>valueAsString</td>
<td>The token represented as a string</td>
</tr>
</tbody>
</table>

Python

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>Token class instance</td>
</tr>
<tr>
<td>value_as_list</td>
<td>The token represented as a string list</td>
</tr>
<tr>
<td>value_as_number</td>
<td>The token represented as a number</td>
</tr>
<tr>
<td>value_as_string</td>
<td>The token represented as a string</td>
</tr>
</tbody>
</table>

Java

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValue()</td>
<td>Token class instance</td>
</tr>
<tr>
<td>getValueAsString()</td>
<td>The token represented as a string</td>
</tr>
<tr>
<td>getValueAsNumber()</td>
<td>The token represented as a number</td>
</tr>
<tr>
<td>getValueAsList()</td>
<td>The token represented as a string list</td>
</tr>
</tbody>
</table>

C#

<table>
<thead>
<tr>
<th>Property</th>
<th>kind of value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Token class instance</td>
</tr>
<tr>
<td>ValueAsList</td>
<td>The token represented as a string list</td>
</tr>
<tr>
<td>ValueAsNumber</td>
<td>The token represented as a number</td>
</tr>
<tr>
<td>ValueAsString</td>
<td>The token represented as a string</td>
</tr>
</tbody>
</table>

For example, to use a parameter in a Bucket definition:

TypeScript

```typescript
const bucket = new Bucket(this, "myBucket", {
    bucketName: uploadBucketName.valueAsString
});
```

JavaScript

```javascript
const bucket = new Bucket(this, "myBucket", {
    bucketName: uploadBucketName.valueAsString
});
```
Deploying with parameters

A generated template containing parameters can be deployed in the usual way through the AWS CloudFormation console; you are prompted for the values of each parameter.

The AWS CDK Toolkit (cdk command-line tool) also supports specifying parameters at deployment. You may provide these on the command line following the --parameters flag. You might deploy a stack that uses the uploadBucketName parameter like this.

cdk deploy MyStack --parameters uploadBucketName=UploadBucket

To define multiple parameters, use multiple --parameters flags.

cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters downloadBucketName=DownBucket

If you are deploying multiple stacks, you can specify a different value of each parameter for each stack by prefixing the name of the parameter with the stack name and a colon.

cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --parameters YourStack:uploadBucketName=UpBucket

By default, the AWS CDK retains values of parameters from previous deployments and uses them in subsequent deployments if they are not specified explicitly. Use the --no-previous-parameters flag to require all parameters to be specified.

Tagging

The Tag class includes two methods that you can use to create and delete tags:

- Tag.add() applies a new tag to a construct and all of its children.
- Tag.remove() removes a tag from a construct and any of its children, including tags a child construct may have applied to itself.
Tagging is implemented using the section called "Aspects" (p. 148). Aspects are a way to apply an operation (such as tagging) to all constructs in a given scope.

Let's look at a couple of examples. The following example applies the tag key with the value value to a construct.

TypeScript

```typescript
Tag.add(myConstruct, 'key', 'value');
```

JavaScript

```javascript
Tag.add(myConstruct, 'key', 'value');
```

Python

```python
Tag.add(my_construct, "key", "value")
```

Java

```java
Tag.add(myConstruct, "key", "value");
```

C#

```csharp
Tag.Add(myConstruct, "key", "value");
```

The following example deletes the tag key from a construct.

TypeScript

```typescript
Tag.remove(my_construct, 'key');
```

JavaScript

```javascript
Tag.remove(my_construct, 'key');
```

Python

```python
Tag.remove(my_construct, "key")
```

Java

```java
Tag.remove(myConstruct, "key");
```

C#

```csharp
Tag.Remove(myConstruct, "key");
```

The AWS CDK applies and removes tags recursively. If there are conflicts, the tagging operation with the highest priority wins. If the priorities are the same, the tagging operation closest to the bottom of the construct tree wins. By default, applying a tag has a priority of 100 and removing a tag has a
Tag.add()

Tag.add() supports properties that fine-tune how tags are applied to resources. All properties are optional.

The following example applies the tag `tagname` with the value `value` and priority 100 to resources of type `AWS::Xxx::Yyy` in the construct, but not to instances launched in an Amazon EC2 Auto Scaling group or to resources of type `AWS::Xxx::Zzz`. (These are placeholders for two arbitrary but different AWS CloudFormation resource types.)

TypeScript

```typescript
Tag.add(myConstruct, 'tagname', 'value', {
    applyToLaunchedInstances: false,
    includeResourceTypes: ['AWS::Xxx::Yyy'],
    excludeResourceTypes: ['AWS::Xxx::Zzz'],
    priority: 100,
});
```

JavaScript

```javascript
Tag.add(myConstruct, 'tagname', 'value', {
    applyToLaunchedInstances: false,
    includeResourceTypes: ['AWS::Xxx::Yyy'],
    excludeResourceTypes: ['AWS::Xxx::Zzz'],
    priority: 100
});
```

Python

```python
Tag.add(my_construct, "tagname", "value", priority=100)
```

Java

```java
Tag.add(myConstruct, "tagname", "value", TagProps.builder()
    .applyToLaunchedInstances(false)
    .includeResourceTypes("AWS::Xxx::Yyy")
    .excludeResourceTypes("AWS::Xxx::Zzz")
    .priority(100)
    .build());
```

C#

```csharp
Tag.Add(myConstruct, "tagname", "value", new TagProps { Priority = 100 });
```
These properties have the following meanings.

applyToLaunchedInstances (Python: apply_to_launched_instances)

By default, tags are applied to instances launched in an Auto Scaling group. Set this property to `false` to not apply tags to instances launched in an Auto Scaling group.

includeResourceTypes/excludeResourceTypes (Python: include_resource_types, exclude_resource_types)

Use these to apply tags only to a subset of resources, based on AWS CloudFormation resource types. By default, the tag is applied to all resources in the construct subtree, but this can be changed by including or excluding certain resource types. Exclude takes precedence over include, if both are specified.

priority

Use this to set the priority of this operation with respect to other `Tag.add()` and `Tag.remove()` operations. Higher values take precedence over lower values. The default is 100.

Tag.remove()

Tag.remove() supports properties to fine-tune how tags are removed from resources. All properties are optional.

The following example removes the tag `tagname` with priority 200 from resources of type `AWS::Xxx::Yyy` in the construct, but not from resources of type `AWS::Xxx::Zzz`.

TypeScript

```typescript
Tag.remove(myConstruct, 'tagname', {
  priority: 200
});
```
These properties have the following meanings.

**includeResourceTypes/excludeResourceTypes**

(Java: includeResourceTypes/excludeResourceTypes) Use these properties to remove tags only from subset of resources based on AWS CloudFormation resource types. By default, the tag is removed from all resources in the construct subtree, but this can be changed by including or excluding certain resource types. Exclude takes precedence over include, if both are specified.

**priority**

Use this property to specify the priority of this operation with respect to other `Tag.add()` and `Tag.remove()` operations. Higher values take precedence over lower values. The default is 200.

**Example**

The following example adds the tag key **StackType** with value **TheBest** to any resource created within the Stack named **MarketingSystem**. Then it removes it again from all resources except Amazon EC2 VPC subnets. The result is that only the subnets have the tag applied.
TypeScript

```typescript
import { App, Stack, Tag } from '@aws-cdk/core';

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tag.add(theBestStack, 'StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tag.remove(theBestStack, 'StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

JavaScript

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

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tag.add(theBestStack, 'StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tag.remove(theBestStack, 'StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

Python

```python
from aws_cdk.core import App, Stack, Tag

app = App();
the_best_stack = Stack(app, 'MarketingSystem')

# Add a tag to all constructs in the stack
Tag.add(the_best_stack, "StackType", "TheBest")

# Remove the tag from all resources except subnet resources
Tag.remove(the_best_stack, "StackType",
    exclude_resource_types=['AWS::EC2::Subnet'])
```

Java

```java
import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Tag;

// Add a tag to all constructs in the stack
Tag.add(theBestStack, "StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tag.remove(theBestStack, "StackType", TagProps.builder()
    .excludeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
    .build());
```

C#

```csharp
using Amazon.CDK;
```
var app = new App();
var theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tag.Add(theBestStack, "StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tag.Remove(theBestStack, "StackType", new TagProps
{
  ExcludeResourceTypes = ["AWS::EC2::Subnet"]
});

The following code achieves the same result. Consider which approach (inclusion or exclusion) makes your intent clearer.

**TypeScript**

```typescript
Tag.add(theBestStack, 'StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

**JavaScript**

```javascript
Tag.add(theBestStack, 'StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

**Python**

```python
Tag.add(the_best_stack, "StackType", "TheBest",
    include_resource_types=["AWS::EC2::Subnet"])
```

**Java**

```java
Tag.add(theBestStack, "StackType", "TheBest", TagProps.builder()
  .includeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
  .build());
```

**C#**

```csharp
Tag.Add(theBestStack, "StackType", "TheBest", new TagProps
  { IncludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

### Assets

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

You typically reference assets through APIs that are exposed by specific AWS constructs. For example, when you define a `lambda.Function` construct, the `code` property lets you pass an `asset` (directory). Function uses assets to bundle the contents of the directory and use it for the function's code. Similarly, `ecs.ContainerImage.fromAsset` uses a Docker image built from a local directory when defining an Amazon ECS task definition.
Assets in detail

When you refer to an asset in your app, the cloud assembly (p. 69) synthesized from your application includes metadata information with instructions for the AWS CDK CLI on where to find the asset on the local disk, and what type of bundling to perform based on the type of asset, such as a directory to compress (zip) or a Docker image to build.

The AWS CDK generates a source hash for assets, which can be used at construction time to determine whether the contents of an asset have changed.

By default, the AWS CDK creates a copy of the asset in the cloud assembly directory, which defaults to cdk.out, under the source hash. This is so that the cloud assembly is self-contained and moved over to a different host for deployment. See the section called “Cloud assemblies” (p. 69) for details.

The AWS CDK also synthesizes AWS CloudFormation parameters that the AWS CDK CLI specifies during deployment. The AWS CDK uses those parameters to refer to the deploy-time values of the asset.

When the AWS CDK deploys an app that references assets (either directly by the app code or through a library), the AWS CDK CLI first prepares and publishes them to Amazon S3 or Amazon ECR, and only then deploys the stack. The AWS CDK specifies the locations of the published assets as AWS CloudFormation parameters to the relevant stacks, and uses that information to enable referencing these locations within an AWS CDK app.

This section describes the low-level APIs available in the framework.

Asset types

The AWS CDK supports the following types of assets:

Amazon S3 Assets

These are local files and directories that the AWS CDK uploads to Amazon S3.

Docker Image

These are Docker images that the AWS CDK uploads to Amazon ECR.

These asset types are explained in the following sections.

Amazon S3 assets

You can define local files and directories as assets, and the AWS CDK packages and uploads them to Amazon S3 through the aws-s3-assets module.

The following example defines a local directory asset and a file asset.

TypeScript

```typescript
import { Asset } from '@aws-cdk/aws-s3-assets';

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```
Asset types

JavaScript

```javascript
const { Asset } = require('@aws-cdk/aws-s3-assets');

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
    path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
    path: path.join(__dirname, 'file-asset.txt')
});
```

Python

```python
import os.path
dirname = os.path.dirname(__file__)

from aws_cdk.aws_s3_assets import Asset

# Archived and uploaded to Amazon S3 as a .zip file
directory_asset = Asset(self, "SampleZippedDirAsset",
    path=os.path.join(dirname, "sample-asset-directory")
)

# Uploaded to Amazon S3 as-is
file_asset = Asset(self, 'SampleSingleFileAsset",
    path=os.path.join(dirname, 'file-asset.txt')
)
```

Java

```java
import java.io.File;
import software.amazon.awscdk.services.s3.assets.Asset;

// Directory where app was started
File startDir = new File(System.getProperty("user.dir"));

// Archived and uploaded to Amazon S3 as a .zip file
Asset directoryAsset = Asset.Builder.create(this, "SampleZippedDirAsset")
    .path(new File(startDir, "sample-asset-directory").toString()).build();

// Uploaded to Amazon S3 as-is
Asset fileAsset = Asset.Builder.create(this, "SampleSingleFileAsset")
    .path(new File(startDir, "file-asset.txt").toString()).build();
```

C#}

```csharp
using System.IO;
using Amazon.CDK.AWS.S3.Assets;

// Archived and uploaded to Amazon S3 as a .zip file
var directoryAsset = new Asset(this, "SampleZippedDirAsset", new AssetProps {
    Path = Path.Combine(Directory.GetCurrentDirectory(), "sample-asset-directory")
});
```
// Uploaded to Amazon S3 as-is
var fileAsset = new Asset(this, "SampleSingleFileAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "file-asset.txt")
});

In most cases, you don't need to directly use the APIs in the aws-s3-assets module. Modules that support assets, such as aws-lambda, have convenience methods that enable you to use assets. For Lambda functions, the asset property enables you to specify a directory or a .zip file in the local file system.

Lambda function example

A common use case is to create AWS Lambda functions with the handler code, which is the entry point for the function, as an Amazon S3 asset.

The following example uses an Amazon S3 asset to define a Python handler in the local directory handler and creates a Lambda function with the local directory asset as the code property. Below is the Python code for the handler.

def lambda_handler(event, context):
    message = 'Hello World!'
    return {
        'message': message
    }

The code for the main AWS CDK app should look like the following.

TypeScript

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

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

        new lambda.Function(this, 'myLambdaFunction', {
            code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
            runtime: lambda.Runtime.PYTHON_3_6,
            handler: 'index.lambda_handler'
        });
    }
}
```

JavaScript

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

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

        new lambda.Function(this, 'myLambdaFunction', {
            code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
            runtime: lambda.Runtime.PYTHON_3_6,
            handler: 'index.lambda_handler'
        });
    }
}
```
Python

```python
from aws_cdk.core import Stack, Construct
from aws_cdk import aws_lambda as lambda_

import os.path
dirname = os.path.dirname(__file__)

class HelloAssetStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)
        lambda_.Function(self, 'myLambdaFunction',
             code=lambda_.Code.from_asset(os.path.join(dirname, 'handler')),
             runtime=lambda_.Runtime.PYTHON_3_6,
             handler="index.lambda_handler")
```

Java

```java
import java.io.File;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

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

    public HelloAssetStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Function.Builder.create(this, "myLambdaFunction")
            .code(Code.fromAsset(new File(startDir, "handler").toString()))
            .runtime(Runtime.PYTHON_3_6)
            .handler("index.lambda_handler").build();
    }
}
```

C#

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using System.IO;

public class HelloAssetStack : Stack
{
    public HelloAssetStack(Construct scope, string id, StackProps props) : base(scope, id, props)
    {
        new Function(this, "myLambdaFunction", new FunctionProps
```
Asset types

```javascript
{  
    "handler")),  
  Runtime = Runtime.PYTHON_3_6,  
  Handler = "index.lambda_handler"
}
```

The `Function` method uses assets to bundle the contents of the directory and use it for the function’s code.

**Deploy-time attributes example**

Amazon S3 asset types also expose deploy-time attributes (p. 84) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command `cdk synth` displays asset properties as AWS CloudFormation parameters.

The following example uses deploy-time attributes to pass the location of an image asset into a Lambda function as environment variables.

**TypeScript**

```typescript
import { Asset } from '@aws-cdk/aws-s3-assets';
import * as path from 'path';

const imageAsset = new Asset(this, "SampleAsset", {  
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {  
  code: lambda.Code.asset(path.join(__dirname, "handler")),  
  runtime: lambda.Runtime.PYTHON_3_6,  
  handler: "index.lambda_handler",  
  environment: {  
    'S3_BUCKET_NAME': imageAsset.s3BucketName,  
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,  
    'S3_URL': imageAsset.s3Url  
  }  
});
```

**JavaScript**

```javascript
const { Asset } = require('@aws-cdk/aws-s3-assets');
const path = require('path');

const imageAsset = new Asset(this, "SampleAsset", {  
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {  
  code: lambda.Code.asset(path.join(__dirname, "handler")),  
  runtime: lambda.Runtime.PYTHON_3_6,  
  handler: "index.lambda_handler",  
  environment: {  
    'S3_BUCKET_NAME': imageAsset.s3BucketName,  
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,  
    'S3_URL': imageAsset.s3Url  
  }  
});
```
Python

```python
import os.path

from aws_cdk import aws_lambda as lambda_
from aws_cdk.aws_s3_assets import Asset

dirname = os.path.dirname(__file__)

image_asset = Asset(self, "SampleAsset",
    path=os.path.join(dirname, "images/my-image.png"))

lambda_.Function(self, "myLambdaFunction",
    code=lambda_.Code.asset(os.path.join(dirname, "handler")),
    runtime=lambda_.Runtime.PYTHON_3_6,
    handler="index.lambda_handler",
    environment=dict(
        S3_BUCKET_NAME=image_asset.s3_bucket_name,
        S3_OBJECT_KEY=image_asset.s3_object_key,
        S3_URL=image_asset.s3_url))
```

Java

```java
import java.io.File;

import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.assets.Asset;

public class FunctionStack extends Stack {
    public FunctionStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset imageAsset = Asset.Builder.create(this, "SampleAsset")
            .path(new File(startDir, "images/my-image.png").toString()).build()

        Function.Builder.create(this, "myLambdaFunction")
            .code(Code.fromAsset(new File(startDir, "handler").toString()))
            .runtime(Runtime.PYTHON_3_6)
            .handler("index.lambda_handler")
            .environment(new HashMap<String, String>() {
                put("S3_BUCKET_NAME", imageAsset.getS3BucketName());
                put("S3_OBJECT_KEY", imageAsset.getS3ObjectKey());
                put("S3_URL", imageAsset.getS3Url());
            }).build();
    }
}
```

C#

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png"))
```
Asset types

new Function(this, "myLambdaFunction", new FunctionProps
{
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionary<string, string>
    {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_URL"] = imageAsset.S3Url
    }
});

Permissions

If you use Amazon S3 assets directly through the aws-s3-assets module, IAM roles, users, or groups, and need to read assets in runtime, grant those assets IAM permissions through the asset.grantRead method.

The following example grants an IAM group read permissions on a file asset.

TypeScript

```javascript
import { Asset } from '@aws-cdk/aws-s3-assets';
import * as path from 'path';

const asset = new Asset(this, 'MyFile', {
    path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

JavaScript

```javascript
const { Asset } = require('@aws-cdk/aws-s3-assets');
const path = require('path');

const asset = new Asset(this, 'MyFile', {
    path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

Python

```python
from aws_cdk.aws_s3_assets import Asset
from aws_cdk import aws_iam as iam
import os.path
dirname = os.path.dirname(__file__)
asset = Asset(self, "MyFile",
    path=os.path.join(dirname, "my-image.png"))

group = iam.Group(self, "MyUserGroup")
asset.grantRead(group)
```
Java

```java
import java.io.File;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);
        File startDir = new File(System.getProperty("user.dir"));
        Asset asset = Asset.Builder.create(this, "SampleAsset")
            .path(new File(startDir, "images/my-image.png").toString()).build();

        Group group = new Group(this, "MyUserGroup");
        asset.grantRead(group);
    }
}
```

C#

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png"))
});

var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
```

Docker image assets

The AWS CDK supports bundling local Docker images as assets through the `aws-ecr-assets` module. The following example defines a docker image that is built locally and pushed to Amazon ECR. Images are built from a local Docker context directory (with a Dockerfile) and uploaded to Amazon ECR by the AWS CDK CLI or your app's CI/CD pipeline, and can be naturally referenced in your AWS CDK app.

TypeScript

```typescript
import { DockerImageAsset } from '@aws-cdk/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
    directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```javascript
const { DockerImageAsset } = require('@aws-cdk/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'MyBuildImage', {
    directory: path.join(__dirname, 'my-image')
});
```
Asset types

Python

```python
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))
```

Java

```java
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
    .directory(new File(startDir, "my-image").toString()).build();
```

C#

```csharp
using System.IO;
using Amazon.CDK.AWS.Ecr.Assets;

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
    {
    });
```

The `my-image` directory must include a Dockerfile. The AWS CDK CLI builds a Docker image from `my-image`, pushes it to an Amazon ECR repository, and specifies the name of the repository as an AWS CloudFormation parameter to your stack. Docker image asset types expose deploy-time attributes (p. 84) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command `cdk synth` displays asset properties as AWS CloudFormation parameters.

**Amazon ECS task definition example**

A common use case is to create an Amazon ECS TaskDefinition to run docker containers. The following example specifies the location of a Docker image asset that the AWS CDK builds locally and pushes to Amazon ECR.

TypeScript

```typescript
import * as ecs from '@aws-cdk/aws-ecs';
import * as path from 'path';

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
    memoryLimitMiB: 1024,
    cpu: 512
});

taskDefinition.addContainer("my-other-container", {
    image: ecs.ContainerImage.fromAsset(path.join(__dirname, '..', "demo-image"))
});
```

JavaScript

```javascript
const ecs = require('@aws-cdk/aws-ecs');
```
const path = require('path');

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
    memoryLimitMiB: 1024,
    cpu: 512
});

taskDefinition.addContainer("my-other-container", {
    image: ecs.ContainerImage.fromAsset(path.join(__dirname, '..', "demo-image"))
});

Python

import aws_cdk.aws_ecs as ecs
import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_asset(os.path.join(dirname, '..', "demo-image")))

Java

import java.io.File;
import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

taskDefinition.addContainer("my-other-container",
    ContainerDefinitionOptions.builder()
        .image(ContainerImage.fromAsset(new File(startDir, 
            "demo-image").toString())).build());

C#

using System.IO;
using Amazon.CDK.AWS.ECS;

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
    FargateTaskDefinitionProps
    {
        MemoryLimitMiB = 1024,
        Cpu = 512
    });

    taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
        {
            "demo-image"));
        });
Deploy-time attributes example

The following example shows how to use the deploy-time attributes `repository` and `imageUri` to create an Amazon ECS task definition with the AWS Fargate launch type. Note that the Amazon ECR repo lookup requires the image's tag, not its URI, so we snip it from the end of the asset's URI.

**TypeScript**

```typescript
import * as ecs from '@aws-cdk/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from '@aws-cdk/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, '..', 'demo-image')
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository, asset.imageUri.split(':')[0].pop())
});
```

**JavaScript**

```javascript
const ecs = require('@aws-cdk/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('@aws-cdk/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, '..', 'demo-image')
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository, asset.imageUri.split(':')[0].pop())
});
```

**Python**

```python
import awscdk.aws_ecs as ecs
from awscdk.aws_ecr_assets import DockerImageAsset
import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'my-image',
  directory=os.path.join(dirname, '..', "demo-image"))

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
  memory_limit_mib=1024, cpu=512)

task_definition.add_container("my-other-container",
  image=ecs.ContainerImage.fromEcrRepository(asset.repository, asset.imageUri.split(':')[0].pop())
)```
Java

```java
import java.io.File;
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;
import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

File startDir = new File(System.getProperty("user.dir"));
DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
    .directory(new File(startDir, "demo-image").toString()).build();

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
    this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset's image URI for use in ECR repo lookup
String imageUri = asset.getImageUri();
String imageTag = imageUri.substring(imageUri.lastIndexOf(":" + 1);

taskDefinition.addContainer("my-other-container",
    ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
        asset.getRepository(), imageTag)).build());
```

C#

```csharp
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.Ecr.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
});

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
    FargateTaskDefinitionProps {
        MemoryLimitMiB = 1024,
        Cpu = 512
    });

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
    { Image = ContainerImage.FromEcrRepository(asset.Repository,
        asset.ImageUri.Split(":").Last())
});
```

Build arguments example

You can provide customized build arguments for the Docker build step through the `buildArgs` (Python: `build_args`) property option when the AWS CDK CLI builds the image during deployment.

TypeScript

```typescript
const asset = new DockerImageAsset(this, 'MyBuildImage', {
    directory: path.join(__dirname, 'my-image'),
    buildArgs: {
```
Permissions

If you use a module that supports Docker image assets, such as `aws-ecs`, the AWS CDK manages permissions for you when you use assets directly or through `ContainerImage.fromEcrRepository` (Python: `from_ecr_repository`). If you use Docker image assets directly, you need to ensure that the consuming principal has permissions to pull the image.

In most cases, you should use `asset.repository.grantPull` method (Python: `grant_pull`). This modifies the IAM policy of the principal to enable it to pull images from this repository. If the principal that is pulling the image is not in the same account or is an AWS service, such as AWS CodeBuild, that does not assume a role in your account, you must grant pull permissions on the resource policy and not on the principal's policy. Use the `asset.repository.addToResourcePolicy` method (Python: `add_to_resource_policy`) to grant the appropriate principal permissions.

AWS CloudFormation resource metadata

**Note**

This section is relevant only for construct authors. In certain situations, tools need to know that a certain CFN resource is using a local asset. For example, you can use the AWS SAM
CLI to invoke Lambda functions locally for debugging purposes. See the section called “SAM CLI” (p. 266) for details.

To enable such use cases, external tools consult a set of metadata entries on AWS CloudFormation resources:

- `aws:asset:path` – Points to the local path of the asset.
- `aws:asset:property` – The name of the resource property where the asset is used.

Using these two metadata entries, tools can identify that assets are used by a certain resource, and enable advanced local experiences.

To add these metadata entries to a resource, use the `asset.addResourceMetadata` (Python: `add_resource_metadata`) method.

### Permissions

The AWS Construct Library uses a few common, widely-implemented idioms to manage access and permissions. The IAM module provides you with the tools you need to use these idioms.

### Principals

An IAM principal is an entity that can be authenticated in order to access AWS resources, such as a user, a service, or an application. The AWS Construct Library supports many types of principals, including:

1. IAM resources such as `Role`, `User`, and `Group`
2. Service principals (`new iam.ServicePrincipal('service.amazonaws.com')`)
3. Federated principals (`new iam.FederatedPrincipal('cognito-identity.amazonaws.com')`)
4. Account principals (`new iam.AccountPrincipal('0123456789012')`)
5. Canonical user principals (`new iam.CanonicalUserPrincipal('79a59d[...]7ef2be')`)
6. AWS organizations principals (`new iam.OrganizationPrincipal('org-id')`)
7. Arbitrary ARN principals (`new iam.ArnPrincipal(res.arn)`)  
8. An `iam.CompositePrincipal(principal1, principal2, ...)` to trust multiple principals

### Grants

Every construct that represents a resource that can be accessed, such as an Amazon S3 bucket or Amazon DynamoDB table, has methods that grant access to another entity. All such methods have names starting with `grant`. For example, Amazon S3 buckets have the methods `grantRead` and `grantReadWrite` (Python: `grant_read`, `grant_read_write`) to enable read and read/write access, respectively, from an entity to the bucket without having to know exactly which Amazon S3 IAM permissions are required to perform these operations.

The first argument of a `grant` method is always of type `IGrantable`. This interface represents entities that can be granted permissions—that is, resources with roles, such as the IAM objects `Role`, `User`, and `Group`.

Other entities can also be granted permissions. For example, later in this topic, we show how to grant a CodeBuild project access to an Amazon S3 bucket. Generally, the associated role is obtained via a role property on the entity being granted access. Other entities that can be granted permissions are Amazon EC2 instances and CodeBuild projects.
Resources that use execution roles, such as `lambda.Function`, also implement `IGrantable`, so you can grant them access directly instead of granting access to their role. For example, if `bucket` is an Amazon S3 bucket, and `function` is a Lambda function, the code below grants the function read access to the bucket.

**TypeScript**

```typescript
bucket.grantRead(function);
```

**JavaScript**

```javascript
bucket.grantRead(function);
```

**Python**

```python
bucket.grant_read(function)
```

**Java**

```java
bucket.grantRead(function);
```

**C#**

```c#
bucket.GrantRead(function);
```

Sometimes permissions must be applied while your stack is being deployed. One such case is when you grant a AWS CloudFormation custom resource access to some other resource. The custom resource will be invoked during deployment, so it must have the specified permissions at deployment time. Another case is when a service verifies that the role you pass to it has the right policies applied (a number of AWS services do this to make sure you didn't forget to set the policies). In those cases, the deployment may fail if the permissions are applied too late.

To force the grant's permissions to be applied before another resource is created, you can add a dependency on the grant itself, as shown here. Though the return value of grant methods is commonly discarded, every grant method in fact returns an `iam.Grant` object.

**TypeScript**

```typescript
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

**JavaScript**

```javascript
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

**Python**

```python
grant = bucket.grant_read(function)
custom = CustomResource(...)  
custom.node.add_dependency(grant)
```
Roles

The IAM package contains a `Role` construct that represents IAM roles. The following code creates a new role, trusting the Amazon EC2 service.

**TypeScript**

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

const role = new iam.Role(this, 'Role', {
    assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

**JavaScript**

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

const role = new iam.Role(this, 'Role', {
    assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com')   // required
});
```

**Python**

```python
import aws_cdk.aws_iam as iam

role = iam.Role(self, "Role",
    assumed_by=iam.ServicePrincipal("ec2.amazonaws.com")) # required
```

**Java**

```java
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.iam.ServicePrincipal;

Role role = Role.Builder.create(this, "Role")
    .assumedBy(new ServicePrincipal("ec2.amazonaws.com")).build();
```

**C#**

```csharp
using Amazon.CDK.AWS.IAM;

var role = new Role(this, "Role", new RoleProps
    {   
    AssumedBy = new ServicePrincipal("ec2.amazonaws.com"),   // required
    });
```
AWS Cloud Development Kit (AWS CDK) Developer Guide
Roles

You can add permissions to a role by calling the role's addToPolicy method (Python: add_to_policy),
passing in a PolicyStatement that deﬁnes the rule to be added. The statement is added to the role's
default policy; if it has none, one is created.
The following example adds a Deny policy statement to the role for the actions ec2:SomeAction
and s3:AnotherAction on the resources bucket and otherRole (Python: other_role), under the
condition that the authorized service is AWS CodeBuild.
TypeScript
role.addToPolicy(new iam.PolicyStatement({
effect: iam.Effect.DENY,
resources: [bucket.bucketArn, otherRole.roleArn],
actions: ['ec2:SomeAction', 's3:AnotherAction'],
conditions: {StringEquals: {
'ec2:AuthorizedService': 'codebuild.amazonaws.com',
}}}));

JavaScript
role.addToPolicy(new iam.PolicyStatement({
effect: iam.Effect.DENY,
resources: [bucket.bucketArn, otherRole.roleArn],
actions: ['ec2:SomeAction', 's3:AnotherAction'],
conditions: {StringEquals: {
'ec2:AuthorizedService': 'codebuild.amazonaws.com'
}}}));

Python
role.add_to_policy(iam.PolicyStatement(
effect=iam.Effect.DENY,
resources=[bucket.bucket_arn, other_role.role_arn],
actions=["ec2:SomeAction", "s3:AnotherAction"],
conditions={"StringEquals": {
"ec2:AuthorizedService": "codebuild.amazonaws.com"}}
))

Java
role.addToPolicy(PolicyStatement.Builder.create()
.effect(Effect.DENY)
.resources(Arrays.asList(bucket.getBucketArn(), otherRole.getRoleArn()))
.actions(Arrays.asList("ec2:SomeAction", "s3:AnotherAction"))
.conditions(new HashMap<String, Object>() {{
put("StringEquals", new HashMap<String, String>() {{
put("ec2:AuthorizedService", "codebuild.amazonaws.com");
}});
}}).build());

C#
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
Effect = Effect.DENY,
Resources = new string[] { bucket.BucketArn, otherRole.RoleArn },
Actions = new string[] { "ec2:SomeAction", "s3:AnotherAction" },
Conditions = new Dictionary<string, object>
{

138


In our example above, we've created a new PolicyStatement inline with the addToPolicy (Python: add_to_policy) call. You can also pass in an existing policy statement or one you've modified. The PolicyStatement object has numerous methods for adding principals, resources, conditions, and actions.

If you're using a construct that requires a role to function correctly, you can either pass in an existing role when instantiating the construct object, or let the construct create a new role for you, trusting the appropriate service principal. The following example uses such a construct: a CodeBuild project.

**TypeScript**

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

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole: iam.IRole | undefined = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole,
});
```

**JavaScript**

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

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole
});
```

**Python**

```python
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
  # if role is None, the Project creates a new default role,
  # trusting the codebuild.amazonaws.com service principal
  role=some_role)
```

**Java**

```java
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;
```
Roles

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
  .role(someRole).build();

C# using Amazon.CDK.AWS.CodeBuild;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
var someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
var project = new Project(this, "Project", new ProjectProps
{
    Role = someRole
});

Once the object is created, the role (whether the role passed in or the default one created by the
construct) is available as the property role. This property is not available on imported resources,
however, so such constructs have an addToRolePolicy (Python: add_to_role_policy) method
that does nothing if the construct is an imported resource, and calls the addToPolicy (Python:
add_to_policy) method of the role property otherwise, saving you the trouble of handling the
undefined case explicitly. The following example demonstrates:

TypeScript

// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,   // ... and so on defining the policy
 }));

JavaScript

// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW   // ... and so on defining the policy
 }));

Python

# project is imported into the CDK application
project = codebuild.Project.from_project_name(self, 'Project', 'ProjectName')

# project is imported, so project.role is undefined, and this call has no effect
project.add_to_role_policy(iam.PolicyStatement(}
Resource policies

A few resources in AWS, such as Amazon S3 buckets and IAM roles, also have a resource policy. These constructs have an `addToResourcePolicy` method (Python: `add_to_resource_policy`), which takes a `PolicyStatement` as its argument. Every policy statement added to a resource policy must specify at least one principal.

In the following example, the Amazon S3 bucket grants a role with the `s3:SomeAction` permission to itself.

TypeScript

```typescript
bucket.addToResourcePolicy(new iam.PolicyStatement(
    effect: iam.Effect.ALLOW,
    actions: ['s3:SomeAction'],
    resources: [bucket.bucketArn],
    principals: [role]
));
```

JavaScript

```javascript
bucket.addToResourcePolicy(new iam.PolicyStatement({
    effect: iam.Effect.ALLOW,
    actions: ['s3:SomeAction'],
    resources: [bucket.bucketArn],
    principals: [role]
}));
```

Python

```python
bucket.add_to_resource_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=['s3:SomeAction'],
```
Context

resources=[bucket.bucket_arn],
    principals=role))

Java

bucket.addToResourcePolicy(PolicyStatement.Builder.create()
    .effect(Effect.ALLOW)
    .actions(Arrays.asList("s3:SomeAction"))
    .resources(Arrays.asList(bucket.getBucketArn()))
    .principals(Arrays.asList(role))
    .build());

C#

bucket.AddToResourcePolicy(new PolicyStatement(new PolicyStatementProps

  {
    Effect = Effect.ALLOW,
    Actions = new string[] { "s3:SomeAction" },
    Resources = new string[] { bucket.BucketArn },
    Principals = new IPrincipal[] { role }
  }));

Runtime context

Context values are key-value pairs that can be associated with a stack or construct. The AWS CDK uses context to cache information from your AWS account, such as the Availability Zones in your account or the Amazon Machine Image (AMI) IDs used to start your instances. The section called “Feature flags” (p. 147) are also context values. You can create your own context values for use by your apps or constructs.

Construct context

Context values are made available to your AWS CDK app in six different ways:

- Automatically from the current AWS account.
- Through the --context option to the cdk command.
- In the project's cdk.context.json file.
- In the project's cdk.json file.
- In the context key of your ~/.cdk.json file.
- In your AWS CDK app using the construct.node.setContext method.

The project file cdk.context.json is where the AWS CDK caches context values retrieved from your AWS account. This practice avoids unexpected changes to your deployments when, for example, a new Amazon Linux AMI is released, changing your Auto Scaling group. The AWS CDK does not write context data to any of the other files listed.

We recommend that your project's context files be placed under version control along with the rest of your application, as the information in them is part of your app's state and is critical to being able to synthesize and deploy consistently.

Context values are scoped to the construct that created them; they are visible to child constructs, but not to siblings. Context values set by the AWS CDK Toolkit (the cdk command), whether automatically, from a file, or from the --context option, are implicitly set on the App construct, and so are visible to every construct in the app.
You can get a context value using the `construct.node.tryGetContext` method. If the requested entry is not found on the current construct or any of its parents, the result is undefined (or your language's equivalent, such as `None` in Python).

**Context methods**

The AWS CDK supports several context methods that enable AWS CDK apps to get contextual information. For example, you can get a list of Availability Zones that are available in a given AWS account and AWS Region, using the `stack.availabilityZones` method.

The following are the context methods:

- **HostedZone.fromLookup**
  
  Gets the hosted zones in your account.

- **stack.availabilityZones**
  
  Gets the supported Availability Zones.

- **StringParameter.valueFromLookup**
  
  Gets a value from the current Region's Amazon EC2 Systems Manager Parameter Store.

- **Vpc.fromLookup**
  
  Gets the existing Amazon Virtual Private Clouds in your accounts.

- **LookupMachineImage**
  
  Looks up a machine image for use with a NAT instance in an Amazon Virtual Private Cloud.

If a given context information isn't available, the AWS CDK app notifies the AWS CDK CLI that the context information is missing. The CLI then queries the current AWS account for the information, stores the resulting context information in the `cdk.context.json` file, and executes the AWS CDK app again with the context values.

Don't forget to add the `cdk.context.json` file to your source control repository to ensure that subsequent `synth` commands will return the same result, and that your AWS account won't be needed when synthesizing from your build system.

**Viewing and managing context**

Use the `cdk context` command to view and manage the information in your `cdk.context.json` file. To see this information, use the `cdk context` command without any options. The output should be something like the following.

```
Context found in cdk.json:

# # # Key # # # Value

# # # availability-zones:account=123456789012:region=eu-central-1 # [ "eu-central-1a", "eu-central-1b", "eu-central-1c" ] #
# # # availability-zones:account=123456789012:region=eu-west-1 # [ "eu-west-1a", "eu-west-1b", "eu-west-1c" ] #

Run cdk context --reset KEY_OR_NUMBER to remove a context key. If it is a cached value, it will be refreshed on the next cdk synth.
```
To remove a context value, run `cdk context --reset`, specifying the value’s corresponding key or number. The following example removes the value that corresponds to the second key in the preceding example, which is the list of availability zones in the Ireland region.

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

Context value
availability-zones:account=123456789012:region=eu-west-1
reset. It will be refreshed on the next SDK synthesis run.

Therefore, if you want to update to the latest version of the Amazon Linux AMI, you can use the preceding example to do a controlled update of the context value and reset it, and then synthesize and deploy your app again.

```
$ cdk synth
```

To clear all of the stored context values for your app, run `cdk context --clear`, as follows.

```
$ cdk context --clear
```

Only context values stored in `cdk.context.json` can be reset or cleared. The AWS CDK does not touch other context files. To protect a context value from being reset using these commands, then, you might copy the value to `cdk.json`.

---

**AWS CDK Toolkit `--context` flag**

Use the `--context` (`-c` for short) option to pass runtime context values to your CDK app during synthesis or deployment.

```
# specify a single context value
cdk synth --context key=value MyStack

# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

When deploying multiple stacks, the specified context values are normally passed to all of them. If you wish, you may specify different values for each stack by prefixing the stack name to the context value.

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

**Example**

Below is an example of importing an existing Amazon VPC using AWS CDK context.

**TypeScript**

```typescript
import * as cdk from '@aws-cdk/core';
import * as ec2 from '@aws-cdk/aws-ec2';

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

---

144
const vpcid = this.node.tryGetContext('vpcid');
const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
  vpcId: vpcid,
});

const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

new cdk.CfnOutput(this, 'publicsubnets', {
  value: pubsubnets.subnetIds.toString(),
});
}
}

module.exports = { ExistsVpcStack }

JavaScript

const cdk = require('@aws-cdk/core');
const ec2 = require('@aws-cdk/aws-ec2');

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

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString()
    });
  }
}

module.exports = { ExistsVpcStack }

Python

import aws_cdk.core as cdk
import aws_cdk.aws_ec2 as ec2

class ExistsVpcStack(cdk.Stack):
  def __init__(scope: cdk.Construct, id: str, **kwargs):
    super().__init__(scope, id, **kwargs)

    vpcid = self.node.try_get_context("vpcid");
    vpc = ec2.Vpc.from_lookup(self, "VPC", vpc_id=vpcid)

    pubsubnets = vpc.select_subnets(subnetType=ec2.SubnetType.PUBLIC);

    cdk.CfnOutput(self, "publicsubnets",
      value=pubsubnets.subnet_ids.to_string())

Java

import software.amazon.awscdk.core.CfnOutput;
import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.ec2.VpcLookupOptions;
import software.amazon.awscdk.services.ec2.SelectedSubnets;
import software.amazon.awscdk.services.ec2.SubnetSelection;
import software.amazon.awscdk.services.ec2.SubnetType;

public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(App context, String id) {
        this(context, id, null);
    }

    public ExistsVpcStack(App context, String id, StackProps props) {
        super(context, id, props);

        String vpcId = (String) this.getNode().tryGetContext("vpcid");
        Vpc vpc = (Vpc) Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()
            .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()
            .subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
            .value(pubSubNets.getSubnetIds().toString()).build();
    }
}

C#

using Amazon.CDK;
using Amazon.CDK.AWS.EC2;

class ExistsVpcStack : Stack {
    public ExistsVpcStack(App scope, string id, StackProps props) : base(scope, id, props) {
        var vpcId = (string) this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
            {
                VpcId = vpcId
            });

        SelectedSubnets pubSubNets = vpc.SelectSubnets(new SubnetSelection
            {
                SubnetType = SubnetType.PUBLIC
            });

        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
}

You can use cdk diff to see the effects of passing in a context value on the command line:

```bash
$ cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
    {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
Feature flags

The AWS CDK uses feature flags to enable potentially breaking behaviors in a release. Flags are stored as the section called "Context" (p. 142) values in cdk.json (or ~/.cdk.json) as shown here.

```json
{
  "app": "npx ts-node bin/tscdk.ts",
  "context": {
    "@aws-cdk/core:enableStackNameDuplicates": "true"
  }
}
```

The names of all feature flags begin with the NPM name of the package affected by the particular flag. In the example above, this is @aws-cdk/core, the AWS CDK framework itself, since the flag affects stack naming rules, a core AWS CDK function. AWS Construct Library modules can also use feature flags.

Feature flags are disabled by default, so existing projects that do not specify the flag will continue to work as expected with later AWS CDK releases. New projects created using cdk init include flags enabling all features available in the release that created the project. Edit cdk.json to disable any flags for which you prefer the old behavior, or to add flags to enable new behaviors after upgrading the AWS CDK.
Aspects

See the CHANGELOG in a given release for a description of any new feature flags added in that release. The AWS CDK source file `features.ts` provides a complete list of all current feature flags.

As feature flags are stored in `cdk.json`, they are not removed by the `cdk context --reset` or `cdk context --clear` commands.

Aspects are the way to apply an operation to all constructs in a given scope. The functionality could modify the constructs, such as by adding tags, or it could be verifying something about the state of the constructs, such as ensuring that all buckets are encrypted.

To apply an aspect to a construct and all constructs in the same scope, call `node.applyAspect` (Python: `apply_aspect`) with a new aspect, as shown in the following example.

TypeScript

```typescript
myConstruct.node.applyAspect(new SomeAspect(/*...*/));
```

JavaScript

```javascript
myConstruct.node.applyAspect(new SomeAspect());
```

Python

```python
my_construct.node.apply_aspect(SomeAspect(...))
```

Java

```java
myConstruct.getNode().applyAspect(new SomeAspect(...));
```

C#

```csharp
myConstruct.Node.ApplyAspect(new SomeAspect(...));
```

The AWS CDK currently uses aspects only to tag resources (p. 115), but the framework is extensible and can also be used for other purposes. For example, you can use it to validate or change the AWS CloudFormation resources that are defined for you.

Aspects in detail

The AWS CDK implements tagging using a more generic system, called aspects, which is an instance of the visitor pattern. An aspect is a class that implements the following interface.

TypeScript

```typescript
interface IAspect {
    visit(node: IConstruct): void;
}
```

JavaScript

JavaScript doesn’t have interfaces as a language feature, so an aspect is simply an instance of a class having a `visit` method that accepts the node to be operated on.
Python

Python doesn't have interfaces as a language feature, so an aspect is simply an instance of a class having a visit method that accepts the node to be operated on.

Java

```java
public interface IAspect {
    public void visit(Construct node);
}
```

C#

```csharp
public interface IAspect
{
    void Visit(IConstruct node);
}
```

When you call `construct.node.applyAspect(aspect)` (Python: `apply_aspect`) the construct adds the aspect to an internal list of aspects.

During the prepare phase (p. 68), the AWS CDK calls the visit method of the object for the construct and each of its children in top-down order.

Although the aspect object is free to change any aspect of the construct object, it only operates on a specific subset of construct types. After determining the construct type, it can call any method and inspect or assign any property on the construct.

**Example**

The following example validates that all buckets created in the stack have versioning enabled. The aspect adds an error to the constructs that fail the validation, which results in the synth operation failing and prevents deploying the resulting cloud assembly.

TypeScript

```typescript
class BucketVersioningChecker implements IAspect {
    public visit(node: IConstruct): void {
        // See that we're dealing with a CfnBucket
        if (node instanceof s3.CfnBucket) {
            // Check for versioning property, exclude the case where the property
            // can be a token (IResolvable).
            if (!node.versioningConfiguration
                || (!Tokenization.isResolvable(node.versioningConfiguration)
                    && node.versioningConfiguration.status !== 'Enabled')) {
                node.node.addError('Bucket versioning is not enabled');
            }
        }
    }
}
```

// Apply to the stack
stack.node.applyAspect(new BucketVersioningChecker());

JavaScript

```javascript
class BucketVersioningChecker {
    visit(node) {
```

149
// See that we’re dealing with a CfnBucket
if (node instanceof s3.CfnBucket) {

    // Check for versioning property, exclude the case where the property
    // can be a token (IResolvable).
    if (!node.versioningConfiguration
        || !Tokenization.isResolvable(node.versioningConfiguration)
        && node.versioningConfiguration.status !== 'Enabled') {
        node.node.addError('Bucket versioning is not enabled');
    }
    
} 

// Apply to the stack
stack.node.applyAspect(new BucketVersioningChecker());

Python

@jsii.implements(core.IAspect)
class BucketVersioningChecker:

def visit(self, node):
    # See that we’re dealing with a CfnBucket
    if isinstance(node, s3.CfnBucket):

        # Check for versioning property, exclude the case where the property
        # can be a token (IResolvable).
        if (!node.versioning_configuration or
            !Tokenization.is_resolvable(node.versioning_configuration)
            and node.versioning_configuration.status != "Enabled"):
            node.node.add_error('Bucket versioning is not enabled')

    # Apply to the stack
    stack.node.apply_aspect(BucketVersioningChecker())

Java

public class BucketVersioningChecker implements IAspect
{
    @Override
    public void visit(Construct node) {
        // See that we’re dealing with a CfnBucket
        if (node instanceof CfnBucket) {
            CfnBucket bucket = (CfnBucket)node;
            Object versioningConfiguration = bucket.getVersioningConfiguration();
            if (versioningConfiguration == null ||
                !Tokenization.is_resolvable(versioningConfiguration.toString()) &&
                !versioningConfiguration.toString().contains("Enabled")
            )
                bucket.getNode().addError("Bucket versioning is not enabled");
        }
    }
}

C#

class BucketVersioningChecker : Amazon.Jsii.Runtime.DeputyBase, IAspect
{
    public void Visit(IConstruct node) {
        
}
Escape hatches

It's possible that neither the high-level constructs nor the low-level CFN Resource constructs have a specific feature you are looking for. There are three possible reasons for this lack of functionality:

- The AWS service feature is available through AWS CloudFormation, but there are no Construct classes for the service.
- The AWS service feature is available through AWS CloudFormation, and there are Construct classes for the service, but the Construct classes don't yet expose the feature.
- The feature is not yet available through AWS CloudFormation.

To determine whether a feature is available through AWS CloudFormation, see AWS Resource and Property Types Reference.

Using AWS CloudFormation constructs directly

If there are no Construct classes available for the service, you can fall back to the automatically generated CFN Resources, which map 1:1 onto all available AWS CloudFormation resources and properties. These resources can be recognized by their name starting with `Cfn`, such as `CfnBucket` or `CfnRole`. You instantiate them exactly as you would use the equivalent AWS CloudFormation resource. For more information, see AWS Resource and Property Types Reference.

For example, to instantiate a low-level Amazon S3 bucket CFN Resource with analytics enabled, you would write something like the following.

**TypeScript**

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

**JavaScript**

```javascript
new s3.CfnBucket(this, 'MyBucket', {
  analyticsConfigurations: [
    {
      id: 'Config',
      // ...
    }
  ]
});
```
Using AWS CloudFormation constructs directly

In the rare case where you want to define a resource that doesn't have a corresponding CfnXxx class, such as a new resource type that hasn't yet been published in the AWS CloudFormation resource specification, you can instantiate the cdk.CfnResource directly and specify the resource type and properties. This is shown in the following example.

TypeScript

```typescript
new cdk.CfnResource(this, 'MyBucket', {
    type: 'AWS::S3::Bucket',
    properties: {
        /* Note the PascalCase here! These are CloudFormation identifiers. */
        AnalyticsConfigurations: [
            { "id": "Config", // ...
            }
        ]
    }
});
```

JavaScript

```javascript
new cdk.CfnResource(this, 'MyBucket', {
    type: 'AWS::S3::Bucket',
    properties: {
        /* Note the PascalCase here! These are CloudFormation identifiers. */
        AnalyticsConfigurations: [
            { "id": "Config", // ...
            }
        ]
    }
});
```

Java

```java
CfnBucket.Builder.create(this, "MyBucket")
    .analyticsConfigurations(Arrays.asList(new HashMap<String, String>() {
        put("id", "Config");
        // ...
    })).build();
```
Modify the AWS CloudFormation resource behind AWS constructs

If a Construct is missing a feature or you are trying to work around an issue, you can modify the CFN Resource that is encapsulated by the Construct.

For more information, see AWS Resource and Property Types Reference.
All Constructs contain within them the corresponding CFN Resource. For example, the high-level Bucket construct wraps the low-level CfnBucket construct. Because the CfnBucket corresponds directly to the AWS CloudFormation resource, it exposes all features that are available through AWS CloudFormation.

The basic approach to get access to the CFN Resource class is to use `construct.node.defaultChild` (Python: `default_child`), cast it to the right type (if necessary), and modify its properties. Again, let's take the example of a Bucket.

**TypeScript**

```typescript
// Get the AWS CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }];
```

**JavaScript**

```javascript
// Get the AWS CloudFormation resource
const cfnBucket = bucket.node.defaultChild;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }];
```

**Python**

```python
# Get the AWS CloudFormation resource
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
  {
    "id": "Config",
    # ...
  }]
```

**Java**

```java
// Get the AWS CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations(
  Arrays.asList(new HashMap<String, String>() {
    put("Id", "Config");
    // ...
  }));
```

**C#**

```csharp
// Get the AWS CloudFormation resource
```
Raw overrides

If there are properties that are missing from the CFN Resource, you can bypass all typing using raw overrides. This also makes it possible to delete synthesized properties.

Use one of the addOverride methods (Python: add_override) methods, as shown in the following example.

TypeScript

```typescript
// Get the AWS CloudFormation resource
```
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// addPropertyOverride is a convenience function, which implies the
// path starts with "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
Custom resources

If the feature isn’t available through AWS CloudFormation, but only through a direct API call, the only solution is to write an AWS CloudFormation Custom Resource to make the API call you need. Don’t worry, the AWS CDK makes it easier to write these, and wrap them up into a regular construct interface, so from another user’s perspective the feature feels native.

Building a custom resource involves writing a Lambda function that responds to a resource’s CREATE, UPDATE and DELETE lifecycle events. If your custom resource needs to make only a single API call, consider using the AwsCustomResource. This makes it possible to perform arbitrary SDK calls during an AWS CloudFormation deployment. Otherwise, you should write your own Lambda function to perform the work you need to get done.

The subject is too broad to completely cover here, but the following links should get you started:

- Custom Resources
- Custom-Resource Example
- For a more fully fledged example, see the DnsValidatedCertificate class in the CDK standard library. This is implemented as a custom resource.
API reference

The API Reference contains information about the AWS CDK libraries.

Each library contains information about how to use the library. For example, the S3 library demonstrates how to set default encryption on an Amazon S3 bucket.

Versioning

Version numbers consist of three numeric version parts: major.minor.patch, and generally adhere to the semantic versioning model. This means that breaking changes to stable APIs are limited to major releases. Minor and patch releases are backward compatible, meaning that the code written in a previous version with the same major version can be upgraded to a newer version and be expected to continue to build and run, producing the same output.

Note
This compatibility promise does not apply to APIs under active development, which are designated as experimental. See the section called “AWS CDK stability index” (p. 158) for more details.

AWS CDK Toolkit (CLI) compatibility

The AWS CDK Toolkit (that is, the cdk command line command) is always compatible with construct libraries of a semantically lower or equal version number. It is, therefore, always safe to upgrade the AWS CDK Toolkit within the same major version.

The AWS CDK Toolkit may be, but is not always, compatible with construct libraries of a semantically higher version, depending on whether the same cloud assembly schema version is employed by the two components. The AWS CDK framework generates a cloud assembly during synthesis; the AWS CDK Toolkit consumes it for deployment. The schema that defines the format of the cloud assembly is strictly specified and versioned. AWS construct libraries using a given cloud assembly schema version are compatible with AWS CDK toolkit versions using that schema version or later, which may include releases of the AWS CDK Toolkit older than a given construct library release.

When the cloud assembly version required by the construct library is not compatible with the version supported by the AWS CDK Toolkit, you receive an error message like this one.

Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but found 4.0.0.
   Please upgrade your CLI in order to interact with this app.

Note
For more details on the cloud assembly schema, see Cloud Assembly Versioning.

AWS CDK stability index

The modules in the AWS Construct Library move through various stages as they are developed from concept to mature API. Different stages imply different promises for API stability in subsequent versions of the AWS CDK.
Stage 0: CFN resources

All construct library modules start in stage 0 when they are auto-generated from the AWS CloudFormation resource specification. The goal of stage 0 is to make new AWS CloudFormation resources/properties available to AWS CDK customers as quickly as possible. We capture feedback from customers to better understand what L2 resources to add.

AWS CloudFormation resources themselves are considered stable APIs, regardless of whether other constructs in the module are under active development.

Stage 1: Experimental

The goal of the experimental stage is to retain the freedom to make breaking changes to APIs while we design and build a module. During this stage, the primary use cases and the set of L2 constructs required to support them are incrementally identified, implemented, and validated.

Development of L2 constructs is community-oriented and transparent. For large and/or complex changes, we author a Request for Comments (RFC) that outlines our intended design and publish it for feedback. We also use pull requests to conduct API design reviews.

At this stage, individual APIs may be in flux, and breaking changes may occur from release to release if we deem these necessary to support customer use cases.

Stage 2: Developer preview (DP)

At the developer preview stage, our aim is to deliver a release candidate with a stable API with which to conduct user acceptance testing. When the API passes acceptance, it is deemed suitable for general availability.

We make breaking changes at this stage only when required to address unforeseen customer use cases or issues. Since breaking changes are still possible, the package itself retains the "experimental" label while in developer preview.

Stage 3: General availability (GA)

The module is generally available with a compatibility guarantee across minor versions. We will only make backward-compatible changes to the API, so that your existing apps will continue to work until the next major AWS CDK release.

In some cases, we may use feature flags (p. 147) to optionally enable new behavior while retaining the previous behavior to support existing apps.

Each module’s Overview in the API Reference describes its stability level.

For more information about these maturity stages, see AWS Construct Library Module Lifecycle.

Language binding stability

From time to time, we may add support to the AWS CDK for additional programming languages. Although the API described in all the languages is the same, the way that API is expressed varies by language and may change as the language support evolves. For this reason, language bindings are deemed experimental for a time until they are considered ready for production use. Currently, all supported languages are considered stable.

<table>
<thead>
<tr>
<th>Language</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeScript</td>
<td>Stable</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Stable</td>
</tr>
</tbody>
</table>
## Language binding stability

<table>
<thead>
<tr>
<th>Language</th>
<th>Stability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Stable</td>
</tr>
<tr>
<td>Java</td>
<td>Stable</td>
</tr>
<tr>
<td>C#/.NET</td>
<td>Stable</td>
</tr>
</tbody>
</table>
Examples

This topic contains the following examples:

- Creating a serverless application using the AWS CDK (p. 161) Creates a serverless application using Lambda, API Gateway, and Amazon S3.
- Creating an AWS Fargate service using the AWS CDK (p. 174) Creates an Amazon ECS Fargate service from an image on DockerHub.
- Creating a pipeline using the AWS CDK (p. 180) Creates a CI/CD pipeline.

Creating a serverless application using the AWS CDK

This example walks you through how to create the resources for a simple widget dispensing service. (For the purpose of this example, a widget is just a name or identifier that can be added to, retrieved from, and deleted from a collection.) The example includes:

- An AWS Lambda function.
- An Amazon API Gateway API to call the Lambda function.
- An Amazon S3 bucket that 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 HTTP GET /
3. Creates the service that calls the Lambda function
4. Adds the service to the AWS CDK app
5. Tests the app
6. Adds 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 app MyWidgetService in the current folder.

TypeScript

```bash
mkdir MyWidgetService
cd MyWidgetService
cdk init --language typescript
```
Create a AWS CDK app

**JavaScript**

```shell
mkdir MyWidgetService
cd MyWidgetService
cdk init --language javascript
```

**Python**

```shell
mkdir MyWidgetService
cd MyWidgetService
cdk init --language python
source .env/bin/activate
pip install -r requirements.txt
```

**Java**

```shell
mkdir MyWidgetService
cd MyWidgetService
cdk init --language java
```

You may now import the Maven project into your IDE.

**C#**

```shell
mkdir MyWidgetService
cd MyWidgetService
cdk init --language csharp
```

You may now open src/MyWidgetService.sln in Visual Studio.

The important files in the blank project are as follows. (We will also be adding a couple of new files.)

**TypeScript**

- `bin/my_widget_service.ts` – Main entry point for the application
- `lib/my_widget_service-stack.ts` – Defines the widget service stack

**JavaScript**

- `bin/my_widget_service.js` – Main entry point for the application
- `lib/my_widget_service-stack.js` – Defines the widget service stack

**Python**

- `app.py` – Main entry point for the application
- `my_widget_service/my_widget_service_stack.py` – Defines the widget service stack

**Java**

- `src/main/java/com/myorg/MyWidgetServiceApp.java` – Main entry point for the application
- `src/main/java/com/myorg/MyWidgetServiceStack.java` – Defines the widget service stack
Remember to replace the bucket name with the name of your bucket.

Run the app and note that it synthesizes an empty stack.

cdk synth

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

<table>
<thead>
<tr>
<th>Resources:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDRMetadata:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
| | 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. We will provide the Lambda function’s code in JavaScript.

Create the `resources` directory in the project’s main directory.

```bash
mkdir resources
```

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

```javascript
/*
This code uses callbacks to handle asynchronous function responses.
It currently demonstrates using an async-await pattern.
AWS supports both the async-await and promises patterns.
For more information, see the following:
https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/calling-services-asynchronously.html
https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html
*/
const AWS = require('aws-sdk');
const S3 = new AWS.S3();
const bucketName = process.env.BUCKET;
exports.main = async function(event, context) {
  try {
    var method = event.httpMethod;
    if (method === "GET") {
      if (event.path === "/") {
        const data = await S3.listObjectsV2({ Bucket: bucketName }).promise();
        var body = {
          widgets: data.Contents.map(function(e) { return e.Key })
        };
        return {
          statusCode: 200,
```
headers: {},
body: JSON.stringify(body)
};

// We only accept GET for now
return {
  statusCode: 400,
  headers: {},
  body: "We only accept GET /"
};
} catch(error) {
  var body = error.stack || JSON.stringify(error, null, 2);
  return {
    statusCode: 400,
    headers: {},
    body: JSON.stringify(body)
  }
}

Save it and be sure the project still results in an empty stack. We haven't yet wired the Lambda function to the AWS CDK app, so the Lambda asset doesn't appear in the output.

cdk synth

Creating a widget service

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

TypeScript

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

JavaScript

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

Python

```bash
pip install aws_cdk.aws_apigateway aws_cdk.aws_lambda aws_cdk.aws_s3
```

Java

Using your IDE's Maven integration (e.g., in Eclipse, right-click your project and choose Maven > Add Dependency), install the following artifacts from the group software.amazon.awscdk:

```java
apigateway
lambda
s3
```

C#

Choose Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio and add the following packages.
Tip
If you don’t see these packages in the Browse tab of the Manage Packages for Solution page, make sure the Include prerelease checkbox is ticked. For a better experience, also add the Amazon.Jsii.Analyzers package to provide compile-time checks for missing required properties.

Create a new source file to define the widget service with the source code shown below.

TypeScript

File: lib/widget_service.ts

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

export class WidgetService extends core.Construct {
  constructor(scope: core.Construct, id: string) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'WidgetStore');
    const handler = new lambda.Function(this, 'WidgetHandler', {
      runtime: lambda.Runtime.NODEJS_10_X, // So we can use async in widget.js
      code: lambda.Code.asset('resources'),
      handler: 'widgets.main',
      environment: {
        BUCKET: bucket.bucketName
      }
    });
    bucket.grantReadWrite(handler); // was: handler.role);
    const api = new apigateway.RestApi(this, 'widgets-api', {
      restApiName: 'Widget Service',
      description: 'This service serves widgets.'
    });
    const getWidgetsIntegration = new apigateway.LambdaIntegration(handler, {
      requestTemplates: { 'application/json': '{ "statusCode": "200" }' }
    });
    api.root.addMethod('GET', getWidgetsIntegration); // GET /
  }
}
```

JavaScript

File: lib/widget_service.js

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

class WidgetService extends core.Construct {
```

165
constructor(scope, id) {
    super(scope, id);

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

    const handler = new lambda.Function(this, "WidgetHandler", {
        runtime: lambda.Runtime.NODEJS_10_X, // So we can use async in widget.js
        code: lambda.Code.asset("resources"),
        handler: "widgets.main",
        environment: {  
            BUCKET: bucket.bucketName
        }
    });

    bucket.grantReadWrite(handler); // was: handler.role);

    const api = new apigateway.RestApi(this, "widgets-api", {
        restApiName: "Widget Service",
        description: "This service serves widgets."
    });

    const getWidgetsIntegration = new apigateway.LambdaIntegration(handler, {
        requestTemplates: { "application/json": '{ "statusCode": "200" }' }
    });

    api.root.addMethod("GET", getWidgetsIntegration); // GET /
}

module.exports = { WidgetService }

Python

File: my_widget_service/widget_service.py

    from aws_cdk import (core,  
                        aws_apigateway as apigateway,  
                        aws_s3 as s3,  
                        aws_lambda as lambda_)

    class WidgetService(core.Construct):
        def __init__(self, scope: core.Construct, id: str):
            super().__init__(scope, id)

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

            handler = lambda_.Function(self, "WidgetHandler",  
                                    runtime=lambda_.Runtime.NODEJS_10_X,  
                                    code=lambda_.Code.asset("resources"),  
                                    handler="widgets.main",  
                                    environment=dict(  
                                        BUCKET=bucket.bucket_name)
                        )

                        bucket.grant_read_write(handler)

            api = apigateway.RestApi(self, "widgets-api",  
                                      rest_api_name="Widget Service",  
                                      description="This service serves widgets.")

            get_widgets_integration = apigateway.LambdaIntegration(handler,  
                                                                  request_templates={'"application/json": "{ "statusCode": "200" }"'})

            api.root.add_method("GET", get_widgets_integration)  # GET /
Java

File: src/src/main/java/com/myorg/WidgetService.java

```java
package com.myorg;

import java.util.HashMap;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.services.apigateway.LambdaIntegration;
import software.amazon.awscdk.services.apigateway.Resource;
import software.amazon.awscdk.services.apigateway.RestApi;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.Bucket;

public class WidgetService extends Construct {
    public WidgetService(Construct scope, String id) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "WidgetStore");

        Function handler = Function.Builder.create(this, "WidgetHandler")
            .runtime(Runtime.NODEJS_10_X)
            .code(Code.fromAsset("resources"))
            .handler("widgets.main")
            .environment(new HashMap<String, String>() {
                put("BUCKET", bucket.getBucketName());
            }).build();

        bucket.grantReadWrite(handler);

        RestApi api = RestApi.Builder.create(this, "Widgets-API")
            .restApiName("Widget Service").description("This service services widgets.")
            .build();

        LambdaIntegration getWidgetsIntegration = LambdaIntegration.Builder.create(handler)
            .requestTemplates(new HashMap<String, String>() {
                put("application/json", "{ \"statusCode\": \"200\" }"));
            }).build();

        api.getRoot().addMethod("GET", getWidgetsIntegration);
    }
}
```

C#

File: src/MyWidgetService/WidgetService.cs

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3;
using System.Collections.Generic;

namespace MyWidgetService
{
    public class WidgetService : Construct
    {
    }
}
```
Add the service to the app

To add the widget service to our AWS CDK app, we'll need to modify the source file that defines the stack to instantiate the service construct.

TypeScript

File: lib/my_widget_service-stack.ts

Add the following line of code after the existing import statement.

```typescript
import * as widget_service from '../lib/widget_service';
```

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

```typescript
new widget_service.WidgetService(this, 'Widgets');
```
Deploy and test the app

Before you can deploy your first AWS CDK app containing a lambda function, you must bootstrap your AWS environment. This creates a staging bucket that the AWS CDK uses to deploy stacks containing assets. For details, see the section called “Bootstrapping your AWS environment” (p. 257). If you've already bootstrapped, you'll get a warning and nothing will change.

```
cdk synth
```
Add the individual widget functions

Now we're ready to deploy the app as follows.

```bash
cdk bootstrap
```

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.

```text
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.

```bash
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
```

You can also test the app by:

1. Opening the AWS Management Console.
2. Navigating to the API Gateway service.
3. Finding **Widget Service** in the list.
4. Selecting **GET** and **Test** to test the function.

Because we haven't stored any widgets yet, the output should be similar to the following.

```json
{ "widgets": [] }
```

### Add the individual widget functions

The next step is to create Lambda functions to create, show, and delete individual widgets.

Replace the code in `widgets.js` (in `resources`) with the following.

```javascript
const AWS = require('aws-sdk');
const S3 = new AWS.S3();

const bucketName = process.env.BUCKET;

/*
This code uses callbacks to handle asynchronous function responses. It currently demonstrates using an async-await pattern. AWS supports both the async-await and promises patterns. For more information, see the following:
https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/calling-services-asynchronously.html
https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html
*/
exports.main = async function(event, context) {
  try {
    var method = event.httpMethod;
    // Get name, if present
```
var widgetName = event.path.startsWith('/') ? event.path.substring(1) : event.path;

if (method === "GET") {
    // GET / to get the names of all widgets
    if (event.path === "/") {
        const data = await S3.listObjectsV2({ Bucket: bucketName }).promise();
        var body = {
            widgets: data.Contents.map(function(e) { return e.Key })
        };
        return {
            statusCode: 200,
            headers: {},
            body: JSON.stringify(body)
        };
    }
    if (widgetName) {
        // GET /name to get info on widget name
        const data = await S3.getObject({ Bucket: bucketName, Key: widgetName}).promise();
        var body = data.Body.toString('utf-8');
        return {
            statusCode: 200,
            headers: {},
            body: JSON.stringify(body)
        };
    }
}

if (method === "POST") {
    // POST /name
    // Return error if we do not have a name
    if (!widgetName) {
        return {
            statusCode: 400,
            headers: {},
            body: "Widget name missing"
        };
    }
    // Create some dummy data to populate object
    const now = new Date();
    var data = widgetName + " created: " + now;
    var base64data = new Buffer(data, 'binary');
    await S3.putObject({
        Bucket: bucketName,
        Key: widgetName,
        Body: base64data,
        ContentType: 'application/json'
    }).promise();
    return {
        statusCode: 200,
        headers: {},
        body: JSON.stringify(event.widgets)
    };
}

if (method === "DELETE") {
    // DELETE /name
    // Return an error if we do not have a name
    if (!widgetName) {
        return {
            statusCode: 400,
headers: {},
   body: "Widget name missing"
};

await S3.deleteObject({
   Bucket: bucketName, Key: widgetName
}).promise();

return {
   statusCode: 200,
   headers: {},
   body: "Successfully deleted widget " + widgetName
};

// We got something besides a GET, POST, or DELETE
return {
   statusCode: 400,
   headers: {},
   body: "We only accept GET, POST, and DELETE, not " + method
};
} catch(error) {
   var body = error.stack || JSON.stringify(error, null, 2);
   return {
      statusCode: 400,
      headers: {},
      body: body
   }
}

Wire up these functions to your API Gateway code at the end of the WidgetService constructor.

TypeScript

File: lib/widget_service.ts

const widget = api.root.addResource("{id}");

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

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

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

widget.addMethod("POST", postWidgetIntegration); // POST /{id}
widget.addMethod("GET", getWidgetIntegration); // GET /{id}
widget.addMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}

JavaScript

File: lib/widget_service.js

const widget = api.root.addResource("{id}");

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

// Get a specific widget from bucket with: GET /{id}
Add the individual widget functions

```python
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", getWidgetIntegration); // POST /{id}
widget.addMethod("GET", getWidgetIntegration); // GET /{id}
widget.addMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}
```

Python

File: my_widget_service/widget_service.py

```python
widget = api.root.add_resource("{id}"
#
post_widget_integration = apigateway.LambdaIntegration(handler)
#
get_widget_integration = apigateway.LambdaIntegration(handler)
#
delete_widget_integration = apigateway.LambdaIntegration(handler)

widget.add_method("POST", post_widget_integration); # POST /{id}
widget.add_method("GET", get_widget_integration); # GET /{id}
widget.add_method("DELETE", delete_widget_integration); # DELETE /{id}
```

Java

File: src/src/main/java/com/myorg/WidgetService.java

```java
// Add new widget to bucket with: POST /{id}
LambdaIntegration postWidgetIntegration = new LambdaIntegration(handler);

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

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

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

C#

File: src/MyWidgetService/WidgetService.cs

```csharp
var widget = api.Root.AddResource("{id}"
#
var postWidgetIntegration = new LambdaIntegration(handler);
#
var getWidgetIntegration = new LambdaIntegration(handler);
#
var deleteWidgetIntegration = new LambdaIntegration(handler);

widget.AddMethod("POST", postWidgetIntegration); // POST /{id}
widget.AddMethod("GET", getWidgetIntegration); // GET /{id}
```
Save and deploy the app.

```
cdk deploy
```

We can now store, show, or delete an individual widget. Use the following commands to list the widgets, create the widget `example`, list all of the widgets, show the contents of `example` (it should show today's date), delete `example`, and then show the list of widgets again.

```
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
curl -X POST 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X DELETE 'https://GUID.execute-api.REGION.amazonaws.com/prod/example'
curl -X GET 'https://GUID.execute-api.REGION.amazonaws.com/prod'
```

You can also use the API Gateway console to test these functions. Set the `name` value to the name of a widget, such as `example`.

### Clean up

To avoid unexpected AWS charges, destroy your AWS CDK stack after you're done with this exercise.

```
cdk destroy
```

### Creating an AWS Fargate service using the AWS CDK

This example walks you through how to create an AWS Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster that's fronted by an internet-facing Application Load Balancer from an image on Amazon ECR.

Amazon ECS is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on a serverless infrastructure that's managed by Amazon ECS by launching your services or tasks using the Fargate launch type. For more control, you can host your tasks on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances that you manage by using the Amazon EC2 launch type.

This tutorial shows you how to launch some services using the Fargate launch type. If you've used the AWS Management Console to create a Fargate service, you know that there are many steps to follow to accomplish that task. AWS has several tutorials and documentation topics that walk you through creating a Fargate service, including:

- How to Deploy Docker Containers - AWS
- Setting Up with Amazon ECS
- Getting Started with Amazon ECS Using Fargate

This example creates a similar Fargate service in AWS CDK code.
The Amazon ECS construct used in this tutorial helps you use AWS services by providing the following benefits:

- Automatically configures a load balancer.
- Automatically opens a security group for load balancers. This enables load balancers to communicate with instances without you explicitly creating a security group.
- Automatically orders dependency between the service and the load balancer attaching to a target group, where the AWS CDK enforces the correct order of creating the listener before an instance is created.
- Automatically configures user data on automatically scaling groups. This creates the correct configuration to associate a cluster to AMIs.
- Validates parameter combinations early. This exposes AWS CloudFormation issues earlier, thus saving you deployment time. For example, depending on the task, it's easy to misconfigure the memory settings. Previously, you would not encounter an error until you deployed your app. But now the AWS CDK can detect a misconfiguration and emit an error when you synthesize your app.
- Automatically adds permissions for Amazon Elastic Container Registry (Amazon ECR) if you use an image from Amazon ECR.
- Automatically scales. The AWS CDK supplies a method so you can autoscaling instances when you use an Amazon EC2 cluster. This happens automatically when you use an instance in a Fargate cluster.

In addition, the AWS CDK prevents an instance from being deleted when automatic scaling tries to kill an instance, but either a task is running or is scheduled on that instance.

Previously, you had to create a Lambda function to have this functionality.

- Provides asset support, so that you can deploy a source from your machine to Amazon ECS in one step. Previously, to use an application source you had to perform several manual steps, such as uploading to Amazon ECR and creating a Docker image.

See ECS for details.

Creating the directory and initializing the AWS CDK

Let's start by creating a directory to hold the AWS CDK code, and then creating a AWS CDK app in that directory.

**TypeScript**

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

**JavaScript**

```bash
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language javascript
```

**Python**

```bash
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language python
source .env/bin/activate
pip install -r requirements.txt
```
### Java

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

You may now import the Maven project into your IDE.

### C#

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

You may now open `src/MyEcsConstruct.sln` in Visual Studio.

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

```
cdk synth
```

You should see a stack like the following, where `CDK-VERSION` is the version of the CDK and `NODE-VERSION` is the version of Node.js. (Your output may differ slightly from what's shown here.)

```
Resources:
CDKMetadata:
  Type: AWS::CDK::Metadata
  Properties:
    Modules: aws-cdk=CDK-VERSION,aws-cdk/core=CDK-VERSION,aws-cdk/cx-api=CDK-VERSION,jsii-runtime=node.js/NODE-VERSION
```

### Add the Amazon EC2 and Amazon ECS packages

Install the AWS construct library modules for Amazon EC2 and Amazon ECS.

#### TypeScript

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

#### JavaScript

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

#### Python

```
pip install aws_cdk.aws_ec2 aws_cdk.aws_ecs aws_cdk.aws_ecs_patterns
```

#### Java

Using your IDE's Maven integration (e.g., in Eclipse, right-click your project and choose Maven > Add Dependency), install the following artifacts from the group `software.amazon.awscdk`:

```
ec2
ecs
ecs-patterns
```
C#

Choose Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio and add the following packages.

Amazon.CDK.AWS.EC2
Amazon.CDK.AWS.ECS
Amazon.CDK.AWS.ECS.Patterns

Tip
If you don’t see these packages in the Browse tab of the Manage Packages for Solution page, make sure the Include prerelease checkbox is ticked. For a better experience, also add the Amazon.Jsii.Analyzers package to provide compile-time checks for missing required properties.

Create a Fargate service

There are two different ways to run your container tasks with Amazon ECS:

- Use the Fargate launch type, where Amazon ECS manages the physical machines that your containers are running on for you.
- Use the EC2 launch type, where you do the managing, such as specifying automatic scaling.

For this example, we’ll create a Fargate service running on an ECS cluster fronted by an internet-facing Application Load Balancer.

Add the following AWS Construct Library module imports to the indicated file.

TypeScript

File: lib/my_ecs_construct-stack.ts

```typescript
import * as ec2 from "@aws-cdk/aws-ec2";
import * as ecs from "@aws-cdk/aws-ecs";
import * as ecs_patterns from "@aws-cdk/aws-ecs-patterns";
```

JavaScript

File: lib/my_ecs_construct-stack.js

```javascript
const ec2 = require("@aws-cdk/aws-ec2");
const ecs = require("@aws-cdk/aws-ecs");
const ecs_patterns = require("@aws-cdk/aws-ecs-patterns");
```

Python

File: my_ecs_construct/my_ecs_construct_stack.py

```python
from aws_cdk import (core, aws_ec2 as ec2, aws_ecs as ecs, aws_ecs_patterns as ecs_patterns)
```

Java

File: src/main/java/com/myorg/MyEcsConstructStack.java
import software.amazon.awscdk.services.ec2.*;
import software.amazon.awscdk.services.ecs.*;
import software.amazon.awscdk.services.ecs.patterns.*;

C#
File: src/MyEcsConstruct/MyEcsConstructStack.cs

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

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

TypeScript

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

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

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

JavaScript

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

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

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

Python

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

```python
class = ecs.Cluster(self, "MyCluster", vpc=vpc)
ecas_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
class=cluster, # Required
cpu=512, # Default is 256
desired_count=6, # Default is 1
task_image_options=ecas_patterns.ApplicationLoadBalancedTaskImageOptions(
  image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
memory_limit_mib=2048, # Default is 512
public_load_balancer=True) # Default is False
```

Java

```java
Vpc vpc = Vpc.Builder.create(this, "MyVpc")
  .maxAzs(3) // Default is all AZs in region
  .build();
Cluster cluster = Cluster.Builder.create(this, "MyCluster")
  .vpc(vpc).build();

// Create a load-balanced Fargate service and make it public
ApplicationLoadBalancedFargateService.Builder.create(this, "MyFargateService")
  .cluster(cluster) // Required
  .cpu(512) // Default is 256
  .desiredCount(6) // Default is 1
  .taskImageOptions(
    ApplicationLoadBalancedTaskImageOptions.builder()
    .image(ContainerImage.fromRegistry("amazon/amazon-ecs-sample"))
    .build())
  .memoryLimitMib(2048) // Default is 512
  .publicLoadBalancer(true) // Default is false
  .build();
```

C#

```csharp
var vpc = new Vpc(this, "MyVpc", new VpcProps
{
  MaxAzs = 3 // Default is all AZs in region
});
var cluster = new Cluster(this, "MyCluster", new ClusterProps
{
  Vpc = vpc
});

// Create a load-balanced Fargate service and make it public
new ApplicationLoadBalancedFargateService(this, "MyFargateService", new ApplicationLoadBalancedFargateServiceProps
{
  Cluster = cluster, // Required
  DesiredCount = 6, // Default is 1
  TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
  {
    Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
  },
  MemoryLimitMib = 2048, // Default is 256
  PublicLoadBalancer = true // Default is false
});
```

Save it and make sure it runs and creates a stack.
Clean up

To avoid unexpected AWS charges, destroy your AWS CDK stack after you're done with this exercise.

```
cdk destroy
```

Creating a pipeline using the AWS CDK

The AWS CDK lets you easily define applications in the AWS Cloud using your programming language of choice. But creating an application is just the start of the journey. You also want to make changes to it and deploy them. You can do this through the Code suite of tools: AWS CodeCommit, AWS CodeBuild, AWS CodeDeploy, and AWS CodePipeline. Together, they allow you to build what's called a deployment pipeline for your application. This example shows how to deploy an AWS Lambda function using such a pipeline.

How it works

Our application contains two AWS CDK stacks. The first stack, PipelineStack, defines the pipeline itself. The second, LambdaStack, is used to deploy the Lambda function.

The key to this example is that you deploy PipelineStack from your own workstation, but LambdaStack is deployed by the pipeline; you never deploy it yourself.

Since the LambdaStack is deployed by the pipeline, it must be available to the pipeline (along with the Lambda code). Therefore, this app and the Lambda function are stored in a CodeCommit repository.

The PipelineStack contains the definitions of the pipeline, which includes build steps for both the Lambda function and the LambdaStack.

Prerequisites

Beyond having the AWS CDK set up and configured, your workstation needs to be able to push to AWS CodeCommit using Git, which means you need some way of identifying yourself to CodeCommit. The easiest way to do this is to configure Git credentials for an IAM user, as described in Setup for HTTPS users using Git credentials.

You can also use the git-remote-codecommit Git add-on or other methods of connecting and authenticating supported by CodeCommit.
Also, make sure you have issued `cdk bootstrap`, as the Amazon S3 bucket in the bootstrap stack is required to deploy a Lambda function with the AWS CDK.

## Setting up the project

To set up a new AWS CDK project in CodeCommit:

1. **Create a new CodeCommit repository** named `pipeline` using the CodeCommit console or the AWS CLI.

   if you already have a CodeCommit repository named `pipeline`, you can use another name. Just make sure you clone it to a directory named `pipeline` on your local system.

2. **Clone this new repository** to your local computer in a directory named `pipeline`. If you are authenticating with an IAM user with Git credentials, copy the HTTPS URL from the CodeCommit console. (Other authentication methods require a different URL.)

   ```
   git clone CODECOMMIT-REPO-URL pipeline
   ```

   Enter your credentials if prompted for them.

   **Note**
   During cloning, Git will warn you that you appear to have cloned an empty repository; this is normal and expected.

3. **Change to the pipeline directory** and initialize it as a new CDK project, then install the AWS Construct Libraries we'll use in our app.

   **TypeScript**

   ```
   cd pipeline
   cdk init --language typescript
   npm install @aws-cdk/aws-codedeploy @aws-cdk/aws-lambda @aws-cdk/aws-codebuild @aws-cdk/aws-codepipeline
   npm install @aws-cdk/aws-codecommit @aws-cdk/aws-codepipeline-actions @aws-cdk/aws-s3
   ```

   **JavaScript**

   ```
   cd pipeline
   cdk init --language javascript
   npm install @aws-cdk/aws-codedeploy @aws-cdk/aws-lambda @aws-cdk/aws-codebuild @aws-cdk/aws-codepipeline
   npm install @aws-cdk/aws-codecommit @aws-cdk/aws-codepipeline-actions @aws-cdk/aws-s3
   ```

   **Python**

   A couple of commands differ between Windows and Mac OS X or Linux.

   **Mac OS X/Linux**

   ```
   cd pipeline
   cdk init --language python
   source .env/bin/activate
   git commit -m "project started"
   pip install -r requirements.txt
   pip install aws_cdk.aws_codepipeline
   pip install aws_cdk.aws_codecommit aws_cdk.aws_codepipeline_actions aws_cdk.aws_s3
   pip freeze | grep -v '-e git' > requirements.txt
   ```
### Setting up the project

#### Windows

```
cd pipeline
cdk init --language python
.env\Scripts\activate.bat
pip install -r requirements.txt
pip install aws_cdk.aws_codedeploy aws_cdk.aws_lambda aws_cdk.aws_codebuild
aws_cdk.aws_codepipeline
pip install aws_cdk.aws_codecommit aws_cdk.aws_codepipeline_actions aws_cdk.aws_s3
pip freeze | find /V "-e git" > requirements.txt
```

#### Java

```
cd pipeline
cdk init --language java
```

You can import the resulting Maven project into your Java IDE.

Using the Maven integration in your IDE (for example, in Eclipse, right-click the project and choose Maven > Add Dependency), add the following packages in the group software.amazon.awscdk.

```
lambda
codedeploy
codebuild
codecommit
codepipeline
codepipeline-actions
s3
```

Alternatively, add `<dependency>` elements like the following to `pom.xml`. You can copy the existing dependency for the AWS CDK core module and modify it. For example, a dependency for the AWS Lambda module looks like this.

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

#### C#

```
cd pipeline
cdk init --language csharp
```

You can open the file `src/Pipeline.sln` in Visual Studio.

Choose Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio and add the following packages.

```
Amazon.CDK.AWS.CodeDeploy
Amazon.CDK.AWS.CodeBuild
Amazon.CDK.AWS.CodeCommit
Amazon.CDK.AWS.CodePipeline
Amazon.CDK.AWS.CodePipeline.Actions
Amazon.CDK.AWS.Lambda
```
Add Lambda code

1. Create a directory for your AWS Lambda code.

    mkdir lambda

2. Place your AWS Lambda function in the new directory. Our CDK app expects a Lambda function written in TypeScript, with a main file of `index.ts` and a main function named `main()`, regardless of what language the rest of the app is written in. The function will be built (transpiled to JavaScript) by the TypeScript compiler as part of the pipeline. Some changes will be needed in the Lambda build process if your function is written in another language.

   If you don't have a function handy to play with, this one will do:

   ```ts
   // index.ts
   const GREETING = "Hello, AWS!";
   export async function main(event: any, context: any) {
     console.log(GREETING);
     return GREETING;
   }
   ```

3. Commit your changes and push.

    ```
    git add --all
    git commit -m "add lambda function"
    git push
    ```

Define Lambda stack

Let's define the AWS CloudFormation stack that will create the Lambda function, the stack that we'll deploy in our pipeline. We'll create a new file to hold this stack.
We need some way to get a reference to the Lambda function we’ll be deploying. This code is built by the pipeline, and the pipeline passes us a reference to it as AWS CloudFormation parameters. We get it using the `fromCfnParameters()` method and store it as an attribute named `lambdaCode`, where it can be picked up by the deployment stage of the pipeline.

The example also uses the CodeDeploy support for blue-green deployments to Lambda, transferring traffic to the new version in 10-percent increments every minute. As blue-green deployment can only operate on aliases, not on the function directly, we create an alias for our function, named `Prod`.

The alias uses a Lambda version obtained using the function's `currentVersion` property. This ensures that every invocation of the AWS CDK code publishes a new version of the function.

If the Lambda function needs any other resources when executing, such as an Amazon S3 bucket, Amazon DynamoDB table, or Amazon API Gateway, you’d declare those resources here.

**TypeScript**

```typescript
File: lib/lambda-stack.ts

import * as codedeploy from '@aws-cdk/aws-codedeploy';
import * as lambda from '@aws-cdk/aws-lambda';
import { App, Stack, StackProps } from '@aws-cdk/core';

export class LambdaStack extends Stack {
  public readonly lambdaCode: lambda.CfnParametersCode;

  constructor(app: App, id: string, props?: StackProps) {
    super(app, id, props);

    this.lambdaCode = lambda.Code.fromCfnParameters();

    const func = new lambda.Function(this, 'Lambda', {
      code: this.lambdaCode,
      handler: 'index.main',
      runtime: lambda.Runtime.NODEJS_10_X,
    });

    const alias = new lambda.Alias(this, 'LambdaAlias', {
      aliasName: 'Prod',
      version: func.currentVersion,
    });

    new codedeploy.LambdaDeploymentGroup(this, 'DeploymentGroup', {
      alias,
      deploymentConfig:
        codedeploy.LambdaDeploymentConfig.LINEAR_10PERCENT_EVERY_1MINUTE,
    });
  }
}
```

**JavaScript**

```javascript
File: lib/lambda-stack.js

const codedeploy = require('@aws-cdk/aws-codedeploy');
const lambda = require('@aws-cdk/aws-lambda');
const { Stack } = require('@aws-cdk/core');

class LambdaStack extends Stack {
  constructor(app, id, props) {
    super(app, id, props);
```
**Define Lambda stack**

```python
this.lambdaCode = lambda.Code.fromCfnParameters();

const func = new lambda.Function(this, 'Lambda', {
    code: this.lambdaCode,
    handler: 'index.main',
    runtime: lambda.Runtime.NODEJS_10_X
});

const alias = new lambda.Alias(this, 'LambdaAlias', {
    aliasName: 'Prod',
    version: func.currentVersion
});

new codedeploy.LambdaDeploymentGroup(this, 'DeploymentGroup', {
    alias,
    deploymentConfig: codedeploy.LambdaDeploymentConfig.LINEAR_10PERCENT_EVERY_1MINUTE
});

module.exports = { LambdaStack }
```

**Python**

File: `pipeline/lambda_stack.py`

```python
from aws_cdk import core, aws_codedeploy as codedeploy, aws_lambda as lambda_

class LambdaStack(core.Stack):
    def __init__(self, app: core.App, id: str, **kwargs):
        super().__init__(app, id, **kwargs)
        self.lambda_code = lambda_.Code.from_cfn_parameters()
        func = lambda_.Function(self, "Lambda",
                                code=self.lambda_code,
                                handler="index.main",
                                runtime=lambda_.Runtime.NODEJS_10_X,
                                )

        alias = lambda_.Alias(self, "LambdaAlias", alias_name="Prod",
                               version=func.current_version)

        codedeploy.LambdaDeploymentGroup(self, "DeploymentGroup",
                                         alias=alias,
                                         deployment_config= codedeploy.LambdaDeploymentConfig.LINEAR_10PERCENT_EVERY_1_MINUTE
                                         )

    module.exports = { LambdaStack }
```

**Java**

File: `src/main/java/com/myorg/LambdaStack.java`

```java
package com.myorg;

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

import software.amazon.awscdk.services.codedeploy.*;
import software.amazon.awscdk.services.lambda.*;
import software.amazon.awscdk.services.lambda.Runtime;
```
public class LambdaStack extends Stack {

    // private attribute to hold our Lambda's code, with public getters
    private CfnParametersCode lambdaCode;

    public CfnParametersCode getLambdaCode() {
        return lambdaCode;
    }

    // Constructor without props argument
    public LambdaStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public LambdaStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        lambdaCode = CfnParametersCode.fromCfnParameters();

        Function func = Function.Builder.create(this, "Lambda")
            .code(lambdaCode)
            .handler("index.main")
            .runtime(Runtime.NODEJS_10_X).build();

        Version version = func.getCurrentVersion();
        Alias alias = Alias.Builder.create(this, "LambdaAlias")
            .aliasName("LambdaAlias")
            .version(version).build();

        LambdaDeploymentGroup.Builder.create(this, "DeploymentGroup")
            .alias(alias)
            .deploymentConfig(LambdaDeploymentConfig.LINEAR_10_PERCENT_EVERY_1_MINUTE).build();
    }
}

C#

File: src/Pipeline/LambdaStack.cs

using Amazon.CDK;
using Amazon.CDK.AWS.CodeDeploy;
using Amazon.CDK.AWS.Lambda;

namespace Pipeline
{
    public class LambdaStack : Stack
    {
        public readonly CfnParametersCode lambdaCode;

        public LambdaStack(Construct scope, string id, StackProps props = null) :
            base(scope, id, props)
        {
            lambdaCode = Code.FromCfnParameters();

            var func = new Function(this, "Lambda", new FunctionProps
            {
                Code = lambdaCode,
                Handler = "index.main",
                Runtime = Runtime.NODEJS_10_X
            });

            var version = func.currentVersion;
            var alias = new Alias(this, "LambdaAlias", new AliasProps
            {
                version
Define pipeline stack

Our second stack, PipelineStack, contains the code that defines our pipeline.

First it needs a reference to the Lambda code it's deploying. For that, we define a new props interface for it, PipelineStackProps. This extends the standard StackProps and is how clients of this class (including ourselves) pass the Lambda code that the class needs.

The name of the CodeCommit repo hosting our source code is also passed in the stack's props. The Repository.fromRepositoryName method is a standard AWS CDK idiom for referencing a resource, such as a CodeCommit repository, that lives outside the AWS CDK code.

The pipeline has two CodeBuild projects. The first project synthesizes the AWS CloudFormation template to deploy the Lambda function from the AWS CDK code. To do that, it installs the AWS CDK Toolkit using npm, then any dependencies, and then issues cdk synth command to produce AWS CloudFormation templates in the target directory dist. The dist/LambdaStack.template.json file is this step's output.

The second project builds the Lambda code. It begins by changing the current directory to lambda, which is where the Lambda code lives. It then installs any dependencies and the TypeScript compiler, then builds the code. The output is the contents of the node_modules directory, plus the index.js file. The Lambda runtime will call the `handler()` function in this file to handle requests.

**Tip**
This is where you'll need some changes if you use a Lambda function written in a language other than TypeScript.

Finally, we define our pipeline. It has a source Action targeting the CodeCommit repository, two build Actions using the previously defined projects, and finally a deploy Action that uses AWS CloudFormation. It takes the template generated by the AWS CDK build Project (stored in the LambdaStack.template.json file), passes it to AWS CloudFormation for deployment. To make the Lambda build output an input to the AWS CloudFormation action, we pass it in the extraInputs property.

We also change the name of the stack that will be deployed, from LambdaStack to LambdaDeploymentStack. This isn't required; it's just an example of how you'd do this if you wanted.

TypeScript

File: lib/pipeline-stack.ts

```javascript
import * as codebuild from '@aws-cdk/aws-codebuild';
import * as codecommit from '@aws-cdk/aws-codecommit';
import * as codepipeline from '@aws-cdk/aws-codepipeline';
import * as codepipeline_actions from '@aws-cdk/aws-codepipeline-actions';
```
import * as lambda from '@aws-cdk/aws-lambda';
import * as s3 from '@aws-cdk/aws-s3';
import { App, Stack, StackProps } from '@aws-cdk/core';

export interface PipelineStackProps extends StackProps {
  readonly lambdaCode: lambda.CfnParametersCode;
  readonly repoName: string
}

export class PipelineStack extends Stack {
  constructor(app: App, id: string, props: PipelineStackProps) {
    super(app, id, props);
    const code = codecommit.Repository.fromRepositoryName(this, 'ImportedRepo', props.repoName);
    const cdkBuild = new codebuild.PipelineProject(this, 'CdkBuild', {
      buildSpec: codebuild.BuildSpec.fromObject({
        version: '0.2',
        phases: {
          install: {
            commands: 'npm install',
          },
          build: {
            commands: [
              'npm run build',
              'npm run cdk synth -- -o dist'
            ],
          },
        },
        artifacts: {
          'base-directory': 'dist',
          files: [
            'LambdaStack.template.json',
          ],
        },
      },
      environment: {
        buildImage: codebuild.LinuxBuildImage.STANDARD_2_0,
      },
    }));
    const lambdaBuild = new codebuild.PipelineProject(this, 'LambdaBuild', {
      buildSpec: codebuild.BuildSpec.fromObject({
        version: '0.2',
        phases: {
          install: {
            commands: [
              'cd lambda',
              'npm install',
            ],
          },
          build: {
            commands: 'npm run build',
          },
        },
        artifacts: {
          'base-directory': 'lambda',
          files: [
            'index.js',
            'node_modules/**/*',
          ],
        },
      },
      environment: {
        buildImage: codebuild.LinuxBuildImage.STANDARD_2_0,
      },
    });
  }
}
const sourceOutput = new codepipeline.Artifact();
const cdkBuildOutput = new codepipeline.Artifact('CdkBuildOutput');
const lambdaBuildOutput = new codepipeline.Artifact('LambdaBuildOutput');
new codepipeline.Pipeline(this, 'Pipeline', {
  stages: [
    {
      stageName: 'Source',
      actions: [
        new codepipeline_actions.CodeCommitSourceAction({
          actionName: 'CodeCommit_Source',
          repository: code,
          output: sourceOutput,
        }),
      ],
    },
    {
      stageName: 'Build',
      actions: [
        new codepipeline_actions.CodeBuildAction({
          actionName: 'Lambda_Build',
          project: lambdaBuild,
          input: sourceOutput,
          outputs: [lambdaBuildOutput],
        }),
        new codepipeline_actions.CodeBuildAction({
          actionName: 'CDK_Build',
          project: cdkBuild,
          input: sourceOutput,
          outputs: [cdkBuildOutput],
        }),
      ],
    },
    {
      stageName: 'Deploy',
      actions: [
        new codepipeline_actions.CloudFormationCreateUpdateStackAction({
          actionName: 'Lambda_CFN_Deploy',
          templatePath: cdkBuildOutput.atPath('LambdaStack.template.json'),
          stackName: 'LambdaDeploymentStack',
          adminPermissions: true,
          parameterOverrides: {
            ...props.lambdaCode.assign(lambdaBuildOutput.s3Location),
          },
          extraInputs: [lambdaBuildOutput],
        }),
      ],
    },
  ],
});

const codebuild = require('@aws-cdk/aws-codebuild');
const codecommit = require('@aws-cdk/aws-codecommit');
const codepipeline = require('@aws-cdk/aws-codepipeline');
const codepipeline_actions = require('@aws-cdk/aws-codepipeline-actions');
const { Stack } = require('@aws-cdk/core');
class PipelineStack extends Stack {
    constructor(app, id, props) {
        super(app, id, props);

        const code = codecommit.Repository.fromRepositoryName(this, 'ImportedRepo', props.repoName);

        const cdkBuild = new codebuild.PipelineProject(this, 'CdkBuild', {
            buildSpec: codebuild.BuildSpec.fromObject({
                version: '0.2',
                phases: {
                    install: {
                        commands: 'npm install'
                    },
                    build: {
                        commands: 'npm run cdk synth --o dist'
                    }
                },
                artifacts: {
                    'base-directory': 'dist',
                    files: [
                        'LambdaStack.template.json'
                    ]
                }
            }),
            environment: {
                buildImage: codebuild.LinuxBuildImage.STANDARD_2_0
            }
        });

        const lambdaBuild = new codebuild.PipelineProject(this, 'LambdaBuild', {
            buildSpec: codebuild.BuildSpec.fromObject({
                version: '0.2',
                phases: {
                    install: {
                        commands: [
                            'cd lambda',
                            'npm install',
                            'npm install typescript'
                        ],
                    },
                    build: {
                        commands: 'npx tsc index.ts'
                    }
                },
                artifacts: {
                    'base-directory': 'lambda',
                    files: [
                        'index.js',
                        'node_modules/**/*'
                    ]
                }
            }),
            environment: {
                buildImage: codebuild.LinuxBuildImage.STANDARD_2_0
            }
        });

        const sourceOutput = new codepipeline.Artifact();
        const cdkBuildOutput = new codepipeline.Artifact('CdkBuildOutput');
        const lambdaBuildOutput = new codepipeline.Artifact('LambdaBuildOutput');
        new codepipeline.Pipeline(this, 'Pipeline', {
            stages: [
                {
                    stageName: 'Source',
                    actions: [
                    ]
                },
                {n
                    stageName: 'CdkBuild'
                },
                {n
                    stageName: 'LambdaBuild'
                }
            ]
        });
Define pipeline stack

```python
new codepipeline_actions.CodeCommitSourceAction({
    actionName: 'CodeCommit_Source',
    repository: code,
    output: sourceOutput
})
},
{
    stageName: 'Build',
    actions: [
        new codepipeline_actions.CodeBuildAction({
            actionName: 'Lambda_Build',
            project: lambdaBuild,
            input: sourceOutput,
            outputs: [lambdaBuildOutput]
        }),
        new codepipeline_actions.CodeBuildAction({
            actionName: 'CDK_Build',
            project: cdkBuild,
            input: sourceOutput,
            outputs: [cdkBuildOutput]
        })
    ]
},
{
    stageName: 'Deploy',
    actions: [
        new codepipeline_actions.CloudFormationCreateUpdateStackAction({
            actionName: 'Lambda_CFN_Deploy',
            templatePath: cdkBuildOutput.atPath('LambdaStack.template.json'),
            stackName: 'LambdaDeploymentStack',
            adminPermissions: true,
            parameterOverides: {
                ...props.lambdaCode.assign(lambdaBuildOutput.s3Location)
            },
            extraInputs: [lambdaBuildOutput]
        })
    ]
}
});

module.exports = { PipelineStack }
```

```python
from aws_cdk import (core, aws_codebuild as codebuild,
                      aws_codecommit as codecommit,
                      aws_codepipeline as codepipeline,
                      aws_codepipeline_actions as codepipeline_actions,
                      aws_lambda as lambda_,
                      aws_s3 as s3)

class PipelineStack(core.Stack):
    def __init__(self, scope: core.Construct, id: str, *, repo_name: str=None,
                 lambda_code: lambda_.CfnParametersCode=None, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
        code = codecommit.Repository.from_repository_name(self, "ImportedRepo", repo_name)
```
Define pipeline stack

cdk_build = codebuild.PipelineProject(self, "CdkBuild",
    build_spec=codebuild.BuildSpec.from_object(dict(
        version="0.2",
        phases=dict(
            install=dict(
                commands=[
                    "npm install aws-cdk",
                    "npm update",
                    "python -m pip install -r requirements.txt"
                ]),
            build=dict(commands=[
                "npx cdk synth -o dist"]))),
        artifacts={
            "base-directory": "dist",
            "files": [
                "LambdaStack.template.json"],
        environment=dict(buildImage=codebuild.LinuxBuildImage.STANDARD_2_0))))

lambda_build = codebuild.PipelineProject(self, 'LambdaBuild',
    build_spec=codebuild.BuildSpec.from_object(dict(
        version="0.2",
        phases=dict(
            install=dict(
                commands=[
                    "cd lambda",
                    "npm install",
                    "npm install typescript"],
            build=dict(
                commands=[
                    "npx tsc index.ts"]))),
        artifacts={
            "base-directory": "lambda",
            "files": [
                "index.js",
                "node_modules/**/*"]},
        environment=dict(buildImage=codebuild.LinuxBuildImage.STANDARD_2_0))))

source_output = codepipeline.Artifact()

cdk_build_output = codepipeline.Artifact("CdkBuildOutput")

lambda_build_output = codepipeline.Artifact("LambdaBuildOutput")

lambda_location = lambda_build_output.s3_location

codepipeline.Pipeline(self, "Pipeline",
    stages=[
        codepipeline.StageProps(stage_name="Source",
            actions=[
                codepipeline_actions.CodeCommitSourceAction(
                    action_name="CodeCommit_Source",
                    repository=code,
                    output=source_output)),
        codepipeline.StageProps(stage_name="Build",
            actions=[
                codepipeline_actions.CodeBuildAction(
                    action_name="Lambda_Build",
                    project=lambda_build,
                    input=source_output,
                    outputs=[lambda_build_output]),
                codepipeline_actions.CodeBuildAction(
                    action_name="CDK_Build",
                    project=cdk_build,
                    input=source_output,
                    outputs=[cdk_build_output]))],
        codepipeline.StageProps(stage_name="Deploy",}
Define pipeline stack

```java
actions= [    codepipeline_actions.CloudFormationCreateUpdateStackAction(        action_name="Lambda_CFN_Deploy",        template_path=cdk_build_output.at_path("LambdaStack.template.json"),        stack_name="LambdaDeploymentStack",        admin_permissions=True,        parameter_overrides=dict(            lambda_code.assign(                bucket_name=lambda_location.bucket_name,                object_key=lambda_location.object_key,                object_version=lambda_location.object_version)),        extra_inputs=[lambda_build_output]))]

Java

File: src/main/java/com/myorg/PipelineStack.java

package com.myorg;
import java.util.Arrays;
import java.util.List;
import java.util.HashMap;
import software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.codebuild.*;
import software.amazon.awscdk.services.codecommit.*;
import software.amazon.awscdk.services.codepipeline.*;
import software.amazon.awscdk.services.codepipeline.StageProps;
import software.amazon.awscdk.services.codepipeline.actions.*;
import software.amazon.awscdk.services.lambda.*;
public class PipelineStack extends Stack {
    // alternate constructor for calls without props.
    // lambdaCode and repoName are both required.
    public PipelineStack(final App scope, final String id, final CfnParametersCode lambdaCode, final String repoName) {
        this(scope, id, null, lambdaCode, repoName);
    }

    @SuppressWarnings("serial")
    public PipelineStack(final App scope, final String id, final StackProps props, final CfnParametersCode lambdaCode, final String repoName) {
        super(scope, id, props);
        IRepository code = Repository.fromRepositoryName(this, "ImportedRepo", repoName);
        PipelineProject cdkBuild = PipelineProject.Builder.create(this, "CDKBuild")
            .buildSpec(BuildSpec.fromObject(new HashMap<String, Object>() {
                put("version", "0.2");
                put("phases", new HashMap<String, Object>() {
                    put("install", new HashMap<String, String>() {
                        put("commands", "npm install aws-cdk"),
                    });
                    put("build", new HashMap<String, Object>() {
                        put("commands", Arrays.asList("mvn compile -q -DskipTests",
                            "npx cdk synth -o dist")));
                });
            }));
    }
```
Define pipeline stack

```java
put("files", Arrays.asList("LambdaStack.template.json"));
})
.environment(BuildEnvironment.builder().buildImage(
  LinuxBuildImage.STANDARD_2_0).build())
.build();

PipelineProject lambdaBuild = PipelineProject.Builder.create(this,
  "LambdaBuild")
  .buildSpec(BuildSpec.fromObject(new HashMap<String, Object>() {
    put("version", "0.2");
    put("phases", new HashMap<String, Object>() {
      put("install", new HashMap<String, List<String>>() {
        put("commands", Arrays.asList("cd lambda", "npm
install", "npm install typescript");
      });
      put("build", new HashMap<String, List<String>>() {
        put("commands", Arrays.asList("npx tsc index.ts");
      });
    });
    put("artifacts", new HashMap<String, Object>() {
      put("base-directory", "lambda");
      put("files", Arrays.asList("index.js", "node_modules/**/
*"));
    });
  })
  .environment(BuildEnvironment.builder().buildImage(
    LinuxBuildImage.STANDARD_2_0).build())
  .build();

Artifact sourceOutput = new Artifact();
Artifact cdkBuildOutput = new Artifact("CdkBuildOutput");
Artifact lambdaBuildOutput = new Artifact("LambdaBuildOutput");

Pipeline.Builder.create(this, "Pipeline")
  .stages(Arrays.asList(
    StageProps.builder()
      .stageName("Source")
      .actions(Arrays.asList(
        CodeCommitSourceAction.Builder.create()
          .actionName("Source")
          .repository(code)
          .output(sourceOutput)
          .build()))
      .build(),
    StageProps.builder()
      .stageName("Build")
      .actions(Arrays.asList(
        CodeBuildAction.Builder.create()
          .actionName("Lambda_Build")
          .project(lambdaBuild)
          .input(sourceOutput)
          .outputs(Arrays.asList(lambdaBuildOutput)).build(),
        CodeBuildAction.Builder.create()
          .actionName("CDK_Build")
          .project(cdkBuild)
          .input(sourceOutput)
          .outputs(Arrays.asList(cdkBuildOutput)).build())
      .build(),
    StageProps.builder()
      .stageName("Deploy")
      .actions(Arrays.asList(
        CloudFormationCreateUpdateStackAction.Builder.create()
          .actionName("Lambda_CFN_Deploy")
```
Define pipeline stack

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.CodeBuild;
using Amazon.CDK.AWS.CodeCommit;
using Amazon.CDK.AWS.CodePipeline;
using Amazon.CDK.AWS.CodePipeline.Actions;
using Amazon.CDK.AWS.Lambda;
using System.Collections.Generic;

namespace Pipeline
{
    public class PipelineStackProps : StackProps
    {
        public CfnParametersCode LambdaCode { get; set; }
        public string RepoName { get; set; }
    }

    public class PipelineStack : Stack
    {
        public PipelineStack(Construct scope, string id, PipelineStackProps props = null) :
            base(scope, id, props)
        {
            var code = Repository.FromRepositoryName(this, "ImportedRepo", props.RepoName);
            var cdkBuild = new PipelineProject(this, "CDKBuild", new PipelineProjectProps
            {
                BuildSpec = BuildSpec.FromObject(new Dictionary<string, object>
                {
                    ["version"] = "0.2",
                    ["phases"] = new Dictionary<string, object>
                    {
                        ["install"] = new Dictionary<string, object>
                        {
                            ["commands"] = "npm install aws-cdk"
                        },
                        ["build"] = new Dictionary<string, object>
                        {
                            ["commands"] = "npx cdk synth -o dist"
                        }
                    },
                    ["artifacts"] = new Dictionary<string, object>
                    {
                        ["base-directory"] = "dist",
                        ["files"] = new string[]
                        {
                            "LambdaStack.template.json"
                        }
                    }
                }
            });
        }
    }
}
```
Define pipeline stack

```typescript
Environment = new BuildEnvironment
{
    BuildImage = WindowsBuildImage.WINDOWS_BASE_2_0
}

var lambdaBuild = new PipelineProject(this, "LambdaBuild", new
PipelineProjectProps
{
    BuildSpec = BuildSpec.FromObject(new Dictionary<string, object>
    {
        ["version"] = "0.2",
        ["phases"] = new Dictionary<string, object>
        {
            ["install"] = new Dictionary<string, object>
            {
                ["commands"] = new string[
                {
                    "cd lambda",
                    "npm install",
                    "npm install typescript"
                }
            },
            ["build"] = new Dictionary<string, string>
            {
                ["commands"] = "npx tsc index.ts"
            }
        },
        ["artifacts"] = new Dictionary<string, object>
        {
            ["base-directory"] = "lambda",
            ["files"] = new string[
            {
                "index.js",
                "node_modules/**/*"
            }
        }
    }
},
Environment = new BuildEnvironment
{
    BuildImage = LinuxBuildImage.STANDARD_2_0
}

var sourceOutput = new Artifact_();
var cdkBuildOutput = new Artifact_("CdkBuildOutput");
var lambdaBuildOutput = new Artifact_("LambdaBuildOutput");

new Amazon.CDK.AWS.CodePipeline.Pipeline(this, "Pipeline", new
PipelineProps
{
    Stages = new[]
    {
        new Amazon.CDK.AWS.CodePipeline.StageProps
        {
            StageName = "Source",
            Actions = new []
            {
                new CodeCommitSourceAction(new CodeCommitSourceActionProps
                {
                    ActionName = "Source",
                    Repository = code,
                    Output = sourceOutput
                }
            }
        }
    }
})
```

Main program

Finally, we have our main AWS CDK application file.

Note
If you didn’t name your new CodeCommit repository pipeline, here’s where you’d change it. Just edit the value of CODECOMMIT_REPO_NAME.

This code is straightforward: it first instantiates the LambdaStack class as LambdaStack, which is what the AWS CDK build in the pipeline expects. Then it instantiates the PipelineStack class, passing the Lambda code from the LambdaStack object.

TypeScript

File: bin/pipeline.ts
#!/usr/bin/env node

const CODECOMMIT_REPO_NAME = "pipeline";

import { App } from '@aws-cdk/core';
import { LambdaStack } from '../lib/lambda-stack';
import { PipelineStack } from '../lib/pipeline-stack';

const app = new App();

const lambdaStack = new LambdaStack(app, "LambdaStack");
new PipelineStack(app, 'PipelineDeployingLambdaStack', {
  lambdaCode: lambdaStack.lambdaCode,
  repoName: CODECOMMIT_REPO_NAME
});

app.synth();

JavaScript
File: bin/pipeline.js

#!/usr/bin/env node

const CODECOMMIT_REPO_NAME = "pipeline";

const { App } = require('@aws-cdk/core');
const { LambdaStack } = require('../lib/lambda-stack');
const { PipelineStack } = require('../lib/pipeline-stack');

const app = new App();

const lambdaStack = new LambdaStack(app, 'LambdaStack');
new PipelineStack(app, 'PipelineDeployingLambdaStack', {
  lambdaCode: lambdaStack.lambdaCode,
  repoName: CODECOMMIT_REPO_NAME
});

app.synth();

Python
File: app.py

#!/usr/bin/env python3

CODECOMMIT_REPO_NAME = "pipeline"
from aws_cdk import core
from pipeline.pipeline_stack import PipelineStack
from pipeline.lambda_stack import LambdaStack

app = core.App()

lambda_stack = LambdaStack(app, "LambdaStack")
PipelineStack(app, "PipelineDeployingLambdaStack",
  lambda_code=lambda_stack.lambda_code,
  repo_name=CODECOMMIT_REPO_NAME)

app.synth()
Deploying the pipeline

Now we can deploy the pipeline.

1. Check the code into Git and push it to AWS CodeCommit.

   ```
   git add --all
   git commit -m "add CDK app"
   git push
   ```
The name, `PipelineDeployingLambdaStack`, is the name we used when we instantiated `PipelineStack`.

**Tip**
Rather than typing that whole name out, this is a good place to use a wildcard! Put quotes around the name pattern to prevent the shell from trying to expand it.

```
cdk deploy "Pipe*"
```

You'll be asked to approve your stack's security changes. Type `y` to accept them and continue with deployment.

Don't deploy `LambdaStack`. This stack is deployed by the pipeline, and it won't deploy without values provided by the pipeline.

After the deployment finishes, you should have a three-stage pipeline that looks something like the following.
Cleaning up

Try making a change to your Lambda function code and push it to the repository. The pipeline should pick up your change, build it, and deploy it automatically, without any other action from you.

Cleaning up

To avoid unexpected AWS charges, destroy your AWS CDK stacks after you're done with this exercise.

Delete the LambdaStack first using the AWS CloudFormation console. The IAM role needed to delete LambdaStack is provided by PipelineDeployingLambdaStack, so if you delete it first, you no longer have permission to destroy LambdaStack.

Then you may delete the PipelineDeployingLambdaStack.
cdk destroy LambdaStack
cdk destroy PipelineDeployingLambdaStack

Finally, delete your AWS CodeCommit repository from the AWS Console.

**AWS CDK examples**

For more examples of AWS CDK stacks and apps in your favorite supported programming language, see:

- The [CDK Examples repository on GitHub](#)
- The [AWS Code Example Repository](#)
AWS CDK how-tos

This section contains short code examples that show you how to accomplish a task using the AWS CDK.

Get a value from an environment variable

To get the value of an environment variable, use code like the following. This code gets the value of the environment variable `MYBUCKET`.

**TypeScript**

```typescript
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

**JavaScript**

```javascript
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.MYBUCKET || "DefaultName";
```

**Python**

```python
import os

# Raises KeyError if environment variable doesn't exist
bucket_name = os.environ["MYBUCKET"]

# Sets bucket_name to None if environment variable doesn't exist
bucket_name = os.getenv("MYBUCKET")

# Sets bucket_name to a default if env var doesn't exist
bucket_name = os.getenv("MYBUCKET", "DefaultName")
```

**Java**

```java
// Sets bucketName to null if environment variable doesn't exist
String bucketName = System.getenv("MYBUCKET");

// Sets bucketName to a default if env var doesn't exist
String bucketName = System.getenv("MYBUCKET");
if (bucketName == null) bucketName = "DefaultName";
```

**C#**

```csharp
using System;

// Sets bucket name to null if environment variable doesn't exist
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET");

// Sets bucket_name to a default if env var doesn't exist
```
string bucketName = Environment.GetEnvironmentVariable("MYBUCKET") ?? "DefaultName";

Use an AWS CloudFormation parameter

See Parameters for information about using the optional Parameters section to customize your AWS CloudFormation templates.

You can also get a reference to a resource in an existing AWS CloudFormation template, as described in the section called “Use CloudFormation template” (p. 204).

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:

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

You can include this bucket in your AWS CDK app, as shown in the following example.

TypeScript

```typescript
import * as cdk from '@aws-cdk/core';
import * as fs from 'fs';
new cdk.CfnInclude(this, "ExistingInfrastructure", {
  template: JSON.parse(fs.readFileSync("my-template.json").toString())
});
```

JavaScript

```javascript
const cdk = require('@aws-cdk/core');
const fs = require("fs");
new cdk.CfnInclude(this, "ExistingInfrastructure", {
  template: JSON.parse(fs.readFileSync("my-template.json").toString())
});
```

Python

```python
import json

cdk.CfnInclude(self, "ExistingInfrastructure",
    template=json.load(open("my-template.json")))
```

Java

```java
import java.util.*;
```
Use CloudFormation template

```java
import java.io.File;
import software.amazon.awscdk.core.CfnInclude;

import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

CfnInclude.Builder.create(this, "ExistingInfrastructure")
  .template((ObjectNode)new ObjectMapper().readTree(new File("my-template.json")))
  .build();
```

```csharp
using Newtonsoft.Json.Linq;

new CfnInclude(this, "ExistingInfrastructure", new CfnIncludeProps
{
   Template = JObject.Parse(File.ReadAllText("my-template.json"))
});
```

Then to access an attribute of the resource, such as the bucket's ARN, call `Fn.getAtt()` with the logical name of the resource in the AWS CloudFormation template and the desired attribute of the resource. (The resource must be defined in the template; `Fn.getAtt()` does not query actual resources you have deployed using the template.

**TypeScript**

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

**JavaScript**

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

**Python**

```python
bucket_arn = cdk.Fn.get_att("S3Bucket", "Arn")
```

**Java**

```java
IResolvable bucketArn = Fn.getAtt("S3Bucket", "Arn");
```

**C#**

```csharp
var bucketArn = Fn.GetAtt("S3Bucket", "Arn");
```

The result of a `getAtt()` call is a token (p. 105), a type of placeholder. The actual value of the attribute isn't available until later in the synthesis process. If you need to pass such an attribute to another API that requires a concrete value, such as a string or a number, use the following static methods of the `Token` class to convert the token to a string, number, or list.

- `Token.asString` to generate a string encoding (or call `.toString()` on the token object)
- `Token.asList` to generate a list encoding
- `Token.asNumber` to generate a numeric encoding
In our example of getting a bucket's ARN, you'd convert it to a string, but that string is still a token, just in a string encoding. You still don't have the actual ARN. But in many ways, you can treat the string as if you did have the real value (for example, adding text to the beginning or end) and it will work as you expect.

Get a value from the Systems Manager Parameter Store

The AWS CDK can retrieve the value of AWS Systems Manager Parameter Store attributes. During synthesis, the AWS CDK produces a token (p. 105) that is resolved by AWS CloudFormation during deployment.

The AWS CDK supports retrieving both plain and secure values. You may request a specific version of either kind of value. For plain values only, you may omit the version from your request to receive the latest version. You must always specify the version when requesting the value of a secure attribute.

**Note**

This topic shows how to read attributes from the AWS Systems Manager Parameter Store. You can also read secrets from the AWS Secrets Manager (see Get a value from AWS Secrets Manager (p. 208)).

Reading Systems Manager values at deployment time

To read values from the Systems Manager Parameter Store, use the `valueForStringParameter` and `valueForSecureStringParameter` methods, depending on whether the attribute you want is a plain string or a secure string value. These methods return tokens (p. 105), not the actual value. The value is resolved by AWS CloudFormation during deployment.

**TypeScript**

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

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(this, 'my-secure-parameter-name', 1);   // must specify version
```

**JavaScript**

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

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(this, 'my-secure-parameter-name', 1);   // must specify version
```
Reading Systems Manager values at synthesis time

It is sometimes useful to "bake in" a parameter at synthesis time, so that the resulting AWS CloudFormation template always uses the same value, rather than resolving the value during deployment.

To read a value from the Systems Manager parameter store at synthesis time, use the `valueFromLookup` method (Python: `value_from_lookup`). This method returns the actual value of the parameter as a `valueFromLookup` value. If the value is not already cached in `cdk.json` or passed on the command line, it will be retrieved from the current AWS account. For this reason, the stack must be synthesized with explicit account and region information.

TypeScript

```typescript
import * as ssm from '@aws-cdk/aws-ssm';
```
const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-name');

**JavaScript**

```javascript
const ssm = require('@aws-cdk/aws-ssm');
const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-name');
```

**Python**

```python
import aws_cdk.aws_ssm as ssm
string_value = ssm.StringParameter.value_from_lookup(self, "my-plain-parameter-name")
```

**Java**

```java
import software.amazon.awscdk.services.ssm.StringParameter;
String stringValue = StringParameter.valueFromLookup(this, "my-plain-parameter-name");
```

**C#**

```csharp
using Amazon.CDK.AWS.SSM;
var stringValue = StringParameter.ValueFromLookup(this, "my-plain-parameter-name");
```

Only plain Systems Manager strings may be retrieved, not secure strings. It is not possible to request a specific version; the latest version is always returned.

## Writing values to Systems Manager

You can use the AWS CLI, the AWS Management Console, or an AWS SDK to set Systems Manager parameter values. The following examples use the `ssm put-parameter` CLI command.

```
aws ssm put-parameter --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --name "secure-parameter-name" --type "SecureString" --value "secure-parameter-value"
```

When updating an SSM value that already exists, also include the `--overwrite` option.

```
aws ssm put-parameter --overwrite --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --overwrite --name "secure-parameter-name" --type "SecureString" --value "secure-parameter-value"
```

## Get a value from AWS Secrets Manager

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

```typescript
import * as sm from '@aws-cdk/aws-secretsmanager';

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

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

**JavaScript**

```javascript
const sm = require('@aws-cdk/aws-secretsmanager');

class SecretsManagerStack extends core.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

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

**Python**

```python
import aws_cdk.aws_secretsmanager as sm

class SecretsManagerStack(core.Stack):
  def __init__(self, scope: core.App, id: str, **kwargs):
    super().__init__(scope, name, **kwargs)

    secret = sm.Secret.from_secret_attributes(self, "ImportedSecret",
      secret_arn="arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-<random-6-characters>",
      # If the secret is encrypted using a KMS-hosted CMK, either import or reference that key:
      # encryption_key=....
    )
```

**Java**

```java
import software.amazon.awscdk.services.secretsmanager.Secret;
import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

class SecretsManagerStack extends Stack {
  public SecretsManagerStack(App scope, String id) {
```

---

209
Create an app with multiple stacks

Most of the other code examples in the AWS CDK Developer Guide involve only a single stack. However, you can create apps containing any number of stacks. Each stack results in its own AWS CloudFormation template. Stacks are the unit of deployment: each stack in an app can be synthesized and deployed individually using the cdk deploy command.

This topic illustrates how to extend the Stack class to accept new properties or arguments, how to use these properties to affect what resources the stack contains and their configuration, and how to instantiate multiple stacks from this class. The example uses a Boolean property, named encryptBucket (Python: encrypt_bucket), to indicate whether an Amazon S3 bucket should be encrypted. If so, the stack enables encryption using a key managed by AWS Key Management Service (AWS KMS). The app creates two instances of this stack, one with encryption and one without.
Before you begin

First, install Node.js and the AWS CDK command line tools, if you haven't already. See Getting started with the AWS CDK (p. 9) for details.

Next, create an AWS CDK project by entering the following commands at the command line.

TypeScript

```bash
mkdir multistack
cd multistack
cdk init --language=typescript
```

JavaScript

```bash
mkdir multistack
cd multistack
cdk init --language=javascript
```

Python

```bash
mkdir multistack
cd multistack
cdk init --language=python
source ./env/bin/activate
pip install -r requirements.txt
```

Java

```bash
mkdir multistack
cd multistack
cdk init --language=java
```

You can import the resulting Maven project into your Java IDE.

C#

```bash
mkdir multistack
cd multistack
cdk init --language=csharp
```

You can open the file `src/Pipeline.sln` in Visual Studio.

Finally, install the core and s3 AWS Construct Library modules. We use these modules in our app.

TypeScript

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

JavaScript

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

Python

```bash
pip install aws_cdk.core aws_cdk.aws_s3
```
Add optional parameter

The `props` argument of the `Stack` constructor fulfills the interface `StackProps`. Because we want our stack to accept an additional property to tell us whether to encrypt the Amazon S3 bucket, we should create an interface or class that includes that property. This allows the compiler to make sure the property has a Boolean value and enables autocompletion for it in your IDE.

So open the indicated source file in your IDE or editor and add the new interface, class, or argument. The code should look like this after the changes. The lines we added are shown in boldface.

**TypeScript**

```typescript
File: lib/multistack-stack.ts

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

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

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

    // The code that defines your stack goes here
  }
}
```

**JavaScript**

```javascript
File: lib/multistack-stack.js

```JavaScript doesn't have an interface feature; we don't need to add any code.

```

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

class MultistackStack extends cdk.Stack {
```
Add optional parameter

```javascript
constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your stack goes here
}
}

module.exports = { MultistackStack }
```

Python

File: multistack/multistack_stack.py

Python does not have an interface feature, so we'll extend our stack to accept the new property by adding a keyword argument.

```python
from aws_cdk import aws_s3 as s3

class MultistackStack(core.Stack):
    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: core.Construct, id: str, *, encrypt_bucket=False, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines your stack goes here
```

Java

File: src/main/java/com/myorg/MultistackStack.java

It's more complicated than we really want to get into to extend a props type in Java, so we'll simply write our stack's constructor to accept an optional Boolean parameter. Since `props` is an optional argument, we'll write an additional constructor that allows you to skip it. It will default to `false`.

```java
package com.myorg;

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

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id, boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

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

    public MultistackStack(final Construct scope, final String id, final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // The code that defines your stack goes here
    }
}
```
The new property is optional. If encryptBucket (Python: encrypt_bucket) is not present, its value is undefined, or the local equivalent. The bucket will be unencrypted by default.

Define the stack class

Now let's define our stack class, using our new property. Make the code look like the following. The code you need to add or change is shown in boldface.

TypeScript

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

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

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

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses AWS KMS-managed keys (SSE-KMS).
    if (props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    }
  }
}
```
Define the stack class

JavaScript
File: lib/multistack-stack.js

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

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

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
        // Encrypted bucket uses AWS KMS-managed keys (SSE-KMS).
        if (props && props.encryptBucket) {
            new s3.Bucket(this, "MyGroovyBucket", {
                encryption: s3.BucketEncryption.KMS_MANAGED,
                removalPolicy: cdk.RemovalPolicy.DESTROY
            });
        } else {
            new s3.Bucket(this, "MyGroovyBucket", {
                removalPolicy: cdk.RemovalPolicy.DESTROY
            });
        }
    }
}

module.exports = { MultistackStack }
```

Python
File: multistack/multistack_stack.py

```python
from aws_cdk import core
from aws_cdk import aws_s3 as s3

class MultistackStack(core.Stack):
    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: core.Construct, id: str, *, encrypt_bucket=False, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add a Boolean property "encryptBucket" to the stack constructor.
        # If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
        # Encrypted bucket uses AWS KMS-managed keys (SSE-KMS).
        if encrypt_bucket:
            s3.Bucket(self, "MyGroovyBucket",
                        encryption=s3.BucketEncryption.KMS_MANAGED,
                        removal_policy=core.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                        removal_policy=core.RemovalPolicy.DESTROY)
```

Java
File: src/main/java/com/myorg/MultistackStack.java

```java
package com.myorg;
```
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.RemovalPolicy;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id,
        boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

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

    // main constructor
    public MultistackStack(final Construct scope, final String id,
        final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is
        // unencrypted. Encrypted bucket uses AWS KMS-managed keys (SSE-KMS).
        if (encryptBucket) {
            new Bucket.Builder.create(this, "MyGroovyBucket")
              .encryption(BucketEncryption.KMS_MANAGED)
              .removalPolicy(RemovalPolicy.DESTROY).
              build();
        } else {
            new Bucket.Builder.create(this, "MyGroovyBucket")
              .removalPolicy(RemovalPolicy.DESTROY).
              build();
        }
    }
}

C#

File: src/Multistack/MultistackStack.cs

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

namespace Multistack
{
    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }

    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, IMultiStackProps props = null) : base(scope, id, props)
        {
            // Add a Boolean property "EncryptBucket" to the stack constructor.
            // If true, creates an encrypted bucket. Otherwise, the bucket is
            // unencrypted. Encrypted bucket uses AWS KMS-managed keys (SSE-KMS).
            if (props?.EncryptBucket ?? false)
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
Create two stack instances

Now we'll add the code to instantiate two separate stacks. As before, the lines of code shown in boldface are the ones you need to add. Delete the existing `MultistackStack` definition.

TypeScript

```
File: bin/multistack.ts

#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from '@aws-cdk/core';
import { MultistackStack } from '../lib/multistack-stack';
const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

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

JavaScript

```
File: bin/multistack.js

#!/usr/bin/env node
const cdk = require('@aws-cdk/core');
const { MultistackStack } = require('../lib/multistack-stack');
const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});
```
Create two stack instances

Python

File: ./app.py

```python
#!/usr/bin/env python3
from aws_cdk import core
from multistack.multistack_stack import MultistackStack

app = core.App()
MultistackStack(app, "MyWestCdkStack",
    env=core.Environment(region="us-west-1"),
    encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
    env=core.Environment(region="us-east-1"),
    encrypt_bucket=True)
```

Java

File: src/main/java/com/myorg/MultistackApp.java

```java
package com.myorg;
import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Environment;
import software.amazon.awscdk.core.StackProps;

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

        new MultistackStack(app, "MyWestCdkStack", StackProps.builder()
            .env(Environment.builder()
                .region("us-west-1")
                .build())
            .build(), false);

        new MultistackStack(app, "MyEastCdkStack", StackProps.builder()
            .env(Environment.builder()
                .region("us-east-1")
                .build())
            .build(), true);

        app.synth();
    }
}
```

C#

File: src/Multistack/Program.cs

```csharp
using Amazon.CDK;

namespace Multistack
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
```
This code uses the new encryptBucket (Python: encrypt_bucket) property on the MultistackStack class to instantiate the following:

- One stack with an encrypted Amazon S3 bucket in the `us-east-1` AWS Region.
- One stack with an unencrypted Amazon S3 bucket in the `us-west-1` AWS Region.

### Synthesize and deploy the stack

Now you can deploy stacks from the app. First, synthesize a AWS CloudFormation template for `MyEastCdkStack`—the stack in `us-east-1`. This is the stack with the encrypted S3 bucket.

```bash
$ cdk synth MyEastCdkStack
```

The output should look similar to the following AWS CloudFormation template (there might be slight differences).

```yaml
Resources:
  MyGroovyBucketFD9882AC:
    Type: AWS::S3::Bucket
    Properties:
      BucketEncryption:
        ServerSideEncryptionConfiguration:
          - ServerSideEncryptionByDefault:
            SSEAlgorithm: aws:kms
      UpdateReplacePolicy: Retain
      DeletionPolicy: Retain
      Metadata:
        aws:cdk:path: MyEastCdkStack/MyGroovyBucket/Resource
      CDKMetadata:
        Type: AWS::CDK::Metadata
        Properties:
          Modules:
            - aws-cdk=1.10.0,aws-cdk/aws-events=1.10.0,aws-cdk/aws-iam=1.10.0,aws-cdk/aws-kms=1.10.0,aws-cdk/aws-s3=1.10.0,aws-cdk/core=1.10.0,aws-cdk/cx-api=1.10.0,aws-cdk/region-info=1.10.0,jsii-runtime=node.js/v10.16.2

To deploy this stack to your AWS account, issue one of the following commands. The first command uses your default AWS profile to obtain the credentials to deploy the stack. The second uses a profile you specify: for `PROFILE_NAME`, substitute the name of an AWS CLI profile that contains appropriate credentials for deploying to the `us-east-1` AWS Region.

```bash
cdk deploy MyEastCdkStack
```
Clean up

To avoid charges for resources that you deployed, destroy the stack using the following command.

```bash
cdk destroy MyEastCdkStack
```

The destroy operation fails if there is anything stored in the stack’s bucket. There shouldn’t be if you’ve only followed the instructions in this topic. But if you did put something in the bucket, you must delete the bucket’s contents, but not the bucket itself, using the AWS Management Console or the AWS CLI before destroying the stack.

Set a CloudWatch alarm

The `aws-cloudwatch` package supports setting CloudWatch alarms on CloudWatch metrics. So the first thing you need is a metric. You can use a predefined metric or you can create your own.

Using an existing metric

Many AWS Construct Library modules let you set an alarm on an existing metric by passing the metric’s name to a convenience method on an instance of an object that has metrics. For example, given an Amazon SQS queue, you can get the metric `ApproximateNumberOfMessagesVisible` from the queue’s `metric()` method.

**TypeScript**

```typescript
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

**JavaScript**

```javascript
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

**Python**

```python
metric = queue.metric("ApproximateNumberOfMessagesVisible")
```

**Java**

```java
Metric metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

**C#**

```csharp
var metric = queue.Metric("ApproximateNumberOfMessagesVisible");
```

Creating your own metric

Create your own `metric` as follows, where the `namespace` value should be something like `AWS/SQS` for an Amazon SQS queue. You also need to specify your metric’s name and dimension.
Creating the alarm

Once you have a metric, either an existing one or one you defined, you can create an alarm. In this example, the alarm is raised when there are more than 100 of your metric in two of the last three seconds. Assuming the metric is the `ApproximateNumberOfMessagesVisible` metric from an Amazon SQS queue, it would raise when 100 messages are visible in the queue in two of the last three seconds.

TypeScript

```typescript
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
});
```
Creating the alarm

evaluationPeriods: 3,
datapointsToAlarm: 2,
});

### JavaScript

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

### Python

```python
alarm = cloudwatch.Alarm(self, "Alarm",
    metric=metric,
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

### Java

```java
import software.amazon.awscdk.services.cloudwatch.Alarm;
import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
    .metric(metric)
    .threshold(100)
    .evaluationPeriods(3)
    .datapointsToAlarm(2).build();
```

### C#

```csharp
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
    Metric = metric,
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

An alternative way to create an alarm is using the metric's `createAlarm()` method, which takes essentially the same properties as the `Alarm` constructor; you just don't need to pass in the metric, since it's already known.

### TypeScript

```typescript
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

### JavaScript

```javascript
metric.createAlarm(this, 'Alarm', {
```
Get context value

Python

```python
def metric.create_alarm(self, "Alarm",
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
):
```

Java

```java
metric.createAlarm(this, "Alarm", new CreateAlarmOptions.Builder()
    .threshold(100)
    .evaluationPeriods(3)
    .datapointsToAlarm(2)
    .build());
```

C#

```csharp
metric.CreateAlarm(this, "Alarm", new CreateAlarmOptions
{
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

Get a value from a context variable

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

To create a command line context variable, use the `--context (-c)` option, as shown in the following example.

```
cdk synth -c bucket_name=mygroovybucket
```

To specify the same context variable and value in the `cdk.json` file, use the following code.

```json
{
    "context": {
        "bucket_name": "myotherbucket"
    }
}
```

To get the value of a context variable in your app, use the `TryGetContext` method in the context of a construct (that is, when `this`, or `self` in Python, is an instance of some construct). The example gets the context value `bucket_name`. If the requested value is not defined, `TryGetContext` returns undefined (None in Python; null in Java and C#) rather than raising an exception.

TypeScript

```typescript
const bucket_name = this.node.tryGetContext('bucket_name');
```
Create CDK Pipeline

Outside the context of a construct, you can access the context variable from the app object, like this.

**TypeScript**

```typescript
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

**JavaScript**

```javascript
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

**Python**

```python
app = cdk.App()
bucket_name = app.node.try_get_context("bucket_name")
```

**Java**

```java
App app = App();
String bucketName = (String)app.getNode().tryGetContext("bucket_name");
```

**C#**

```csharp
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

For more details on working with context variables, see the section called “Context” (p. 142).

Continuous integration and delivery (CI/CD) using CDK Pipelines

**CDK Pipelines** is a construct library module for painless continuous delivery of AWS CDK applications. Whenever you check your AWS CDK app's source code in to AWS CodeCommit, GitHub, or BitBucket, CDK Pipelines can automatically build, test, and deploy your new version.
CDK Pipelines are self-updating: if you add new application stages or new stacks, the pipeline automatically reconfigures itself to deploy those new stages and/or stacks.

If you’ve looked at our AWS CodePipeline example (p. 180), CDK Pipelines can do everything that example does, and more, with less code. Going forward, we anticipate widespread adoption of CDK Pipelines by AWS CDK users.

**Note**

CDK Pipelines is currently in developer preview, and its API is subject to change. Breaking API changes will be announced in the AWS CDK Release Notes.

**Bootstrap your AWS environments**

Before you can use CDK Pipelines, you must bootstrap the AWS environment(s) to which you will deploy your stacks. An environment (p. 77) is an account/region pair to which you want to deploy a CDK stack. A CDK Pipeline involves at least two environments: the environment where the pipeline is provisioned, and the environment where you want to deploy the application’s stacks (or its stages, which are groups of related stacks). These environments can be the same, though best practices recommend you isolate stages from each other in different AWS accounts or regions.

You may have already bootstrapped one or more environments so you can deploy assets and Lambda functions using the AWS CDK. Continuous deployment with CDK Pipelines requires that the CDK Toolkit stack include additional resources, so the stack has been extended to include an additional Amazon S3 bucket, an Amazon ECR repository, and IAM roles to give the various parts of a pipeline the permissions they need. This new style of CDK Toolkit stack will eventually become the default, but at this writing, you must opt in. The AWS CDK Toolkit will upgrade your existing bootstrap stack or create a new one, as necessary.

To bootstrap an environment that will provision a pipeline:

**Mac OS X/Linux**

```bash
CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap --profile ADMIN-PROFILE \
  --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess \
  aws://ACCOUNT-ID/REGION
```

**Windows**

```bash
set CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap --profile ADMIN-PROFILE ^
  --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess ^
  aws://ACCOUNT-ID/REGION
```

To bootstrap additional environments into which AWS CDK applications will be deployed by the pipeline:

**Mac OS X/Linux**

```bash
CDK_NEW_BOOTSTRAP=1
npx cdk bootstrap --profile ADMIN-PROFILE \
  --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess \
  --trust PIPELINE-ACCOUNT-ID \
  aws://ACCOUNT-ID/REGION
```

**Windows**

```bash
set CDK_NEW_BOOTSTRAP=1
```
**Initialize project**

Create a new, empty GitHub project and clone it to your workstation in the `cdk-pipeline` directory. (Our code examples in this topic use GitHub; you can also use BitBucket or AWS CodeCommit.)

```bash
git clone GITHUB-CLONE-URL my-pipeline
cd my-pipeline
```

**Note**

You may use a name other than `my-pipeline` for your app's main directory, but since the AWS CDK Toolkit bases some file and class names on the name of the main directory, you'll need to tweak these later in this topic.

After cloning, initialize the project as usual.

**TypeScript**

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

**JavaScript**

```bash
cdk init app --language javascript
```
Python

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

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

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

Java

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

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

C#

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

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

Install the CDK Pipelines module along with others you'll be using.

TypeScript

```bash
npm install @aws-cdk/pipelines @aws-cdk/aws-codebuild
npm install @aws-cdk/aws-codepipeline @aws-cdk/aws-codepipeline-actions
```

JavaScript

```bash
npm install @aws-cdk/pipelines @aws-cdk/aws-codebuild
npm install @aws-cdk/aws-codepipeline @aws-cdk/aws-codepipeline-actions
```

Python

```bash
python -m pip install aws_cdk.pipelines aws_cdk.aws_codebuild
python -m pip install aws_cdk.aws_codepipeline aws_cdk.aws_codepipeline_actions
```

Freeze your dependencies in requirements.txt.

**Mac OS X/Linux**

```bash
python -m pip freeze | grep -v '"[-#]"' > requirements.txt
```

**Windows**

```bash
python -m pip freeze | findstr /R /B /V "[-#]" > requirements.txt
```
Define pipelines

Java

Edit your project's pom.xml and add a `<dependency>` element for the pipeline module and a few others you'll need. Follow the template below for each module, placing each inside the existing `<dependencies>` container.

```xml
<dependency>
  <groupId>software.amazon.awscdk</groupId>
  <artifactId>cdk-pipelines</artifactId>
  <version>${cdk.version}</version>
</dependency>
<dependency>
  <groupId>software.amazon.awscdk</groupId>
  <artifactId>codebuild</artifactId>
  <version>${cdk.version}</version>
</dependency>
<dependency>
  <groupId>software.amazon.awscdk</groupId>
  <artifactId>codepipeline</artifactId>
  <version>${cdk.version}</version>
</dependency>
<dependency>
  <groupId>software.amazon.awscdk</groupId>
  <artifactId>codepipeline-actions</artifactId>
  <version>${cdk.version}</version>
</dependency>
```

C#

In Visual Studio, choose Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio and add the following packages. Make sure the Include prerelease checkbox is marked, since the CDK Pipelines module is in developer preview.

- Amazon.CDK.Pipelines
- Amazon.CDK.AWS.CodeBuild
- Amazon.CDK.AWS.CodePipeline
- Amazon.CDK.AWS.CodePipeline.Actions

Finally, add the `@aws-cdk/core:newStyleStackSynthesis` feature flag (p. 147) to the new project's cdk.json file. The file will already contain some context values; add this new one inside the context object.

```json
{
  ...
  "context": {
    ...
    "@aws-cdk/core:newStyleStackSynthesis": "true"
  }
}
```

In a future release of the AWS CDK, "new style" stack synthesis will become the default, but for now we need to opt in using the feature flag.

**Define pipelines**

The construct `CdkPipeline` is the construct that represents a CDK Pipeline. When you instantiate `CdkPipeline` in a stack, you define the source location for the pipeline as well as the build commands.
For example, the following defines a pipeline whose source is stored in a GitHub repository, and includes a build step for a TypeScript application. The Pipeline will be provisioned in account 111111111111 and region eu-west-1.

**TypeScript**

In `lib/my-pipeline-stack.ts` (may vary if your project folder isn't named `my-pipeline`):

```javascript
import { Stack, StackProps, Construct, SecretValue } from '@aws-cdk/core';
import { CdkPipeline, SimpleSynthAction } from '@aws-cdk/pipelines';
import * as codepipeline from '@aws-cdk/aws-codepipeline';
import * as codepipeline_actions from '@aws-cdk/aws-codepipeline-actions';

export class MyPipelineStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    const sourceArtifact = new codepipeline.Artifact();
    const cloudAssemblyArtifact = new codepipeline.Artifact();
    const pipeline = new CdkPipeline(this, 'Pipeline', {
      pipelineName: 'MyAppPipeline',
      cloudAssemblyArtifact,
      sourceAction: new codepipeline_actions.GitHubSourceAction({
        actionName: 'GitHub',
        output: sourceArtifact,
        oauthToken: SecretValue.secretsManager('GITHUB_TOKEN_NAME'),
        trigger: codepipeline_actions.GitHubTrigger.POLL,
        // Replace these with your actual GitHub project info
        owner: 'GITHUB-OWNER',
        repo: 'GITHUB-REPO',
      }),
      synthAction: SimpleSynthAction.standardNpmSynth({
        sourceArtifact,
        cloudAssemblyArtifact,
        // Use this if you need a build step (if you're not using ts-node
        // or if you have TypeScript Lambdas that need to be compiled).
        buildCommand: 'npm run build',
      }),
    });
  }
}
```

In `bin/my-pipeline.ts` (may vary if your project folder isn't named `my-pipeline`):

```javascript
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from '@aws-cdk/core';
import { MyPipelineStack } from '../lib/my-pipeline-stack';

const app = new cdk.App();
new MyPipelineStack(app, 'PipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```
JavaScript

In `lib/my-pipeline-stack.js` (may vary if your project folder isn't named `my-pipeline`):

```javascript
const { Stack, SecretValue } = require('@aws-cdk/core');
const { CdkPipeline, SimpleSynthAction } = require('@aws-cdk/pipelines');
const codepipeline = require('@aws-cdk/aws-codepipeline');
const codepipeline_actions = require('@aws-cdk/aws-codepipeline-actions');

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

    const sourceArtifact = new codepipeline.Artifact();
    const cloudAssemblyArtifact = new codepipeline.Artifact();

    const pipeline = new CdkPipeline(this, 'Pipeline', {
      pipelineName: 'MyAppPipeline',
      cloudAssemblyArtifact,
      sourceArtifact: new codepipeline_actions.GitHubSourceAction({
        actionName: 'GitHub',
        output: sourceArtifact,
        oauthToken: SecretValue.secretsManager('GITHUB_TOKEN_NAME'),
        trigger: codepipeline_actions.GitHubTrigger.POLL,
        // Replace these with your actual GitHub project info
        owner: 'GITHUB-OWNER',
        repo: 'GITHUB-REPO'
      }),
      synthAction: SimpleSynthAction.standardNpmSynth({
        sourceArtifact,
        cloudAssemblyArtifact,

        // Use this if you need a build step (if you're not using ts-node
        // or if you have TypeScript Lambdas that need to be compiled).
        buildCommand: 'npm run build'
      })
    });

    module.exports = { MyPipelineStack }
}
```

In `bin/my-pipeline.js` (may vary if your project folder isn't named `my-pipeline`):

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

const cdk = require('@aws-cdk/core');
const { MyPipelineStack } = require('../lib/my-pipeline-stack');

const app = new cdk.App();
new MyPipelineStack(app, 'PipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1'
  }
});

app.synth();
```
Python

In my-pipeline/my-pipeline-stack.js (may vary if your project folder isn't named my-pipeline):

```python
from aws_cdk.core import Stack, StackProps, Construct, SecretValue
from aws_cdk.pipelines import CdkPipeline, SimpleSynthAction
import aws_cdk.aws_codepipeline as codepipeline
import aws_cdk.aws_codepipeline_actions as codepipeline_actions

class MyPipelineStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)
        source_artifact = codepipeline.Artifact()
        cloud_assembly_artifact = codepipeline.Artifact()
        pipeline = CdkPipeline(self, "Pipeline",
            pipeline_name="MyAppPipeline",
            cloud_assembly_artifact=cloud_assembly_artifact,
            source_action=codepipeline_actions.GitHubSourceAction(
                action_name="GitHub",
                output=source_artifact,
                oauth_token=SecretValue.secrets_manager("GITHUB_TOKEN_NAME"),
                trigger=codepipeline_actions.GitHubTrigger.POLL,
                # Replace these with your actual GitHub project info
                owner="GITHUB-OWNER",
                repo="GITHUB-REPO"),
            synth_action=SimpleSynthAction.standard_npm_synth(
                source_artifact=source_artifact,
                cloud_assembly_artifact=cloud_assembly_artifact,
                # Use this if you need a build step (if you're not using ts-node
                # or if you have TypeScript Lambdas that need to be compiled).
                build_command="npm run build"
            ))
```

In app.py:

```python
#!/usr/bin/env python3

from aws_cdk import core
from my_pipeline.my_pipeline_stack import MyPipelineStack

app = core.App()
MyPipelineStack(app, "my-pipeline",
    env=core.Environment(account="111111111111", region="eu-west-1"))
app.synth()
```

Java

In src/main/java/com/myorg/MyPipelineStack.java (may vary if your project folder isn't named my-pipeline):

```java
package com.myorg;

import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.SecretValue;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
```
import software.amazon.awscdk.pipelines.CdkPipeline;
import software.amazon.awscdk.pipelines.SimpleSynthAction;
import software.amazon.awscdk.pipelines.StandardNpmSynthOptions;
import software.amazon.awscdk.services.codepipeline.Artifact;
import software.amazon.awscdk.services.codepipeline.actions.GitHubSourceAction;
import software.amazon.awscdk.services.codepipeline.actions.GitHubTrigger;

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

    public MyPipelineStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        final Artifact sourceArtifact = new Artifact();
        final Artifact cloudAssemblyArtifact = new Artifact();

        final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
            .pipelineName("MyAppPipeline")
            .cloudAssemblyArtifact(cloudAssemblyArtifact)
            .sourceAction(GitHubSourceAction.Builder.create()
                .actionName("GitHub")
                .output(sourceArtifact)
                .oauthToken(SecretValue.secretsManager("GITHUB_TOKEN_NAME"))
                .trigger(GitHubTrigger.POLL)
                .owner("GITHUB-OWNER")
                .repo("GITHUB-REPO")
                .build())
            .synthAction(SimpleSynthAction.standardNpmSynth(
                StandardNpmSynthOptions.builder()
                    .sourceArtifact(sourceArtifact)
                    .cloudAssemblyArtifact(cloudAssemblyArtifact)
                    .buildCommand("npm run build")
                    .build())
            .build();
    }
}

In `src/main/java/com/myorg/MyPipelineApp.java` (may vary if your project folder isn't named `my-pipeline`):

```java
package com.myorg;

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

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

        MyPipelineStack.Builder.create(app, "PipelineStack")
            .env(new Environment.Builder()
                    .account("111111111111")
                    .region("eu-west-1")
                    .build())
            .build();

        app.synth();
    }
}
```
C#

In src/MyPipeline/MyPipelineStack.cs (may vary if your project folder isn't named my-pipeline):

```csharp
using Amazon.CDK;
using Amazon.CDK.Pipelines;
using Amazon.CDK.AWS.CodePipeline;
using Amazon.CDK.AWS.CodePipeline.Actions;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props = null) :
            base(scope, id, props)
        {
            var sourceArtifact = new Artifact();
            var cloudAssemblyArtifact = new Artifact();

            var pipeline = new CdkPipeline(this, "Pipeline", new CdkPipelineProps
            {
                PipelineName = "MyAppPipeline",
                CloudAssemblyArtifact = cloudAssemblyArtifact,
                SourceAction = new GitHubSourceAction(new GitHubSourceActionProps
                {
                    ActionName = "GitHub",
                    Output = sourceArtifact,
                    OauthToken = SecretValue.SecretsManager("GITHUB TOKEN_NAME"),
                    Trigger = GitHubTrigger.POLL,
                    Owner = "GITHUB-OWNER",
                    Repo = "GITHUB-REPO"
                }),
                SynthAction = SimpleSynthAction.StandardNpmSynth(new StandardNpmSynthOptions
                {
                    SourceArtifact = sourceArtifact,
                    CloudAssemblyArtifact = cloudAssemblyArtifact,
                    BuildCommand = "npm run build"
                })
            });
        }
    }
}
```

In src/MyPipeline/Program.cs (may vary if your project folder isn't named my-pipeline):

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

namespace MyPipeline
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyPipelineStack(app, "MyPipelineStack", new StackProps
            {
                Env = new Amazon.CDK.Environment
                {
                    Account = "11111111111",
                }
            });
        }
    }
}
Region = "eu-west-1"
});
app.Synth();
}
}
}

Note the following in this example:

- The source code is stored in a GitHub repository.
- The GitHub access token needed to access the repo is retrieved from AWS Secrets Manager. Provide the name of the secret where indicated.
- Specify the owner of the repository and the repo name where indicated.

You must deploy a CDK Pipeline manually once. After that, the pipeline will keep itself up to date from the source code repository. To perform the initial deployment:

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

**Tip**
Now that you've done the initial deployment, you no longer need AWS administrative access.

## Sources and synth actions

As we've seen in the preceding example, the basic pieces of CDK pipelines are **sources** and **synth actions**.

Sources are places where your code lives. Any source from the `codepipeline-actions` module can be used.

Synth actions (synthAction) define how to build and synth the project. A synth action can be any AWS CodePipeline action that produces an artifact containing an AWS CDK Cloud Assembly (the `cdk.out` directory created by `cdk synth`). Pass the output artifact of the synth operation in the Pipeline's `cloudAssemblyArtifact` property.

`SimpleSynthAction` is available for synths that can be performed by running a couple of simple shell commands (install, build, and synth) using AWS CodeBuild. When using these, the source repository does not require a `buildspec.yml`. Here's an example of using `SimpleSynthAction` to run a Maven (Java) build followed by a `cdk synth`:

**TypeScript**

```typescript
const pipeline = new CdkPipeline(this, 'Pipeline', {
  // ...
  synthAction: new SimpleSynthAction({
    sourceArtifact,  
    cloudAssemblyArtifact,  
    installCommand: 'npm install -g aws-cdk',  
    buildCommand: 'mvn package',  
    synthCommand: 'cdk synth',  
  })
});
```
Sources and synth actions

JavaScript

```javascript
const pipeline = new CdkPipeline(this, 'Pipeline', {
  // ...
  synthAction: new SimpleSynthAction({
    sourceArtifact,
    cloudAssemblyArtifact,
    installCommand: 'npm install -g aws-cdk',
    buildCommand: 'mvn package',
    synthCommand: 'cdk synth'
  })
});
```

Python

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

        pipeline = CdkPipeline(self, "Pipeline",
          # ...
          synth_action=SimpleSynthAction(
            source_artifact=source_artifact,
            cloud_assembly_artifact=cloud_assembly_artifact,
            install_command="npm install -g aws-cdk",
            build_command="mvn package",
            synth_command="cdk synth"
          ))
```

Java

```java
final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
    // ...
    .synthAction(SimpleSynthAction.Builder.create()
            .sourceArtifact(sourceArtifact)
            .cloudAssemblyArtifact(cloudAssemblyArtifact)
            .installCommand("npm install -g aws-cdk")
            .buildCommand("mvn package")
            .synthCommand("cdk synth")
            .build());
```

C#

```csharp
var pipeline = new CdkPipeline(this, "Pipeline", new CdkPipelineProps
{
    SynthAction = new SimpleSynthAction(new SimpleSynthActionProps
    {
        SourceArtifact = sourceArtifact,
        CloudAssemblyArtifact = cloudAssemblyArtifact,
        InstallCommand = "npm install -g aws-cdk",
        BuildCommand = "mvn package",
        SynthCommand = "cdk synth"
    })
});
```

A couple of convention-based synth operations for TypeScript or JavaScript projects are available as class methods of SimpleSynthAction:
• **standardNpmSynth()** builds using NPM conventions. Expects a `package-lock.json`, a `cdk.json`, and expects the CDK Toolkit to be a versioned dependency in `package.json`. Does not perform a build step by default.

• **standardYarnSynth()** builds using Yarn conventions. Expects a `yarn.lock`, a `cdk.json`, and expects the CDK Toolkit to be a versioned dependency in `package.json`. Does not perform a build step by default.

If your needs are not covered by `SimpleSynthAction`, you can add a custom build/synth step by creating a custom AWS CodeBuild project and passing a corresponding `CodeBuildAction` to the pipeline.

**Application stages**

To define a multi-stack AWS application that can be added to the pipeline all at once, define a subclass of `Stage` (not to be confused with `CdkStage` in the CDK Pipelines module).

The stage contains the stacks that make up your application. If there are dependencies between the stacks, the stacks are automatically added to the pipeline in the right order. Stacks that don't depend on each other are deployed in parallel. You can add a dependency relationship between stacks by calling `stack1.addDependency(stack2)`.

Stages accept a default `env` argument, which the `Stacks` inside the `Stage` will use if no environment is specified for them.

An application is added to the pipeline by calling `addApplicationStage()` with instances of the `Stage`. A stage can be instantiated and added to the pipeline multiple times to define different stages of your DTAP or multi-region application pipeline:

**TypeScript**

```typescript
import { Construct, Stack, StackProps, Stage, StageProps } from '@aws-cdk/core';
import * as codepipeline from '@aws-cdk/aws-codepipeline';
import { CdkPipeline } from '@aws-cdk/pipelines';

export class DatabaseStack extends Stack {
    // ...
}

export class ComputeStack extends Stack {
    // ...
}

// Your application
// May consist of one or more Stacks
//
export class MyApplication extends Stage {
    constructor(scope: Construct, id: string, props?: StageProps) {
        super(scope, id, props);

        const dbStack = new DatabaseStack(this, 'Database');
        new ComputeStack(this, 'Compute', {
            table: dbStack.table,
        });
    }
}

// Stack to hold the pipeline
//
export class MyPipelineStack extends Stack {
    constructor(scope: Construct, id: string, props?: StackProps) {
        super(scope, id, props);
    }
}
```
const sourceArtifact = new codepipeline.Artifact();
const cloudAssemblyArtifact = new codepipeline.Artifact();

const pipeline = new CdkPipeline(this, 'Pipeline', {
    // ...source and build information here
});

// Do this as many times as necessary with any account and region
// Account and region may be different from the pipeline's.
pipeline.addApplicationStage(new MyApplication(this, 'Prod', {
    env: {
        account: '123456789012',
        region: 'eu-west-1',
    }
]));

class DatabaseStack extends Stack {
    // ...
}

class ComputeStack extends Stack {
    // ...
}

// Your application
// May consist of one or more Stacks
//
class MyApplication extends Stage {
    constructor(scope, id, props) {
        super(scope, id, props);

        const dbStack = new DatabaseStack(this, 'Database');
        new ComputeStack(this, 'Compute', {
            table: dbStack.table
        });
    }
}

// Stack to hold the pipeline
//
class MyPipelineStack extends Stack {
    constructor(scope, id, props) {
        super(scope, id, props);

        const sourceArtifact = new codepipeline.Artifact();
        const cloudAssemblyArtifact = new codepipeline.Artifact();

        const pipeline = new CdkPipeline(this, 'Pipeline', {
            // ...source and build information here
        });

        // Do this as many times as necessary with any account and region
        // Account and region may be different from the pipeline's.
pipeline.addApplicationStage(new MyApplication(this, 'Prod', {
            env: {
                account: '123456789012',
            }
        }));
    }
}
region: 'eu-west-1'
})
})
}
}

module.exports = { MyApplication, MyPipelineStack, ComputeStack, DatabaseStack }

### Python

```python
from my_pipeline.my_pipeline_stack import source_artifact
from aws_cdk.core import Construct, Stack, Stage, Environment
from aws_cdk.pipelines import CdkPipeline
import aws_cdk.aws_codepipeline as code_pipeline

class DatabaseStack(Stack):
    pass  # ...

class ComputeStack(Stack):
    pass  # ...

# Your application
# May consist of one or more Stacks
#
class MyApplication(Stage):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)
        db_stack = DatabaseStack(self, "Database")
        ComputeStack(self, "Compute", table=db_stack.table)

# Stack to hold the pipeline
#
class MyPipelineStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        source_artifact = code_pipeline.Artifact()
        cloud_assembly_artifact = code_pipeline.Artifact()

        pipeline = CdkPipeline(self, "Pipeline",
            # ...source and build information here
        )

        # Do this as many times as necessary with any account and region
        # Account and region may be different from the pipeline's.
        pipeline.add_application_stage(MyApplication(self, 'Prod',
            env=Environment(account="123456789012", region="eu-west-1")))
```

### Java

```java
class DatabaseStack extends Stack {
    Table table;

    public DatabaseStack(Construct scope, String id) {
        super(scope, id);
        // ...
    }

    public Table getTable() {
        return table;
    }
}
```
class ComputeStack extends Stack {
    public ComputeStack(Construct scope, String id, Table table) {
        // ... 
    }
}

// Your application
// May consist of one or more Stacks
//
class MyApplication extends Stage {
    public MyApplication(Construct scope, String id, StageProps props) {
        super(scope, id, props);

        DatabaseStack dbStack = new DatabaseStack(this, "Database");
        new ComputeStack(this, "Compute", dbStack.getTable());
    }
}

// Stack to hold the pipeline
//
public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

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

        final Artifact sourceArtifact = new Artifact();
        final Artifact cloudAssemblyArtifact = new Artifact();

        final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
            // ...source and build information here
            .build();

        // Do this as many times as necessary with any account and region
        // Account and region may be different from the pipeline’s.
        pipeline.addApplicationStage(new MyApplication(this, "Prod", new StackProps.Builder()
            .env(new Environment.Builder()
                .account("123456789012")
                .region("eu-west-1")
                .build())
            .build()));
    }
}

C#

public class DatabaseStack : Stack {
    public Table Table { get; set; }

    public DatabaseStack(Construct scope, string id) : base(scope, id)
    {
        // ...
    }
}

public class ComputeStack : Stack {
    public ComputeStack(Construct scope, string id, Table table) : base(scope, id)
Every application stage added by `addApplicationStage()` leads to the addition of an individual pipeline stage, which is returned by the `addApplicationStage()` call. This stage is represented by the `CdkStage` construct. You can add more actions to the stage by calling its `addActions()` method. For example:

**Note**

- `core.Stage` is a stage in an AWS CDK app containing stacks.
- `pipelines.CdkStage` is a stage in a CDK pipeline.

**TypeScript**

```typescript
// import { ManualApprovalAction } from '@aws-cdk/aws-codepipeline-actions';

const testingStage = pipeline.addApplicationStage(new MyApplication(this, 'Testing', {
    env: { account: '111111111111', region: 'eu-west-1' }
});
```
Add an action -- in this case, a Manual Approval action

```javascript
// Add an action -- in this case, a Manual Approval action
// (testingStage.addManualApprovalAction() is an equivalent convenience method)
// Add an action -- in this case, a Manual Approval action
// (testingStage.addManualApprovalAction() is an equivalent convenience method)
```
var testingStage = pipeline.AddApplicationStage(new MyApplication(this, "Testing", new Amazon.CDK.StageProps
    
    Env = new Amazon.CDK.Environment
    
    Account = "111111111111",
    Region = "eu-west-1"

    )

);

// Add an action -- in this case, a Manual Approval action
// (testingStage.AddManualApprovalAction() is an equivalent convenience method)
testingStage.AddActions(new ManualApprovalAction(new ManualApprovalActionProps {
    ActionName = "ManualApproval",
    RunOrder = testingStage.NextSequentialRunOrder()

    )

    ));

You can also add more than one application stage to a pipeline stage. For example:

**TypeScript**

```typescript
// Add two application stages to the same pipeline stage
testingStage.addApplication(new MyApplication1(this, 'MyApp1', {
    env: { account: '111111111111', region: 'eu-west-1' }

    )

    )

    testingStage.addApplication(new MyApplication2(this, 'MyApp2', {
    env: { account: '111111111111', region: 'eu-west-1' }

    )

    });
```

**JavaScript**

```javascript
// Add two application stages to the same pipeline stage
testingStage.addApplication(new MyApplication1(this, 'MyApp1', {
    env: { account: '111111111111', region: 'eu-west-1' }

    )

    )

    testingStage.addApplication(new MyApplication2(this, 'MyApp2', {
    env: { account: '111111111111', region: 'eu-west-1' }

    )

    });
```

**Python**

```python
# Add two application stages to the same pipeline stage
testing_stage.add_application(MyApplication1(this, 'MyApp1', 
    env=Environment(account="111111111111", region="eu-west-1")))

testing_stage.add_application(MyApplication2(this, 'MyApp2', 
    env=Environment(account="111111111111", region="eu-west-1")))
```

**Java**

```java
// Add two application stages to the same pipeline stage
testingStage.addApplication(new MyApplication1(this, "MyApp1", new StageProps.Builder() 
    .env(new Environment.Builder() 
    .account("111111111111") 
    .region("eu-west-1") 
    .build()) 
    .build()));
```
Testing deployments

You can add any type of AWS CodePipeline action to a CDK Pipeline to validate the deployments you are performing. Using the CDK Pipeline library's `ShellScriptAction`, you can try to access a just-deployed Amazon API Gateway backed by a Lambda function, for example, or issue an AWS CLI command to check some setting of a deployed resource.

In its simplest form, adding validation actions looks like this:

**TypeScript**

```typescript
// stage is a CdkStage returned by pipeline.addApplicationStage
stage.addActions(new ShellScriptAction({
  name: 'MyValidation',
  commands: ['curl -Ssf https://my.webservice.com/'],
  // ... more configuration ...
}));
```

**JavaScript**

```javascript
// stage is a CdkStage returned by pipeline.addApplicationStage
stage.addActions(new ShellScriptAction({
  name: 'MyValidation',
  commands: ['curl -Ssf https://my.webservice.com/'],
  // ... more configuration ...
}));
```
Testing deployments

Because many AWS CloudFormation deployments result in the generation of resources with unpredictable names, CDK Pipelines provide a way to read AWS CloudFormation outputs after a deployment. This makes it possible to pass (for example) the generated URL of a load balancer to a test action.

To use outputs, expose the CfnOutput object you're interested in and pass it pipeline.stackOutput().

TypeScript

```typescript
export class MyLbApplication extends Stage {
  public readonly loadBalancerAddress: CfnOutput;

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

    const lbStack = new LoadBalancerStack(this, 'Stack');

    // Or create this in 'LoadBalancerStack' directly
    this.loadBalancerAddress = new CfnOutput(lbStack, 'LbAddress', { value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/` });
  }
}

const lbApp = new MyLbApplication(this, 'MyApp', { env: { /* ... */ } });
```

const stage = pipeline.addApplicationStage(lbApp);
stage.addActions(new ShellScriptAction(
    // ...
    useOutputs: {
        // When the test is executed, this will make $URL contain the
        // load balancer address.
        URL: pipeline.stackOutput(lbApp.loadBalancerAddress),
    }
));

JavaScript

class MyLbApplication extends Stage {

    constructor(scope, id, props) {
        super(scope, id, props);

        const lbStack = new LoadBalancerStack(this, 'Stack');

        // Or create this in `LoadBalancerStack` directly
        this.loadBalancerAddress = new CfnOutput(lbStack, 'LbAddress', {
            value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
        });
    }

    const lbApp = new MyLbApplication(this, 'MyApp', {
        env: { /* ... */ }
    });

    const stage = pipeline.addApplicationStage(lbApp);
stage.addActions(new ShellScriptAction(
    // ...
    useOutputs: {
        // When the test is executed, this will make $URL contain the
        // load balancer address.
        URL: pipeline.stackOutput(lbApp.loadBalancerAddress)
    }
));

Python

class MyLbApplication(Stage):
    load_balancer_address: CfnOutput = None

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

        lb_stack = LoadBalancerStack(self, "Stack")

        # Or create this in `LoadBalancerStack` directly
        self.load_balancer_address = CfnOutput(lb_stack, "LbAddress",
        value=f"https://{lb_stack.load_balancer_dns_name}"
    }

    lb_app = MyLbApplication(self, "Myapp",
        env=Environment(...) 

    stage = pipeline.add_application_stage(lb_app)
stage.add_actions(ShellScriptAction( 
    # ...
    use_outputs=pipeline.stack_output( 
        # When the test is executed, this will make $URL contain the
        # load balancer address.
        URL=lb_app.load_balancer_address)
Java

class MyLbApplication extends Stage {
    CfnOutput loadBalancerAddress;

    public MyLbApplication(Construct scope, String id, StageProps props) {
        super(scope, id, props);

        LoadBalancerStack lbStack = new LoadBalancerStack(this, "Stack");

        // Or create this in `LoadBalancerStack` directly
        loadBalancerAddress = CfnOutput.Builder.create(lbStack, "LbAddress")
            .value(String.format("https://%s/",
                lbStack.getLoadBalancer().getDnsName()))
            .build();

    }

    public CfnOutput getLoadBalancerAddress() {
        return loadBalancerAddress;
    }

    // some time later...

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

        @SuppressWarnings("serial")
        public MyPipelineStack(final Construct scope, final String id, final StackProps props) {
            super(scope, id, props);

            final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
                .value("...source and build information here.
                .build();

            final MyLbApplication lbApp = // ...

            final CdkStage stage = pipeline.addApplicationStage(lbApp);
            stage.addActions(ShellScriptAction.Builder.create()
                .useOutputs(new HashMap<String, StackOutput>() {
                    put("URL", pipeline.stackOutput(lbApp.getLoadBalancerAddress()));
                })
                .build());
        }
    }

C#

public class MyLbApplication : Stage {
    public CfnOutput LoadBalancerAddress { get; set; }

    public MyLbApplication(Construct scope, string id, Amazon.CDK.StageProps props) :
        base(scope, id, props)
    {
        LoadBalancerStack LbStack = new LoadBalancerStack(this, "Stack");

        // Or create this in `LoadBalancerStack` directly
Testing deployments

```java
var loadBalancerAddress = new CfnOutput(LbStack, "LbAddress", new CfnOutputProps {
    Value = "https://{LbStack.LoadBalancer}/"
});

public class MyPipelineStack : Stack {
    public MyPipelineStack(Construct scope, string id, StackProps props = null) :
        base(scope, id) {
        var pipeline = new CdkPipeline(this, "Pipeline", new CdkPipelineProps {
            // ... source and build information here
        });

        MyLbApplication LbApp = new MyLbApplication(this, "App", new Amazon.CDK.StageProps {
            // set up your application stage
        });

        CdkStage stage = pipeline.AddApplicationStage(LbApp);
        stage.AddActions(new ShellScriptAction(new ShellScriptActionProps {
            // ...
            UseOutputs = new Dictionary<string, StackOutput> {
                ["URL"] = pipeline.StackOutput(LbApp.LoadBalancerAddress)
            }
        }));
    }
}
```

The `ShellScriptAction` limits you to rather small validation tests—basically whatever you can write in a few lines of shell script. You can bring additional files (such as complete shell scripts, or scripts in other languages) into the test via the `additionalArtifacts` property.

Bringing in files from the source repository is appropriate if the files are directly usable in the test (for example, if they are themselves executable). Pass the `sourceArtifact`:

**TypeScript**

```typescript
const sourceArtifact = new codepipeline.Artifact();

const pipeline = new CdkPipeline(this, 'Pipeline', {
    // ...
});

const validationAction = new ShellScriptAction({
    name: 'TestUsingSourceArtifact',
    additionalArtifacts: [sourceArtifact],
    // 'test.sh' comes from the source repository
    commands: ['./test.sh'],
});
```

**JavaScript**

```javascript
const sourceArtifact = new codepipeline.Artifact();

const pipeline = new CdkPipeline(this, 'Pipeline', {
    // ...
});

const validationAction = new ShellScriptAction({
    name: 'TestUsingSourceArtifact',
    additionalArtifacts: [sourceArtifact],
    // 'test.sh' comes from the source repository
    commands: ['./test.sh'],
});
```
const pipeline = new CdkPipeline(this, 'Pipeline', {
    // ...
});

const validationAction = new ShellScriptAction({
    name: 'TestUsingSourceArtifact',
    additionalArtifacts: [sourceArtifact],
    // 'test.sh' comes from the source repository
    commands: ['./test.sh']
});

Python

source_artifact = code_pipeline.Artifact()

pipeline = CdkPipeline(self, "Pipeline", ...)

validation_action = ShellScriptAction(
    name="TestUsingSourceArtifact",
    additional_artifacts=[source_artifact],
    # 'test.sh' comes from the source repository
    commands=["./test.sh"]
)

Java

final Artifact sourceArtifact = new Artifact();

final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
    // ...source and build information here
    .build();

ShellScriptAction validationAction = ShellScriptAction.Builder.create()
    .actionName("TestUsingSourceArtifact")
    .additionalArtifacts(Arrays.asList(sourceArtifact))
    // 'test.sh' comes from the source repository
    .commands(Arrays.asList("./test.sh"))
    .build();

C#

Artifact_ sourceArtifact = new Artifact_();

var pipeline = new CdkPipeline(this, "Pipeline", new CdkPipelineProps
{
    // define your pipeline
});

var validationAction = new ShellScriptAction(new ShellScriptActionProps {
    ActionName = "TestUsingSourceArtifact",
    AdditionalArtifacts = new Artifact_[] { sourceArtifact },
    Commands = new string[] { ".\test.sh" }
});

Getting the additional files from the synth step is appropriate if your tests need the compilation step
that is done as part of synthesis. On the synthesis step, specify additionalArtifacts to package
additional subdirectories into artifacts, and use the same artifact in the ShellScriptAction's
additionalArtifacts:
TypeScript

```typescript
// If you are using additional output artifacts from the synth step, 
// they must be named.
const cloudAssemblyArtifact = new codepipeline.Artifact('CloudAsm');
const integTestsArtifact = new codepipeline.Artifact('IntegTests');

const pipeline = new CdkPipeline(this, 'Pipeline', {
    synthAction: SimpleSynthAction.standardNpmSynth({
        sourceArtifact,
        cloudAssemblyArtifact,
        buildCommand: 'npm run build',
        additionalArtifacts: [
            {directory: 'test',
             artifact: integTestsArtifact},
        ],
    }),
    // ...
});

const validationAction = new ShellScriptAction({
    actionName: 'TestUsingBuildArtifact',
    additionalArtifacts: [integTestsArtifact],
    // 'test.js' was produced from 'test/test.ts' during the synth step
    commands: ['node ./test.js'],
});
```

JavaScript

```javascript
// If you are using additional output artifacts from the synth step, 
// they must be named.
const cloudAssemblyArtifact = new codepipeline.Artifact('CloudAsm');
const integTestsArtifact = new codepipeline.Artifact('IntegTests');

const pipeline = new CdkPipeline(this, 'Pipeline', {
    synthAction: SimpleSynthAction.standardNpmSynth({
        sourceArtifact,
        cloudAssemblyArtifact,
        buildCommand: 'npm run build',
        additionalArtifacts: [
            {directory: 'test',
             artifact: integTestsArtifact},
        ],
    }),
    // ...
});

const validationAction = new ShellScriptAction({
    actionName: 'TestUsingBuildArtifact',
    additionalArtifacts: [integTestsArtifact],
    // 'test.js' was produced from 'test/test.ts' during the synth step
    commands: ['node ./test.js'],
});
```

Python

```python
# If you are using additional output artifacts from the synth step, 
# they must be named.
cloud_assembly_artifact = code_pipeline.Artifact("CloudAsm")
```
Testing deployments

```python
test_integ_tests_artifact = code_pipeline.Artifact("IntegTests")

pipeline = CdkPipeline(self, "Pipeline",
    synth_action=SimpleSynthAction.standard_npm_synth(
        source_artifact=source_artifact,
        cloud_assembly_artifact=cloud_assembly_artifact,
        build_command="tsc",
        additional_artifacts=[dict(directory="test",
                                  artifact=test_integ_tests_artifact)])
```

```java
Java

// If you are using additional output artifacts from the synth step,
// they must be named.
final Artifact cloudAssemblyArtifact = new Artifact("IntegTests");
final Artifact integTestsArtifact = new Artifact("IntegTests");

final CdkPipeline pipeline = CdkPipeline.Builder.create(this, "Pipeline")
    .synthAction(SimpleSynthAction.standardNpmSynth(new StandardNpmSynthOptions.Builder()
        .sourceArtifact(sourceArtifact)
        .cloudAssemblyArtifact(cloudAssemblyArtifact)
        .buildCommand("npm run build")
        .additionalArtifacts(Arrays.asList(new AdditionalArtifact.Builder()
            .directory("test").artifact(integTestsArtifact).build()))
        .build());

final ShellScriptAction validationAction = ShellScriptAction.Builder.create()
    .actionName("TestUsingBuildArtifact")
    .additionalArtifacts(Arrays.asList(integTestsArtifact))
    // 'test.js' was produced from 'test/test.ts' during the synth step
    .commands(Arrays.asList("node ./test.js"));
```

```csharp
C#

// If you are using additional output artifacts from the synth step,
// they must be named.
var sourceArtifact = new Artifact("Source");
var cloudAssemblyArtifact = new Artifact("CloudAssembly");
var integTestsArtifact = new Artifact("IntegTests");

var pipeline = new CdkPipeline(this, "Pipeline", new CdkPipelineProps
{
    SynthAction = SimpleSynthAction.StandardNpmSynth(new StandardNpmSynthOptions
    {
        SourceArtifact = sourceArtifact,
        CloudAssemblyArtifact = cloudAssemblyArtifact,
        BuildCommand = "npm run build",
        AdditionalArtifacts = new AdditionalArtifact[]
    }
    {
        new AdditionalArtifact
        {
            Directory = "test",
```
Security notes

Any form of continuous delivery has inherent security risks. Under the AWS Shared Responsibility Model, you are responsible for the security of your information in the AWS cloud. The CDK Pipelines library gives you a head start by incorporating secure defaults and modeling best practices, but by its very nature a library that needs a high level of access to fulfill its intended purpose cannot assure complete security. There are many attack vectors outside of AWS and your organization.

In particular, keep in mind the following.

- Be mindful of the software you depend on. Vet all third-party software you run on your build machine, as it has the ability to change the infrastructure that gets deployed.
- Use dependency locking to prevent accidental upgrades. The default CdkSynth that come with CDK Pipelines respect package-lock.json and yarn.lock to ensure your dependencies are the ones you expect.
- Credentials for production environments should be short-lived. After bootstrapping and initial provisioning, there is no need for developers to have account credentials; all changes can be deployed through the pipeline. Eliminate the possibility of credentials leaking by not needing them in the first place!

Troubleshooting tips

The following issues are commonly encountered while getting started with CDK Pipelines.

Pipeline: Internal Failure

CREATE_FAILED | AWS::CodePipeline::Pipeline | Pipeline/Pipeline
Internal Failure

Check your GitHub access token. It might be missing, or might not have the permissions to access the repository.

Key: Policy contains a statement with one or more invalid principals

CREATE_FAILED | AWS::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.

One of the target environments has not been bootstrapped with the new bootstrap stack. Make sure all your target environments are bootstrapped.

Stack is in ROLLBACK_COMPLETE state and can not be updated.

Stack STACK_NAME is in ROLLBACK_COMPLETE state and can not be updated. (Service:
The stack failed its previous deployment and is in a non-retryable state. Delete the stack from the AWS CloudFormation console and retry the deployment.

**Known issues and limitations**

We're currently aware of the following issues with CDK Pipelines.

- Context queries are not supported; `Vpc.fromLookup()` and similar functions do not work.
- Console links to other accounts will not work. The AWS CodePipeline console assumes links are relative to the current account. You cannot click through to a AWS CloudFormation stack in a different account.
- If a changeset failed to apply, the pipeline is not retried. The pipeline must be restarted manually from the top by clicking **Release Change**.
- A stack that failed to deploy must be deleted manually using the CloudFormation console before starting the pipeline again by clicking **Release Change**.

Please report any other issues you encounter.
AWS CDK tools

This section contains information about the AWS CDK tools listed below.

Topics
- AWS CDK Toolkit (cdk command) (p. 253)
- AWS Toolkit for Visual Studio Code (p. 265)
- SAM CLI (p. 266)

AWS CDK Toolkit (cdk command)

The AWS CDK Toolkit, the CLI command `cdk`, is the primary tool for interacting with your AWS CDK app. It executes your app, interrogates the application model you defined, and produces and deploys the AWS CloudFormation templates generated by the AWS CDK. It also provides other features useful for creating and working with AWS CDK projects. This topic contains information about common use cases of the CDK Toolkit.

The AWS CDK Toolkit is installed with the Node Package Manager. In most cases, we recommend installing it globally.

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

Tip
If you regularly work with multiple versions of the AWS CDK, you may want to install a matching version of the AWS CDK Toolkit in individual CDK projects. To do this, omit `-g` from the `npm install` command. Then use `npx cdk` to invoke it; this will run the local version if one exists, falling back to a global version if not.

Toolkit commands

All CDK Toolkit commands start with `cdk`, which is followed by a subcommand (`list`, `synthesize`, `deploy`, etc.). Some subcommands have a shorter version (`ls`, `synth`, etc.) that is equivalent. Options and arguments follow the subcommand in any order. The available commands are summarized here.

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk list (ls)</code></td>
<td>Lists the stacks in the app</td>
</tr>
<tr>
<td><code>cdk synthesize (synth)</code></td>
<td>Synthesizes and prints the CloudFormation template for the specified stack(s)</td>
</tr>
<tr>
<td><code>cdk bootstrap</code></td>
<td>Deploys the CDK Toolkit stack, required to deploy stacks containing assets</td>
</tr>
<tr>
<td><code>cdk deploy</code></td>
<td>Deploys the specified stack(s)</td>
</tr>
<tr>
<td><code>cdk destroy</code></td>
<td>Destroys the specified stack(s)</td>
</tr>
<tr>
<td><code>cdk diff</code></td>
<td>Compares the specified stack with the deployed stack or a local CloudFormation template</td>
</tr>
<tr>
<td>Command</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cdk metadata</td>
<td>Displays metadata about the specified stack</td>
</tr>
<tr>
<td>cdk init</td>
<td>Creates a new CDK project in the current directory from a specified template</td>
</tr>
<tr>
<td>cdk context</td>
<td>Manages cached context values</td>
</tr>
<tr>
<td>cdk docs (doc)</td>
<td>Opens the CDK API reference in your browser</td>
</tr>
<tr>
<td>cdk doctor</td>
<td>Checks your CDK project for potential problems</td>
</tr>
</tbody>
</table>

For the options available for each command, see the section called “Toolkit reference” (p. 261) or the section called “Built-in help” (p. 254).

**Built-in help**

The AWS CDK Toolkit has integrated help. You can see general help about the utility and a list of the provided subcommands by issuing:

```bash
cdk --help
```

To see help for a particular subcommand, for example deploy, specify it before the `--help` flag.

```bash
cdk deploy --help
```

Issue `cdk version` to display the version of the AWS CDK Toolkit. Provide this information when requesting support.

**Version reporting**

To gain insight into how the AWS CDK is used, the 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.

By default, the AWS CDK reports the name and version of the following NPM modules that are loaded at synthesis time:

- AWS CDK core module
- AWS Construct Library modules
- AWS Solutions Constructs module
- AWS Render Farm Deployment Kit module

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

```json
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Modules: "@aws-cdk/core=X.YY.Z, @aws-cdk/s3=X.YY.Z, @aws-solutions-consturcts/aws-apigateway-lambda=X.YY.Z, aws-rfdk=X.YY.Z"
```
To opt out of version reporting, use one of the following methods:

- Use the `cdk` command with the `--no-version-reporting` argument to opt out for a single command.

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

  Remember, the AWS CDK Toolkit synthesizes fresh templates before deploying, so you should also add `--no-version-reporting` to `cdk deploy` commands.

- Set `versionReporting` to `false` in `.cdk.json` or `~/.cdk.json`. This opts out unless you opt in by specifying `--version-reporting` on an individual command.

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

### Specifying the environment

In AWS CDK terms, the environment (p. 77) consists of a region and AWS credentials valid in that region. The CDK Toolkit needs credentials in order to query your AWS account and to deploy CloudFormation templates.

**Important**

We strongly recommend against using your AWS root account for day-to-day tasks. Instead, create a user in IAM and use its credentials with the CDK.

If you have the AWS CLI installed, the easiest way to satisfy this requirement is to install the AWS CLI and issue the following command:

```bash
aws configure
```

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

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

- In `~/.aws/config` or `%USERPROFILE%\.aws\config`
  ```ini
  [default]
  region=us-west-2
  ```

- In `~/.aws/credentials` or `%USERPROFILE%\.aws\credentials`
  ```ini
  [default]
  aws_access_key_id=AKIAI44QH8DHBEEXAMPLE
  aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
  ```

Besides specifying AWS credentials and a region under the `[default]` section, you can also put them in a `profile` `NAME` section, where `NAME` is the name of the profile. You can add any number of named profiles, with or without a `[default]` section. Be sure to add the same profile sections to both the configuration and credentials files.

**Tip**

Don't name a profile `default`. That's just confusing.

255
Use the `--profile` flag to choose a set of credentials and default region from these configuration files for a given command.

```bash
cdk deploy --profile test PipelineStack
```

Instead of using the configuration files, you can set the environment variables `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_DEFAULT_REGION` to appropriate values.

You may optionally use the `--role-arn` (or `-r`) option to specify the ARN of an IAM role that should be used for deployment. This role must be assumable by the AWS account being used.

### Specifying the app command

Many features of the CDK Toolkit require one or more AWS CloudFormation templates be synthesized, which in turn requires running your application. Since the AWS CDK supports programs written in a variety of languages, it uses a configuration option to specify the exact command necessary to run your app. This option can be specified in two ways.

First, and most commonly, it can be specified using the `app` key inside the file `cdk.json`, which is in the main directory of your AWS CDK project. The CDK Toolkit provides an appropriate command when creating a new project with `cdk init`. Here is the `cdk.json` from a fresh TypeScript project, for instance.

```json
{
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

The CDK Toolkit looks for `cdk.json` in the current working directory when attempting to run your app, so you might keep a shell open in your project's main directory for issuing CDK Toolkit commands.

The CDK Toolkit also looks for the app key in `~/.cdk.json` (that is, in your home directory) if it can't find it in `./cdk.json`. Adding the app command here can be useful if you usually work with CDK code in the same language, as it does not require you to be in the app's main directory when you run a `cdk` command.

If you are in some other directory, or if you want to run your app via a command other than the one in `cdk.json`, you can use the `--app` (or `-a`) option to specify it.

```bash
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

### Specifying stacks

Many CDK Toolkit commands (for example, `cdk deploy`) work on stacks defined in your app. If your app contains only one stack, the CDK Toolkit assumes you mean that one if you don't specify a stack explicitly.

Otherwise, you must specify the stack or stacks you want to work with. You can do this by specifying the desired stacks by ID individually on the command line. Recall that the ID is the value specified by the second argument when you instantiate the stack.

```bash
cdk synth PipelineStack LambdaStack
```

You may also use wildcards to specify IDs that match a pattern.

- `?` matches any single character
When using wildcards, enclose the pattern in quotes. If you don’t, your shell may try to expand the pattern to the names of files in the current directory. At best, this won’t do what you expect; at worst, you could deploy stacks you didn’t intend to.

When using wildcards, enclose the pattern in quotes. If you don’t, your shell may try to expand the pattern to the names of files in the current directory. At best, this won’t do what you expect; at worst, you could deploy stacks you didn’t intend to.

```
  cdk synth "*Stack"    # PipelineStack, LambdaStack, etc.
  cdk synth "Stack?"    # StackA, StackB, Stack1, etc.
  cdk synth "*"         # All stacks in the app
```

**Note**

The CDK Toolkit does not guarantee that stacks are processed in the specified order. If for some reason the order of the stacks is important, use multiple `cdk` commands.

### Bootstrapping your AWS environment

Stacks that contain assets (p. 121) or large AWS Lambda functions require special dedicated AWS CDK resources to be provisioned. Currently, this is only an Amazon S3 bucket. The `cdk bootstrap` command creates the necessary resources for you. You only need to bootstrap if you are deploying a stack that requires these dedicated resources.

```
  cdk bootstrap                  # bootstraps default account/region
  cdk bootstrap --profile test   # bootstraps test environment
```

You may also bootstrap a specific environment. Credentials must be configured (e.g. in `~/.aws/credentials`) for the specified account and region. You may specify a profile that contains the required credentials.

```
  cdk bootstrap ACCOUNT-NUMBER/REGION # e.g.
  cdk bootstrap 1111111111/us-east-1
  cdk bootstrap --profile test 1111111111/us-east-1
```

**Important**

Each environment (account/region combination) to which you deploy such a stack must be bootstrapped separately.

You may incur 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. From time to time, then, you might want to clear out the bucket from the Amazon S3 console.

You can use the `--bootstrap-bucket-name` option of `cdk bootstrap` to specify the name of the bootstrap bucket, if the default (StagingBucket) is not suitable for some reason. You can use the `--toolkit-stack-name` option if the standard name of the stack itself (CDKToolkit) is not suitable.

### Creating a new app

To create a new app, create a directory for it, then, inside the directory, issue `cdk init`.

```
  mkdir my-cdk-app
  cd my-cdk-app
  cdk init TEMPLATE --language LANGUAGE
```

The supported languages (LANGUAGE) are:
**Template** is an optional template. If the desired template is *app*, the default, you may omit it. The available templates are:

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app (default)</td>
<td>Creates an empty AWS CDK app.</td>
</tr>
<tr>
<td>sample-app</td>
<td>Creates an AWS CDK app with a stack containing an Amazon SQS queue and an Amazon SNS topic.</td>
</tr>
</tbody>
</table>

The templates use the name of the project folder to generate names for files and classes inside your new app.

**Listing stacks**

To see a list of the IDs of the stacks in your AWS CDK application, enter one of the following equivalent commands:

```
cdk list
cdk ls
```

If your app contains many stacks, you can specify full or partial stack IDs of the stacks to be listed; see the section called "Specifying stacks" (p. 256).

Add the `--long` flag to see more information about the stacks, including the stack names and their environments (AWS account and region).

**Synthesizing stacks**

The `cdk synthesize` command (almost always abbreviated `synth`) synthesizes a stack defined in your app into a CloudFormation template.

```
cdk synth # if app contains only one stack
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*" # all stacks in app
```

**Note**

The CDK Toolkit actually runs your app and synthesizes fresh templates before most operations (e.g. when deploying or comparing stacks). These templates are stored by default in the `cdk.out` directory. The `cdk synth` command simply prints the generated templates for the specified stack(s).
See `cdk synth --help` for all available options. A few of the most-frequently-used options are covered below.

### Specifying context values

Use the `--context` or `-c` option to pass runtime context (p. 142) values to your CDK app.

```sh
# specify a single context value
cdk synth --context key=value MyStack
```

```sh
# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

When deploying multiple stacks, the specified context values are normally passed to all of them. If you wish, you may specify different values for each stack by prefixing the stack name to the context value.

```sh
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

### Specifying display format

By default, the synthesized template is displayed in YAML format. Add the `--json` flag to display it in JSON format instead.

```sh
cdk synth --json MyStack
```

### Specifying output directory

Add the `--output` (-o) option to write the synthesized templates to a directory other than `cdk.out`.

```sh
cdk synth --output=~/templates
```

### Deploying stacks

The `cdk deploy` subcommand deploys the specified stack(s) to your AWS account.

```sh
cdk deploy  # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"  # all stacks in app
```

**Note**
The CDK Toolkit runs your app and synthesizes fresh AWS CloudFormation templates before deploying anything. Therefore, most command line options you can use with `cdk synth` (for example, `--context`) can also be used with `cdk deploy`.

See `cdk deploy --help` for all available options. A few of the most-frequently-used options are covered below.

### Specifying AWS CloudFormation parameters

The AWS CDK Toolkit supports specifying AWS CloudFormation parameters (p. 112) at deployment. You may provide these on the command line following the `--parameters` flag.
To define multiple parameters, use multiple `--parameters` flags.

```bash
cdk deploy MyStack --parameters uploadBucketName=UploadBucket --parameters downloadBucketName=DownBucket
```

If you are deploying multiple stacks, you can specify a different value of each parameter for each stack by prefixing the name of the parameter with the stack name and a colon. Otherwise, the same value is passed to all stacks.

```bash
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --parameters YourStack:uploadBucketName=UpBucket
```

By default, the AWS CDK retains values of parameters from previous deployments and uses them in later deployments if they are not specified explicitly. Use the `--no-previous-parameters` flag to require all parameters to be specified.

### Specifying outputs file

If your stack declares AWS CloudFormation outputs, these are normally displayed on the screen at the conclusion of deployment. To write them to a file in JSON format, use the `--output-file` flag.

```bash
cdk deploy --output-file outputs.json MyStack
```

### Security-related changes

To protect you against unintended changes that affect your security posture, the AWS CDK Toolkit prompts you to approve security-related changes before deploying them.

You can change the level of change that requires approval by specifying:

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

`LEVEL` can be one of the following:

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>never</td>
<td>Approval is never required</td>
</tr>
<tr>
<td>any-change</td>
<td>Requires approval on any IAM or security-group-related change</td>
</tr>
<tr>
<td>broadening (default)</td>
<td>Requires approval when IAM statements or traffic rules are added; removals don't require approval</td>
</tr>
</tbody>
</table>

The setting can also be configured in the `cdk.json` file.

```json
{
  "app": "...",
  "requireApproval": "never"
}
```
Comparing stacks

The `cdk diff` command compares the current version of a stack defined in your app with the already-deployed version, or with a saved AWS CloudFormation template, and displays a list of differences.

```bash
[~] AWS::S3::Bucket MyFirstBucket MyFirstBucketB8884501
## [~] DeletionPolicy
#   ## [-] Retain
#   ## [+] Delete
## [~] UpdateReplacePolicy
## [-] Retain
## [+] Delete
```

To compare your app's stack(s) with the existing deployment:

```bash
cdk diff MyStack
```

To compare your app's stack(s) with a saved CloudFormation template:

```bash
cdk diff --template ~/stacks/MyStack.old MyStack
```

Toolkit reference

This section provides a reference for the AWS CDK Toolkit derived from its help, first a general reference with the options available with all commands, then (in collapsible sections) specific references with options available only with specific subcommands.

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

Commands:

- `cdk list [STACKS..]` Lists all stacks in the app [aliases: ls]
- `cdk synthesize [STACKS..]` Synthesizes and prints the CloudFormation template for this stack [aliases: synth]
- `cdk bootstrap [ENVIRONMENTS..]` Deploys the CDK toolkit stack into an AWS environment
- `cdk deploy [STACKS..]` Deploys the stack(s) named STACKS into your AWS account
- `cdk destroy [STACKS..]` Destroy the stack(s) named STACKS
- `cdk diff [STACKS..]` Compares the specified stack with the deployed stack or a local template file, and returns with status 1 if any difference is found
- `cdk metadata [STACK]` Returns all metadata associated with this stack
- `cdk init [TEMPLATE]` Create a new, empty CDK project from a template. Invoked without TEMPLATE, the app template will be used.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cdk context</code></td>
<td>Manage cached context values</td>
</tr>
<tr>
<td><code>cdk docs</code></td>
<td>Opens the reference documentation in a browser</td>
</tr>
<tr>
<td><code>cdk doctor</code></td>
<td>Check your set-up for potential problems</td>
</tr>
</tbody>
</table>

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]
- `--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 when templates are printed to STDOUT [boolean] [default: false]
- `--verbose, -v` Show debug logs (specify multiple times to increase verbosity) [count] [default: false]
- `--profile` Use the indicated AWS profile as the default environment [string]
- `--proxy` Use the indicated proxy. Will read from HTTPS_PROXY environment variable if not specified. [string]
- `--ca-bundle-path` Path to CA certificate to use when validating HTTPS requests. Will read from AWS_CA_BUNDLE environment variable if not specified. [string]
- `--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]
- `--no-color` Removes colors and other style from console output [boolean] [default: false]
If your app has a single stack, there is no need to specify the stack name.

If one of `cdk.json` or `~/.cdk.json` exists, options specified there will be used as defaults. Settings in `cdk.json` take precedence.

### `cdk list (ls)`

`cdk list [STACKS..]`

Lists all stacks in the app.

**Options:**

- `--long, -l` Display environment information for each stack

### `cdk synthesize (synth)`

`cdk synthesize [STACKS..]`

Synthesizes and prints the CloudFormation template for this stack.

**Options:**

- `--exclusively, -e` Only synthesize requested stacks, don’t include dependencies

### `cdk bootstrap`

`cdk bootstrap [ENVIRONMENTS..]`

Deploys the CDK toolkit stack into an AWS environment.

**Options:**

- `--bootstrap-bucket-name, -b, --toolkit-bucket-name` The name of the CDK toolkit bucket; bucket will be created and must not exist
- `--bootstrap-kms-key-id` AWS KMS master key ID used for the SSE-KMS encryption
- `--qualifier` Unique string to distinguish multiple bootstrap stacks
- `--public-access-block-configuration` Block public access configuration on CDK toolkit bucket (enabled by default)
- `--tags, -t` Tags to add for the stack
- `--execute` Whether to execute ChangeSet

 thận trọng
### cdk deploy

cdk deploy [STACKS..]

Deploys the stack(s) named STACKS into your AWS account

Options:

- `--build-exclude`, `-E`  Do not rebuild asset with the given ID. Can be specified multiple times.  
  ([array] [default: []])
- `--exclusively`, `-e`  Only deploy requested stacks, don't include dependencies  
  (boolean)
- `--require-approval`  What security-sensitive changes need manual approval  
  (string) [choices: "never", "any-change", "broadening"]
- `--ci`  Force CI detection  
  (boolean) [default: false]
- `--notification-arns`  ARNs of SNS topics that CloudFormation will notify with stack related events  
  ([array])
- `--tags`, `-t`  Tags to add to the stack (KEY=VALUE)  
  ([array])
- `--execute`  Whether to execute ChangeSet (--no-execute will NOT execute the ChangeSet)  
  (boolean) [default: true]
- `--force`, `-f`  Always deploy stack even if templates are identical  
  (boolean) [default: false]
- `--parameters`  Additional parameters passed to CloudFormation at deploy time (STACK:KEY=VALUE)  
  ([array] [default: {}])
- `--outputs-file`, `-O`  Path to file where stack outputs will be written as JSON  
  (string)
- `--previous-parameters`  Use previous values for existing parameters (you must specify all parameters on every deployment if this is disabled)  
  (boolean) [default: true]

### cdk destroy

cdk destroy [STACKS..]

Destroy the stack(s) named STACKS

Options:

- `--exclusively`, `-e`  Only destroy requested stacks, don't include dependees  
  (boolean)
- `--force`, `-f`  Do not ask for confirmation before destroying the stacks
**cdk diff**

`cdk diff [STACKS..]`

Compares the specified stack with the deployed stack or a local template file, and returns with status 1 if any difference is found.

**Options:**

- `--exclusively, -e` Only diff requested stacks, don't include dependencies
- `--context-lines` Number of context lines to include in arbitrary JSON diff rendering [number] [default: 3]
- `--template` The path to the CloudFormation template to compare with [string]

**cdk init**

`cdk init [TEMPLATE]`

Create a new, empty CDK project from a template. Invoked without TEMPLATE, the app template will be used.

**Options:**

- `--language, -l` The language to be used for the new project (default can be configured in ~/.cdk.json) [string] [choices: "csharp", "fsharp", "java", "javascript", "python", "typescript"]
- `--list` List the available templates [boolean]
- `--generate-only` If true, only generates project files, without executing additional operations such as setting up a git repo, installing dependencies or compiling the project [boolean] [default: false]

**cdk context**

`cdk context`

Manage cached context values

**Options:**

- `--reset, -e` The context key (or its index) to reset [string]
- `--clear` Clear all context [boolean]

---

**AWS Toolkit for Visual Studio Code**

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

**SAM CLI**

This topic describes how to use the 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 * as lambda from '@aws-cdk/aws-lambda';
   ```

3. Replace the comment in `lib/cdk-sam-example-stack.ts` with the following Lambda function:

   ```
   new lambda.Function(this, 'MyFunction', {
       runtime: lambda.Runtime.PYTHON_3_7,
       handler: 'app.lambda_handler',
       code: lambda.Code.asset('./my_function'),
   });
   ```

4. Create the directory `my_function`

   ```
   mkdir my_function
   ```

5. Create the file `app.py` in `my_function` with the following content:

   ```
   def lambda_handler(event, context):
       return "This is a Lambda Function defined through CDK"
   ```

6. Run your AWS CDK app and create a AWS CloudFormation template

   ```
   cdk synth --no-staging > template.yaml
   ```

7. Find the logical ID for your Lambda function in `template.yaml`. It will look like `MyFunction12345678`, 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 Invoking app.lambda_handler (python3.7)

| 2019-04-01 12:22:43 Mounting D:\cdk-sam-example\.cdk.staging |
| \a57f59883918e662ab3c46b964d2faa5 as /var/task:ro,delegated inside runtime container |
| START RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72 Version: $LATEST |
| END RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72 |
| REPORT RequestId: 52fdfc07-2182-154f-163f-5f0f9a621d72 Duration: 3.70 ms Billed |
| Duration: 100 ms Memory Size: 128 MB Max Memory Used: 22 MB |

"This is a Lambda Function defined through CDK"
Testing constructs

With the AWS CDK, your infrastructure can be as testable as any other code you write. This article illustrates one approach to testing AWS CDK apps written in TypeScript using the Jest test framework. Currently, TypeScript is the only supported language for testing AWS CDK infrastructure, though we intend to eventually make this capability available in all languages supported by the AWS CDK.

There are three categories of tests you can write for AWS CDK apps.

- **Snapshot tests** test the synthesized AWS CloudFormation template against a previously-stored "golden master" template. This way, when you're refactoring your app, you can be sure that the refactored code works exactly the same way as the original. If the changes were intentional, you can accept a new master for future tests.

- **Fine-grained assertions** test specific aspects of the generated AWS CloudFormation template, such as "this resource has this property with this value." These tests help when you're developing new features, since any code you add will cause your snapshot test to fail even if existing features still work. When this happens, your fine-grained tests will reassure you that the existing functionality is unaffected.

- **Validation tests** help you "fail fast" by making sure your AWS CDK constructs raise errors when you pass them invalid data. The ability to do this type of testing is a big advantage of developing your infrastructure in a general-purpose programming language.

Getting started

As an example, we'll create a dead letter queue construct. A dead letter queue holds messages from another queue that have failed delivery for some time. This usually indicates failure of the message processor, which we want to know about, so our dead letter queue has an alarm that fires when a message arrives. The user of the construct can hook up actions such as notifying an Amazon SNS topic to this alarm.

Creating the construct

Start by creating an empty construct library project using the AWS CDK Toolkit and installing the construct libraries we'll need:

```bash
mkdir dead-letter-queue && cd dead-letter-queue
cdk init --language=typescript lib
npm install @aws-cdk/aws-sqs @aws-cdk/aws-cloudwatch
```

Place the following code in `lib/index.ts`:

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

export class DeadLetterQueue extends sqs.Queue {
    public readonly messagesInQueueAlarm: cloudwatch.IAlarm;
    constructor(scope: Construct, id: string) {
        super(scope, id);
    }
}
```
Installing the testing framework

Since we're using the Jest framework, our next setup step is to install Jest. We'll also need the AWS CDK assert module, which includes helpers for writing tests for CDK libraries, including assert and expect.

```bash
npm install --save-dev jest @types/jest @aws-cdk/assert
```

Updating `package.json`

Finally, edit the project's `package.json` to tell NPM how to run Jest, and to tell Jest what kinds of files to collect. The necessary changes are as follows.

- Add a new `test` key to the `scripts` section
- Add Jest and its types to the `devDependencies` section
- Add a new `jest` top-level key with a `moduleFileExtensions` declaration

These changes are shown in outline below. Place the new text where indicated in `package.json`. The "..." placeholders indicate existing parts of the file that should not be changed.

```json
{
    ...
    "scripts": {
        ...
        "test": "jest",
    },
    "devDependencies": {
        ...
        "@types/jest": "^24.0.18",
        "jest": "^24.9.0",
    },
    "jest": {
        "moduleFileExtensions": ["js"]
    }
}
```

Snapshot tests

Add a snapshot test by placing the following code in `test/dead-letter-queue.test.ts`.

```typescript
import { SynthUtils } from '@aws-cdk/assert';
import { Stack } from '@aws-cdk/core';
import * as dlq from './lib/index';
```
To build the project and run the test, issue these commands.

```bash
npm run build && npm test
```

The output from Jest indicates that it has run the test and recorded a snapshot.

```diff
PASS test/dead-letter-queue.test.js
# dlq creates an alarm (55ms)
› 1 snapshot written.
Snapshot Summary
› 1 snapshot written
```

Jest stores the snapshots in a directory named `__snapshots__` inside the project. In this directory is a copy of the AWS CloudFormation template generated by the dead letter queue construct. The beginning looks something like this.

```diff
exports['dlq creates an alarm 1'] = `
Object {
  "Resources": Object {
    "DLQ581697C4": Object {
      "Type": "AWS::SQS::Queue",
    },
    "DLQAlarm008FBE3A": Object {
      "Properties": Object {
        "AlarmDescription": "There are messages in the Dead Letter Queue",
        "ComparisonOperator": "GreaterThanOrEqualToThreshold",
```

### Testing the test

To make sure the test works, change the construct so that it generates different AWS CloudFormation output, then build and test again. For example, add a `period` property of 1 minute to override the default of 5 minutes. The boldface line below shows the code that needs to be added to `index.ts`.

```diff
this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
  alarmDescription: 'There are messages in the Dead Letter Queue',
  evaluationPeriods: 1,
  threshold: 1,
  metric: this.metricApproximateNumberOfMessagesVisible(),
  period: Duration.minutes(1),
});
```

Build the project and run the tests again.

```bash
npm run build && npm test
```

```diff
FAIL test/dead-letter-queue.test.js
# dlq creates an alarm (58ms)
# dlq creates an alarm
```
Accepting the new snapshot

Jest has told us that the `Period` attribute of the synthesized AWS CloudFormation template has changed from 300 to 60. To accept the new snapshot, issue:

```
npm test -- -u
```

Now we can run the test again and see that it passes.

Limitations

Snapshot tests are easy to create and are a powerful backstop when refactoring. They can serve as an early warning sign that more testing is needed. Snapshot tests can even be useful for test-driven development: modify the snapshot to reflect the result you’re aiming for, and adjust the code until the test passes.

The chief limitation of snapshot tests is that they test the entire template. Consider that our dead letter queue uses the default retention period. To give ourselves as much time as possible to recover the undelivered messages, for example, we might set the queue’s retention time to the maximum—14 days—by changing the code as follows.

```typescript
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string) {
    super(scope, id, {
      // Maximum retention period
      retentionPeriod: Duration.days(14)
    });
  }
}
```

When we run the test again, it breaks. The name we’ve given the test hints that we are interested mainly in testing whether the alarm is created, but the snapshot test also tests whether the queue is created.
Fine-grained assertions

To avoid needing to review every snapshot whenever you make a change, use the custom assertions in the @aws-cdk/assert/jest module to write fine-grained tests that verify only part of the construct's behavior. For example, the test we called "dlq creates an alarm" in our example really should assert only that an alarm is created with the appropriate metric.

The AWS::CloudWatch::Alarm resource specification reveals that we're interested in the properties Namespace, MetricName and Dimensions. We'll use the expect(stack).toHaveResource(...) assertion, which is in the @aws-cdk/assert/jest module, to make sure these properties have the appropriate values.

Replace the code in test/dead-letter-queue.test.ts with the following.

```javascript
import { Stack } from '@aws-cdk/core';
import '@aws-cdk/assert/jest';
import * as dlq from '../lib/index';

test('dlq creates an alarm', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ');

  expect(stack).toHaveResource('AWS::CloudWatch::Alarm', {
    MetricName: "ApproximateNumberOfMessagesVisible",
    Namespace: "AWS/SQS",
    Dimensions: [
      {
        Name: "QueueName",
        Value: { "Fn::GetAtt": [ "DLQ581697C4", "QueueName" ] }
      }
    ],
  });
});

test('dlq has maximum retention period', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ');

  expect(stack).toHaveResource('AWS::SQS::Queue', {
    MessageRetentionPeriod: 1209600
  });
});
```

There are now two tests. The first checks that the dead letter queue creates an alarm on its ApproximateNumberOfMessagesVisible metric. The second verifies the message retention period.

Again, build the project and run the tests.

```bash
npm run build && npm test
```

**Note**

Since we've replaced the snapshot test, the first time we run the new tests, Jest reminds us that we have a snapshot that is not used by any test. Issue npm test -- -u to tell Jest to clean it up.
Validation tests

Suppose we want to make the dead letter queue's retention period configurable. Of course, we also want to make sure that the value provided by the user of the construct is within an allowable range. We can write a test to make sure that the validation logic works: pass in invalid values and see what happens.

First, create a `props` interface for the construct.

```javascript
export interface DeadLetterQueueProps {
  /**
   * The amount of days messages will live in the dead letter queue
   * Cannot exceed 14 days.
   * @default 14
   */
  retentionDays?: number;
}

export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

  constructor(scope: Construct, id: string, props: DeadLetterQueueProps = {}) {
    if (props.retentionDays !== undefined && props.retentionDays > 14) {
      throw new Error('retentionDays may not exceed 14 days');
    }

    super(scope, id, {
      // Given retention period or maximum
      retentionPeriod: Duration.days(props.retentionDays || 14)
    });
  }
}
```

To test that the new feature actually does what we expect, we write two tests:

- One that makes sure the configured value ends up in the template
- One that supplies an incorrect value to the construct and checks it raises the expected error

Add the following to `test/dead-letter-queue.test.ts`.

```javascript
test('retention period can be configured', () => {
  const stack = new Stack();
  new dlq.DeadLetterQueue(stack, 'DLQ', {
    retentionDays: 7
  });

  expect(stack).toHaveResource('AWS::SQS::Queue', {
    MessageRetentionPeriod: 604800
  });
});

test('configurable retention period cannot exceed 14 days', () => {
  const stack = new Stack();

  expect(() => {
    new dlq.DeadLetterQueue(stack, 'DLQ', {
      retentionDays: 15
    });
  }).toThrow('retentionDays may not exceed 14 days');
});
```
Run the tests to confirm the construct behaves as expected.

```
npm run build && npm test
```

```
PASS  test/dead-letter-queue.test.js
    # dlq creates an alarm (62ms)
    # dlq has maximum retention period (14ms)
    # retention period can be configured (18ms)
    # configurable retention period cannot exceed 14 days (1ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
```

## Tips for tests

Remember, your tests will live just as long as the code they test, and be read and modified just as often, so it pays to take a moment to consider how best to write them. Don't copy and paste setup lines or common assertions, for example; refactor this logic into helper functions. Use good names that reflect what each test actually tests.

Don't assert too much in one test. Preferably, a test should test one and only one behavior. If you accidentally break that behavior, exactly one test should fail, and the name of the test should tell you exactly what failed. This is more an ideal to be striven for, however; sometimes you will unavoidably (or inadvertently) write tests that test more than one behavior. Snapshot tests are, for reasons we've already described, especially prone to this problem, so use them sparingly.
Security for the AWS Cloud Development Kit (AWS CDK)

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The Shared Responsibility Model describes this as Security of the Cloud and Security in the Cloud.

**Security of the Cloud** – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

**Security in the Cloud** – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization’s requirements, and applicable laws and regulations.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Topics
- Identity and access management for the AWS Cloud Development Kit (AWS CDK) (p. 275)
- Compliance validation for the AWS Cloud Development Kit (AWS CDK) (p. 276)
- Resilience for the AWS Cloud Development Kit (AWS CDK) (p. 276)
- Infrastructure security for the AWS Cloud Development Kit (AWS CDK) (p. 277)

Identity and access management for the AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) is an Amazon Web Services (AWS) service that helps an administrator securely control access to AWS resources. IAM administrators control who can be authenticated (signed in) and authorized (have permissions) to use resources in AWS services. IAM is an AWS service that you can use with no additional charge.

To use the AWS CDK to access AWS, you need an AWS account and AWS credentials. To increase the security of your AWS account, we recommend that you use an IAM user to provide access credentials instead of using your AWS account credentials.

For details about working with IAM, see AWS Identity and Access Management.

For an overview of IAM users and why they are important for the security of your account, see AWS Security Credentials in the Amazon Web Services General Reference.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.
Compliance validation for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

The security and compliance of AWS services is assessed by third-party auditors as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using the AWS CDK to access an AWS service is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of an AWS service is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security-focused and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience for the AWS Cloud Development Kit (AWS CDK)

The Amazon Web Services (AWS) global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.
Infrastructure security for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.
Troubleshooting common AWS CDK issues

This topic describes how to troubleshoot the following issues with the AWS CDK.

- After updating the AWS CDK, code that used to work fine now results in errors (p. 278)
- After updating the AWS CDK, the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library (p. 280)
- When deploying my AWS CDK stack, I receive a NoSuchBucket error (p. 281)
- When deploying my AWS CDK stack, I receive a forbidden: null message (p. 281)
- When synthesizing an AWS CDK stack, I get the message --app is required either in command-line, in cdk.json or in ~/.cdk.json (p. 282)
- When deploying an AWS CDK stack, I receive an error because the AWS CloudFormation template contains too many resources (p. 282)
- I specified three (or more) Availability Zones for my EC2 Auto-Scaling Group or Virtual Private Cloud, but it was only deployed in two (p. 283)
- My S3 bucket, DynamoDB table, or other resource is not deleted when I issue cdk destroy (p. 283)

After updating the AWS CDK, code that used to work fine now results in errors

Errors in code that used to work is typically a symptom of having mismatched versions of AWS Construct Library modules. Make sure all library modules are the same version and up-to-date.

The modules that make up the AWS Construct Library are a matched set. They are released together and are intended to be used together. Interfaces between modules are considered private; we may change them when necessary to implement new features in the library.

We also update the libraries that are used by the AWS Construct Library from time to time, and different versions of the library modules may have incompatible dependencies. Synchronizing the versions of the library modules will also address this issue.

JSII is an important AWS CDK dependency, especially if you are using the AWS CDK in a language other than TypeScript or JavaScript. You do not ordinarily have to concern yourself with the JSII versions, since it is a declared dependency of all AWS CDK modules. If a compatible version is not installed, however, you can see unexpected type-related errors, such as ‘undefined’ is not a valid TargetType. Making sure all AWS CDK modules are the same version will resolve JSII compatibility issues, since they will all depend on the same JSII version.

Below, you'll find details on managing the versions of your installed AWS Construct Library modules in TypeScript, JavaScript, Python, Java, and C#.

TypeScript/JavaScript

Install your project's AWS Construct Library modules locally (the default). Use npm to install the modules and keep them up to date.

To see what needs to be updated:
To actually update the modules to the latest version:

```bash
npm update
```

If you are working with a specific older version of the AWS Construct Library, rather than the latest, first uninstall all of your project's `@aws-cdk` modules, then reinstall the specific version you want to use. For example, to install version 1.9.0 of the Amazon S3 module, use:

```bash
npm uninstall @aws-cdk/aws-s3
npm install @aws-cdk/aws-s3@1.9.0
```

Repeat these commands for each module your project uses.

You can edit your `package.json` file to lock the AWS Construct Library modules to a specific version, so `npm update` won't update them. You can also specify a version using `-` or `^` to allow modules to be updated to versions that are API-compatible with the current version, such as `^1.0.0` to accept any update API-compatible with version 1.x. Use the same version specification for all AWS Construct Library modules within a project.

**Python**

Use a virtual environment to manage your project's AWS Construct Library modules. For your convenience, `cdk init` creates a virtual environment for new Python projects in the project's `.env` directory.

Add the AWS Construct Library modules your project uses to its `requirements.txt` file. Use the `=` syntax to specify an exact version, or the `~=` syntax to constrain updates to versions without breaking API changes. For example, the following specifies the latest version of the listed modules that are API-compatible with version 1.x:

```text
aws-cdk.core==1.0
aws-cdk.aws-s3==1.0
```

If you wanted to accept only bug-fix updates to, for example, version 1.9.0, you could instead specify `~=1.9.0`. Use the same version specification for all AWS Construct Library modules within a single project.

Use `pip` to install and update the modules.

To see what needs to be updated:

```bash
pip list --local --outdated
```

To actually update the modules to the latest compatible version:

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

If your project requires a specific older version of the AWS Construct Library, rather than the latest, first uninstall all of your project's `aws-cdk` modules. Edit `requirements.txt` to specify the exact versions of the modules you want to use using `=`, then install from `requirements.txt`.

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

Add your project's AWS Construct Library modules as dependencies in your project's pom.xml. You may specify an exact version, or use Maven's range syntax to specify a range of allowable versions.

For example, to specify an exact version of a dependency:

```xml
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>1.23.0</version>
</dependency>
```

To specify that any 1.x.x version is acceptable (note use of right parenthesis to indicate that the end of the range excludes version 2.0.0):

```xml
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>s3</artifactId>
    <version>[1.0.0,2.0.0)</version>
</dependency>
```

Maven automatically downloads and installs the latest versions that allow all requirements to be fulfilled when you build your application.

If you prefer to pin dependencies to a specific version, you can issue `mvn versions:use-latest-versions` to rewrite the version specifications in pom.xml to the latest available versions when you decide to upgrade.

C#

Use the Visual Studio NuGet GUI (Tools > NuGet Package Manager > Manage NuGet Packages for Solution) to install the desired version of your application's AWS Construct Library modules.

- The Installed panel shows you what modules are currently installed; you can install any available version of any module from this page.
- The Updates panel shows you modules for which updates are available, and lets you update some or all of them.

After updating the AWS CDK, the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library

The version of the AWS CDK Toolkit (which provides the cdk command) must be at least equal to the version of the AWS Construct Library. The Toolkit is intended to be backward compatible within the same major version; the latest 1.x version of the toolkit can be used with any 1.x release of the library. For this reason, we recommend you install this component globally and keep it up-to-date.

```
npm update -g aws-cdk
```

If, for some reason, you need to work with multiple versions of the AWS CDK Toolkit, you can install a specific version of the toolkit locally in your project folder.

If you are using a language other than TypeScript or JavaScript, first create a node_modules folder in your project directory. Then, regardless of language, use `npm` to install the AWS CDK Toolkit, omitting the `-g` flag and specifying the desired version. For example:
npm install aws-cdk@1.9.0

To run a locally-installed AWS CDK Toolkit, use the command `npx cdk` rather than just `cdk`. For example:

```
npx cdk deploy MyStack
```

`npx cdk` runs the local version of the AWS CDK Toolkit if one exists, and falls back to the global version when a project doesn't have a local installation. You may find it convenient to set up a shell alias or batch file to make sure `cdk` is always invoked this way. For example, Linux users might add the following statement to their `.bash_profile` file.

```
alias cdk=npx cdk
```

When deploying my AWS CDK stack, I receive a `NoSuchBucket` error

Your AWS environment does not have a staging bucket, which the AWS CDK uses to hold resources during deployment. Stacks require staging if they contain the section called “Assets” (p. 121) or synthesize to AWS CloudFormation templates larger than 50 kilobytes. You can create the staging bucket with the following command:

```
cdk bootstrap
```

To avoid generating unexpected AWS charges, the AWS CDK does not automatically create a staging bucket. You must bootstrap your environment explicitly.

By default, the staging bucket is created in the region specified by the default AWS profile (set by `aws configure`), using that profile's account. You can specify a different account and region on the command line as follows.

```
cdk bootstrap aws://123456789/us-east-1
```

You must bootstrap in every region where you will deploy stacks that require a staging bucket.

To avoid undesired AWS charges, you can delete the contents of the staging bucket after deploying. You can find the bucket in the Amazon S3 management console; it has a name starting with `cdktoolkit-stagingbucket` (it is possible to specify a different name when bootstrapping, but generally you should use the default name.)

You should not need to delete the bucket itself, but if you do, it is best to delete the entire `CDKToolkit` stack through the AWS CloudFormation management console. If you delete the staging bucket entirely, you must re-bootstrap before deploying a stack that requires staging.

When deploying my AWS CDK stack, I receive a `forbidden: null` message

You are deploying a stack that requires the use of a staging bucket, but are using an IAM role or account that lacks permission to write to it. (The staging bucket is used when deploying stacks that contain assets or that synthesize an AWS CloudFormation template larger than 50K.) Use an account or role that has permission to perform the action `s3:*` against the resource `arn:aws:s3:::cdktoolkit-stagingbucket-*`.

(back to list (p. 278))
When synthesizing an AWS CDK stack, I get the message \texttt{--app is required either in command-line, in cdk.json or in ~/.cdk.json}

This message usually means that you aren't in the main directory of your AWS CDK project when you issue \texttt{cdk synth}. The file \texttt{cdk.json} in this directory, created by the \texttt{cdk init} command, contains the command line needed to run (and thereby synthesize) your AWS CDK app. For a TypeScript app, for example, the default \texttt{cdk.json} looks something like this:

```json
{
    "app": "npx ts-node bin/my-cdk-app.ts"
}
```

We recommend issuing \texttt{cdk} commands only in your project's main directory, so the AWS CDK toolkit can find \texttt{cdk.json} there and successfully run your app.

If this isn't practical for some reason, the AWS CDK Toolkit looks for the app's command line in two other locations:

- in \texttt{cdk.json} in your home directory
- on the \texttt{cdk synth} command itself using the \texttt{-a} option

For example, you might synthesize a stack from a TypeScript app as follows.

```bash
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

When deploying an AWS CDK stack, I receive an error because the AWS CloudFormation template contains too many resources

The AWS CDK generates and deploys AWS CloudFormation templates. AWS CloudFormation has a hard limit of 200 resources per stack. With the AWS CDK, you can run up against this limit more quickly than you might expect, especially if you haven't already worked with AWS CloudFormation enough to know what resources are being generated by the AWS Construct Library constructs you're using.

The AWS Construct Library's higher-level, intent-based constructs automatically provision any auxiliary resources that are needed for logging, key management, authorization, and other purposes. For example, granting one resource access to another generates any IAM objects needed for the relevant services to communicate.

In our experience, real-world use of intent-based constructs results in 1–5 AWS CloudFormation resources per construct, though this can vary. For serverless applications, 5–8 AWS resources per API endpoint is typical.

Patterns, which represent a higher level of abstraction, let you define even more AWS resources with even less code. The AWS CDK code in the section called "ECS" (p. 174), for example, generates more than fifty AWS CloudFormation resources while defining only three constructs!

Synthesize regularly and keep an eye on how many resources your stack contains. You'll quickly get a feel for how many resources will be generated by the constructs you use most frequently.

**Tip**

You can count the resources in your synthesized output using the following short script. (Since every CDK user has Node.js installed, it is written in JavaScript.)

```javascript
// rescount.js - count the resources defined in a stack
// invoke with: node rescount.js <path-to-stack-json>
```
// e.g. node rescount.js cdk.out/MyStack.template.json

import * as fs from 'fs';
const path = process.argv[2];

if (path) fs.readFile(path, 'utf8', function(err, contents) {
    console.log(err ? `Error: ${err}` : `Number of resources defined in
    
    ${Object.keys(JSON.parse(contents).Resources).length} resources defined in
    
    ${path}`);
})); else console.log("Please specify the path to the stack's output .json file");

As your stack's resource count approaches 200, consider re-architecting to reduce the number of
resources your stack contains, for example by combining some Lambda functions, or to break it up into
multiple stacks. The CDK supports references between stacks, so it is straightforward to separate your
app's functionality into different stacks in whatever way makes the most sense to you.

Note

AWS CloudFormation experts often suggest the use of nested stacks as a solution to the 200
resource limit. The AWS CDK supports this approach via the NestedStack (p. 76) construct.

I specified three (or more) Availability Zones for my EC2 Auto-Scaling Group or Virtual Private Cloud,
but it was only deployed in two

To get the number of Availability Zones you requested, specify the account and region in the stack's env
property. If you do not specify both, the AWS CDK, by default, synthesizes the stack as environment-
agnostic, such that it can be deployed to any region. You can then deploy the stack to a specific region
using AWS CloudFormation. Because some regions have only two availability zones, an environment-
agnostic template never uses more than two.

Note

At this writing, there is one AWS region that has only one availability zone: ap-northeast-3
(Osaka, Japan). Environment-agnostic AWS CDK stacks cannot be deployed to this region.

You can change this behavior by overriding your stack's availabilityZones (Python:
availability_zones) property to explicitly specify the zones you want to use.

For more information about specifying a stack's account and region at synthesis time, while retaining the
flexibility to deploy to any region, see the section called "Environments" (p. 77).

My S3 bucket, DynamoDB table, or other resource is not deleted when I issue cdk destroy

By default, resources that can contain user data have a removalPolicy (Python: removal_policy)
property of RETAIN, and the resource is not deleted when the stack is destroyed. Instead, the resource
is orphaned from the stack. You must then delete the resource manually after the stack is destroyed.
Until you do, redeploying the stack fails, because the name of the new resource being created during
deployment conflicts with the name of the orphaned resource.

If you set a resource's removal policy to DESTROY, that resource will be deleted when the stack is
destroyed.

TypeScript

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

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

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

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

module.exports = { CdkTestStack }

import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

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

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

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

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

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

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

using Amazon.CDK;
using Amazon.CDK.AWS.S3;
public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id, props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY
    });
}

**Note**

AWS CloudFormation cannot delete a non-empty Amazon S3 bucket. If you set an Amazon S3 bucket's removal policy to `DESTROY`, and it contains data, attempting to destroy the stack will fail because the bucket cannot be deleted.

It is possible to handle the destruction of an Amazon S3 bucket using an AWS CloudFormation custom resource that deletes the bucket's contents before attempting to delete the bucket itself. The third-party construct `auto-delete-bucket`, for example, uses such a custom resource.
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-----
mQINBFsovE8BEADFEVChmAVPoYggsjYu9FFqczxzy9P+2aG/T3LdJ1v3KJ4l8XV
2N5oxYNDBczdTOZnac/t3k3EYJy0V5W4vzh+uB0XZD/z/VDmSdrRhzfEcmI0sOj3z
Ohy1120f544os8LgdLzldnqSRLf+z2tf3gqpon1W71vickWlXr78jDQ4W8p8z92
WNM3fjg855jyPgaJKgAlNn4Vduni1/1XkrG42nrrxU71uUDPvPZ2ELLJ6n0/xgaG
j3le+xQhj45AIs6PgAAgy7jA2fwbkGTHBjujITIA1DwQH5i531oACM9n4JNpcc
xG2eA4VTYbLiznF2ts/a50t+ZGmpq67Sp2j6qyiiumm0Loq3K/R4/yFOPZ8SL
1TuNXX0ecX6piMvufE1qrl5A9Nq92EPtW2QYOY+WzkbcVykDplqij7bMwA7m1T2G
C1jgaTbcEmodVmdQYS1Gb62ptwVrvgA6Af2xX1HUXLQ7tT/wrrTAbfBQRKah85F
a3U9W4cCl3c1MP5iyhN1W0s2moF1iZ1c012nojTTSG60ucNWN4Q8e08FWjungkinj
yxSzIbNQ0aAeoiN4Bz11+n9iaXVDU7K81q1EUgS4PNpju0yruUIq+a9C5sRA7WF+B
zE0oaBqpoAX83cLadTNO6Acwcd9+z2NLXlhiWC4/uh2YHQU1egUg1wMrwARQQAB
1CPBVI1mgQ0RLIPF1YVWqPFG3cy1JQTGWt1heme9uLmNvB763Aj888ElIAckFaIsO
VERCDy8FCQeZg4AHoWxKzdMCQYVCA1JG5sEfrIGAQ1eAPlXQgAKCRAF2geE84x
4CgC4D/0XHhho2r3xZv3s8NGHQLW12y9W1WVHQRmNDQUaw8E2z7+iR3m7nq3xM7Q
BDbccK5g1VLBSQ6H2v6RpysokhPSH1nNzd08Dtv5KPCxx48+1x7lmo+ksSa/+o0
1U0vmTdaRzU0tYh3kOOGXHHXK/111gtF2FQq2YssX1M4PGrj2SklUnlKx561Ih
OJe2zEYzBaskTu/ytRJ36bPP2KZIEXFzAvhmTytuXWUXEFtxOxc6IAlAcY1KTha
aooG7WYv+Fqvbj5gNLcYb552QXma23NZdsc2ZH7468WE15GJ3A7k5xvsPP0C
2YVPD+V0UOIJJu65B5rkhlEHMLRTlklqvcEHxTHxKcip7iTo6CBmcrWAT4eQr7
htLmQ1xrgkWpWnuDZUxG+WQz3Z/UYz2q8RcLcx2md1PhANmefEiHMF3Z4feNN
WLoNWhf8gVCNMHSh8W9EJ52x3N3e51z22yBBFROeq2ZEEBbgQUP6LWFzPzA2
Ekslco/R0uZqcPbgcQoy1FPxVW/ZLgNuG0TQy146YOYdyl3NaTdyUrurktxsX16d
4gdpbBtvqTgecuUpvV2aEuvhWExLxAbnu780FaDAPyeTX+YCLZ2zf+dQvskPQf
80RE7ay7n8IsaLzFBVux/Z/ugyZuxX18r8tDiVQBL510Rmqw==
=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-----
mQINBFtoSs0BEAD6WweLDQ26h0P7J09iR6tVQ4Pq8KXVa5h/e+p/A2Iqw79UYXZ
WKHJzhQ3Sf5Jyi118gPe6Xy5/LV1N5M67Q5vyRz5ybnyL5y5HnTLYA5NRW86M9Zz6jy1R
p1Gzw6VBO6o0xeyeuB9TSZeAn45dTS5j1o4unj4iGJ6Jn6XKz6mP1APUUC2Zf6dK
8VDrpOqj9LeeGlerYNbx+x7ty1/csFIvqK99FPLNMQGLkbbY0r6Rt92gQ+653
tJn+AUyM137w0UIX1gyR5IONXu8HklPGu0NYu79AY7/63Akrzlyfjo7/PZLvueQ
noQNM3j0nokOS5nE5cYaLQw9iBkpxLnm5RjM5GDDC5j8c9u0nLh7J4EOtgzTS1
pem7Y/c+N/2+Ksg9fP8FVTfY6PrvdIlx2a1rDflOQSG9t5rR5V5wPFI4sGV04sI
x7A18VF/OBjaGz1DaqMg/Q9Vb95KAGUASy630fpe14grDSo0yOEXZ+1annxwp1xF
S6x8RC4JF7mXuS58nTNGudPf/9oD69FRNTNCyRmn1G00cPn5037PAC2L7/Busq
3+K6pTkCzCyF/C/Swi6p/xIDAQsGxftc1l1sFDyXRjsk7u0uwC5j1A06X2/jgX
4wunRJBAV1aW95b1+yfaYYC02afXX06caObv81vH7Plc4eC2D1z6yL1D3kKQAQAB
CNBv1MgSIJN5S9BZ971dBzd3Ma52485998pU8bhWP6G24Yu29vP0k5PQWtAQa5KUC
W2h5Qr0BlwU54T0AaCCLQgHAIW8bU1cA6kKCQWAgMBAh4BAeAAAAoJEBv6zky
obx6B34P/nB5qkYhyT6D1q1wK7tudDRp86fc/pc6jD/GdAhN4j8z1bQBPjwS
95OT7/wqahXJhqa/tQP7EIR+z0fW7my8eOodjiu7x7yjPfrhG9n0cXi1NALDcNoH
Bvab3dkD6WNNWSurg2wbcFMTkTvLxjHjPQas6kt0oojpeZ9865CDS/TKZjyw6/6gFm
DlDh3SnSmDUM34StEgew79Ls2aQ2L7cvM/N59k+AGyEMHZDHkEw/Bge5Ovz50Yo
snp/lisbHbPPRW7u6q1KvPkJMwuo2v1WjDfgeBLyBjyfv55mQx7ZG7ywAx/rz2tA6U
isQ0qOML54683VdVoMj5yYmXaQAgL7Q6C/coUW9K2pSUQy0Wu631ktpFp
MyCQ8rkxH9aSd4+s1eltexM9Y936nLyG2BNXi0zXVQ2osw1LuhPoAP4e4Kw
/unVh5sE6a6P0acho7qZrLkZcm9Y9S3GOn2Z7j1/7k0eogmmd1X7QMSep/Fp
gnhJ7B5x314XqrQg9hWt3mVTNGHY3mX7z1WnlUbBpjb+07hbcxv0U01/P7f7w
ZmLa391LMteqG8MwX7t366wJdwADm7brt2at1tIIlHtWNPcOWBh1XLe/03ENCOp
njZ7o8sBsk720VvCC0H3v568yO7iL93O6r7x0owCwBP5akTCR0u0b
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
AWS CDK Developer Guide history

See Releases for information about AWS CDK releases. The AWS CDK is updated approximately once a week. Maintenance versions may be released between weekly releases to address critical issues. Each release includes a matched AWS CDK Toolkit (CDK CLI), AWS Construct Library, and API Reference. Updates to this Guide generally do not synchronize with AWS CDK releases.

**Note**
The table below represents significant documentation milestones. We fix errors and improve content on an ongoing basis.

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add CDK Pipelines how-to (p. 288)</td>
<td>CDK Pipelines let you easily automate the deployment of your AWS CDK apps from source control whenever they're updated.</td>
<td>July 17, 2020</td>
</tr>
<tr>
<td>Improve CDK Toolkit topic (p. 288)</td>
<td>Include more information and examples around performing common tasks with the CLI (and the relevant flags) rather than just including a copy of the help.</td>
<td>July 9, 2020</td>
</tr>
<tr>
<td>Improve CodePipeline example (p. 288)</td>
<td>Update pipeline stack to build in proper language and add more material dealing with the CodeCommit repository.</td>
<td>July 6, 2020</td>
</tr>
<tr>
<td>Improve Getting Started (p. 288)</td>
<td>Remove extraneous material from Getting Started, use a more conversational tone, incorporate current best practices. Break out Hello World into its own topic.</td>
<td>June 17, 2020</td>
</tr>
<tr>
<td>Update stability index (p. 288)</td>
<td>Incorporate the latest definitions of the stability levels for AWS Construct Library modules.</td>
<td>June 11, 2020</td>
</tr>
<tr>
<td>CDK Toolkit versioning (p. 288)</td>
<td>Add information about cloud assembly versioning and compatibility of the CDK Toolkit (CLI) with the AWS Construct Library</td>
<td>April 22, 2020</td>
</tr>
<tr>
<td>Translating from TypeScript (p. 288)</td>
<td>Updated &quot;CDK in Other Languages&quot; topic to also include JavaScript, Java, and C# and renamed it &quot;Translating from TypeScript.&quot;</td>
<td>April 10, 2020</td>
</tr>
<tr>
<td>Parameters topic (p. 288)</td>
<td>Add Concepts topic on using parameters with the AWS CDK.</td>
<td>April 8, 2020</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Working with the CDK (p. 288)</td>
<td>Add &quot;Working with the CDK&quot; articles for the five supported languages. Various other improvements and fixes.</td>
<td>February 4, 2020</td>
</tr>
<tr>
<td>Java code snippets (p. 288)</td>
<td>Add Java code snippets throughout. Designate Java and C# bindings stable.</td>
<td>November 25, 2019</td>
</tr>
<tr>
<td>C# code snippets (p. 288)</td>
<td>Add C# code snippets throughout.</td>
<td>November 19, 2019</td>
</tr>
<tr>
<td>Python code snippets (p. 288)</td>
<td>Add Python code snippets throughout. Add Troubleshooting and Testing topics.</td>
<td>November 14, 2019</td>
</tr>
<tr>
<td>Troubleshooting topic (p. 288)</td>
<td>Add Troubleshooting topic to AWS CDK Developer Guide.</td>
<td>October 30, 2019</td>
</tr>
<tr>
<td>Improve Environments topic (p. 288)</td>
<td>Add Troubleshooting topic to AWS CDK Developer Guide.</td>
<td>October 10, 2019</td>
</tr>
<tr>
<td>ECS Patterns improvements (p. 288)</td>
<td>Updates to reflect improvements to ECS Patterns module.</td>
<td>September 17, 2019</td>
</tr>
<tr>
<td>New tagging API (p. 288)</td>
<td>Update tagging topic to use new API.</td>
<td>August 13, 2019</td>
</tr>
<tr>
<td>General availability (p. 288)</td>
<td>The AWS CDK Developer Guide is released.</td>
<td>July 11, 2019</td>
</tr>
</tbody>
</table>