AWS Cloud Development Kit (AWS CDK)

Developer Guide
# Table of Contents

What Is the AWS CDK? ................................................................................................................ 1
Why Use the AWS CDK? ............................................................................................................. 2
Developing with the AWS CDK ............................................................................................... 5
Contributing to the AWS CDK ................................................................................................ 6
Additional Documentation and Resources .............................................................................. 6
About Amazon Web Services .................................................................................................. 7

Getting Started ......................................................................................................................... 8
Prerequisites .............................................................................................................................. 8
Installing the AWS CDK .......................................................................................................... 9
Updating Your Language Dependencies .................................................................................... 9
Using the env Property to Specify Account and Region ......................................................... 9
Specifying Your Credentials and Region ............................................................................... 12
   Using the --profile Option to Specify Credentials and Region ..................................... 12
   Using Environment Variables to Specify Credentials and a Region .............................. 12
   Using the AWS CLI to Specify Credentials and a Region ............................................... 13
Hello World ............................................................................................................................ 13
   Creating the App Directory ............................................................................................... 13
   Initializing the App .......................................................................................................... 13
   Compiling the App .......................................................................................................... 15
  Listing the Stacks in the App ............................................................................................ 15
   Adding an Amazon S3 Bucket ......................................................................................... 15
   Synthesizing an AWS CloudFormation Template ......................................................... 18
   Deploying the Stack ........................................................................................................ 19
   Modifying the App .......................................................................................................... 19
   Preparing for Deployment .............................................................................................. 20
   Destroying the App's Resources .................................................................................... 21

Concepts ................................................................................................................................ 22
   Constructs ......................................................................................................................... 22
      AWS Construct Library ................................................................................................. 22
      Composition .................................................................................................................. 23
      Initialization .................................................................................................................. 23
      Apps and Stacks .......................................................................................................... 23
      Using Constructs ........................................................................................................ 25
      Configuration .............................................................................................................. 26
      Interacting with Constructs ...................................................................................... 27
      Authoring Constructs .................................................................................................. 29
   Apps ................................................................................................................................ 33
      The App Construct ....................................................................................................... 34
      App Lifecycle .............................................................................................................. 35
      Cloud Assemblies ....................................................................................................... 37
   Stacks ................................................................................................................................ 38
      Stack API ..................................................................................................................... 42
   Environments ..................................................................................................................... 43
   Resources .......................................................................................................................... 48
      Resource Attributes ..................................................................................................... 49
      Referencing Resources ................................................................................................. 50
      Accessing Resources in a Different Stack ................................................................... 51
      Physical Names ............................................................................................................. 52
      Passing Unique Identifiers ........................................................................................ 53
      Importing Existing External Resources ..................................................................... 54
      Permission Grants ....................................................................................................... 57
      Metrics and Alarms ...................................................................................................... 58
      Network Traffic ........................................................................................................... 59
      Amazon CloudWatch Events ....................................................................................... 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifiers</td>
<td>62</td>
</tr>
<tr>
<td>Construct IDs</td>
<td>62</td>
</tr>
<tr>
<td>Paths</td>
<td>64</td>
</tr>
<tr>
<td>Unique IDs</td>
<td>64</td>
</tr>
<tr>
<td>Logical IDs</td>
<td>65</td>
</tr>
<tr>
<td>Tokens</td>
<td>65</td>
</tr>
<tr>
<td>Tokens and Token Encodings</td>
<td>66</td>
</tr>
<tr>
<td>String-Encoded Tokens</td>
<td>67</td>
</tr>
<tr>
<td>List-Encoded Tokens</td>
<td>68</td>
</tr>
<tr>
<td>Number-Encoded Tokens</td>
<td>69</td>
</tr>
<tr>
<td>Lazy Values</td>
<td>69</td>
</tr>
<tr>
<td>Converting to JSON</td>
<td>70</td>
</tr>
<tr>
<td>Tagging</td>
<td>71</td>
</tr>
<tr>
<td>Tag.add()</td>
<td>72</td>
</tr>
<tr>
<td>Tag.remove()</td>
<td>73</td>
</tr>
<tr>
<td>Example</td>
<td>74</td>
</tr>
<tr>
<td>Assets</td>
<td>76</td>
</tr>
<tr>
<td>Assets in Detail</td>
<td>76</td>
</tr>
<tr>
<td>Asset Types</td>
<td>76</td>
</tr>
<tr>
<td>AWS CloudFormation Resource Metadata</td>
<td>87</td>
</tr>
<tr>
<td>Permissions</td>
<td>87</td>
</tr>
<tr>
<td>Grants</td>
<td>87</td>
</tr>
<tr>
<td>Roles</td>
<td>88</td>
</tr>
<tr>
<td>Resource Policies</td>
<td>91</td>
</tr>
<tr>
<td>Principals</td>
<td>92</td>
</tr>
<tr>
<td>Context</td>
<td>92</td>
</tr>
<tr>
<td>Construct Context</td>
<td>93</td>
</tr>
<tr>
<td>Context Methods</td>
<td>93</td>
</tr>
<tr>
<td>Viewing and Managing Context</td>
<td>94</td>
</tr>
<tr>
<td>Example</td>
<td>94</td>
</tr>
<tr>
<td>Aspects</td>
<td>97</td>
</tr>
<tr>
<td>Aspects in Detail</td>
<td>97</td>
</tr>
<tr>
<td>Example</td>
<td>98</td>
</tr>
<tr>
<td>Escape Hatches</td>
<td>99</td>
</tr>
<tr>
<td>Using AWS CloudFormation Constructs Directly</td>
<td>100</td>
</tr>
<tr>
<td>Modifying the AWS CloudFormation Resource behind AWS Constructs</td>
<td>102</td>
</tr>
<tr>
<td>Raw Overrides</td>
<td>103</td>
</tr>
<tr>
<td>Custom Resources</td>
<td>104</td>
</tr>
<tr>
<td>API Reference</td>
<td>105</td>
</tr>
<tr>
<td>Versioning</td>
<td>105</td>
</tr>
<tr>
<td>AWS CDK Stability Index</td>
<td>105</td>
</tr>
<tr>
<td>Identifying the Support Level of an API</td>
<td>105</td>
</tr>
<tr>
<td>Language Binding Stability</td>
<td>106</td>
</tr>
<tr>
<td>Other Languages</td>
<td>106</td>
</tr>
<tr>
<td>Importing a Module</td>
<td>106</td>
</tr>
<tr>
<td>Instantiating a Class</td>
<td>107</td>
</tr>
<tr>
<td>Methods</td>
<td>107</td>
</tr>
<tr>
<td>Enum Constants</td>
<td>107</td>
</tr>
<tr>
<td>Defining Constructs</td>
<td>108</td>
</tr>
<tr>
<td>Structs (Interfaces)</td>
<td>108</td>
</tr>
<tr>
<td>Object Interfaces</td>
<td>108</td>
</tr>
<tr>
<td>Examples</td>
<td>110</td>
</tr>
<tr>
<td>Serverless</td>
<td>110</td>
</tr>
<tr>
<td>Create a AWS CDK App</td>
<td>110</td>
</tr>
<tr>
<td>Create a Lambda Function to List All Widgets</td>
<td>112</td>
</tr>
<tr>
<td>Creating a Widget Service</td>
<td>113</td>
</tr>
<tr>
<td>Add the Service to the App</td>
<td>117</td>
</tr>
</tbody>
</table>
Resilience ................................................................. 181
Infrastructure Security ............................................... 182
Troubleshooting ...................................................... 183
OpenPGP Keys .......................................................... 189
    AWS CDK OpenPGP Key ........................................... 189
    JSII OpenPGP Key .................................................. 190
Document History ..................................................... 191
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. 22). You compose these together into Stacks (p. 38) and Apps (p. 33).
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 AWS Fargate service (this is the code we use in the Creating an AWS Fargate Service Using the AWS CDK (p. 123)).

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
        });
    }
}```
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

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

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

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster, # Required
cpu=512, # Default is 256
            desired_count=6, # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048, # Default is 512
            public_load_balancer=True) # Default is False

Java

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();

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

public class MyEcsConstructStack : Stack
{
    public MyEcsConstructStack(Construct scope, string id, IStackProps props = null) :
        base(scope, id, props)
    {
        var vpc = new Vpc(this, "MyVpc", new VpcProps
        {
            MaxAzs = 3
        });
        var cluster = new Cluster(this, "MyCluster", new ClusterProps
        {
            Vpc = vpc
        });
        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
            {
                Cluster = cluster,
                Cpu = 512,
                DesiredCount = 6,
                TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
                {
                    Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
                },
                MemoryLimitMiB = 2048,
                PublicLoadBalancer = true,
            }
        );
    }
}

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

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

Other advantages of the AWS CDK include:

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

Developing with the AWS CDK

Unless otherwise indicated, the code examples in this guide are in TypeScript. To aid you in porting a TypeScript example to a supported programming language, see AWS CDK in Other Languages
The AWS CDK also includes examples in the supported programming languages. See AWS CDK Examples (p. 146) for a list of the examples.

The AWS CDK Tools (p. 166) 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. 22) includes a module for each AWS service with constructs that offer rich APIs that encapsulate the details of how to create resources for an Amazon or AWS service. The aim of the AWS Construct Library is to reduce the complexity and glue logic required when integrating various AWS services to achieve your goals on AWS.

**Note**

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

## Contributing to the AWS CDK

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

## Additional Documentation and Resources

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

- 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. 189)
    - JSII OpenPGP Key (p. 190)
- AWS CDK Sample for Cloud9
- AWS CloudFormation Concepts
- AWS Glossary
About Amazon Web Services

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

AWS uses a pay-as-you-go service model. You are charged only for the services that you — or your applications — use. Also, to make AWS useful as a platform for prototyping and experimentation, AWS offers a free usage tier, in which services are free below a certain level of usage. For more information about AWS costs and the free usage tier, see Test-Driving AWS in the Free Usage Tier.

To obtain an AWS account, go to aws.amazon.com, and then choose Create an AWS Account.
Getting Started With the AWS CDK

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

**Note**
Want to dig deeper? Try the CDK Workshop for a more in-depth tour of a real-world project.

**Tip**
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.

Prerequisites

All CDK developers need to install Node.js (>= 10.3.0), 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 backend and toolset.

You must provide your credentials and an AWS Region to use the AWS CDK CLI, as described in Specifying Your Credentials and Region (p. 12).

Other prerequisites depend on your development language, as follows.

**TypeScript**

TypeScript >= 2.7

**Python**

- Python >= 3.6

**Java**

- Maven 3.5.4 or later and Java 8
- A Java IDE is preferred (the examples in this guide may refer to Eclipse). cdk init creates a Maven project; most IDEs can import this style of project. Some IDEs may need to be configured to use Java 8 (also known as 1.8).
- Set the JAVA_HOME environment variable to the path to where you have installed the JDK on your machine

**C#**

- .NET standard 2.1 compatible implementation:
  - .NET Core >= 3.0 (3.1 upon its release)
  - .NET Framework >= 4.6.1, or
  - Mono >= 5.4

## Installing the AWS CDK

Install the AWS CDK using the following 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
```

## Updating Your Language Dependencies

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

### TypeScript

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

### Python

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

You might have to issue this command multiple times to update all dependencies.

### Java

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

### C#

```bash
nuget update
```

Or **Tools > NuGet Package Manager > Manage NuGet Packages for Solution** in Visual Studio

## Using the env Property to Specify Account and Region

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

### TypeScript

```javascript
new MyStack(app, 'MyStack', {
  env: {
    region: 'REGION',
  
```
Using the env Property to Specify Account and Region

account: 'ACCOUNT'

Python

MyStack(app, "MyStack", env=core.Environment(region="REGION", account="ACCOUNT")

Java

new MyStack(app, "MyStack", StackProps.builder().env(
    Environment.builder()
    .account("ACCOUNT")
    .region("REGION")
    .build()).build());

C#

new MyStack(app, "MyStack", new StackProps
{    Env = new Amazon.CDK.Environment
{        Account = "ACCOUNT",
        Region = "REGION"
    }
});

Note

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

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

TypeScript

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

Python

MyStack(app, "Stack-One-W",   env=core.Environment(account="ONE", region="us-west-2"))
MyStack(app, "Stack-One-E",   env=core.Environment(account="ONE", region="us-east-1"))
MyStack(app, "Stack-Two-W",   env=core.Environment(account="TWO", region="us-west-2"))
MyStack(app, "Stack-Two-E",   env=core.Environment(account="TWO", region="us-east-1"))
MyStack(app, "Stack-Three-W", env=core.Environment(account="THREE", region="us-west-2"))
Using the `env` Property to Specify Account and Region

```java
MyStack(app, "Stack-Three-E", env=core.Environment(account="THREE", region="us-east-1"))
```

Java

```java
public class MyApp {

    private static App app;

    // Helper method to declare MyStacks in specific accounts/regions
    private static MyStack makeMyStack(final String name, final String account, final String region) {
        return new MyStack(app, name, StackProps.builder()
            .env(Environment.builder()
                .account(account)
                .region(region)
                .build())
            .build());
    }

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

        makeMyStack("Stack-One-W", "ONE", "us-west-2");
        makeMyStack("Stack-One-E", "ONE", "us-east-1");
        makeMyStack("Stack-Two-W", "TWO", "us-west-2");
        makeMyStack("Stack-Two-E", "TWO", "us-east-1");
        makeMyStack("Stack-Three-W", "THREE", "us-west-2");
        makeMyStack("Stack-Three-E", "THREE", "us-east-1");

        app.synth();
    }
}
```

C#

```csharp
// Helper func to declare MyStacks in specific accounts/regions
Stack makeMyStack(string name, string account, string region)
{
    return new MyStack(app, name, new StackProps
    {
        Env = new Amazon.CDK.Environment
        {
            Account = account,
            Region = region
        }
    });
}
```

```csharp
makeMyStack("Stack-One-W", account: "ONE", region: "us-west-2");
makeMyStack("Stack-One-E", account: "ONE", region: "us-east-1");
makeMyStack("Stack-Two-W", account: "TWO", region: "us-west-2");
makeMyStack("Stack-Two-E", account: "TWO", region: "us-east-1");
makeMyStack("Stack-Three-W", account: "THREE", region: "us-west-2");
makeMyStack("Stack-Three-E", account: "THREE", region: "us-east-1");
```

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

```bash
cdk deploy Stack-Two-E
```
Specifying Your Credentials and Region

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

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

Using the `--profile` Option to Specify Credentials and Region

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

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

```
[profile test]
aws_access_key_id=AKIAI44QH8DHBEEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Ut/h3yC08nvbEXaMPLvKEY
region=us-west-2
```

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

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

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

See Named Profiles in the AWS CLI documentation for details.

Using Environment Variables to Specify Credentials and a Region

Use environment variables to specify your credentials and region.

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

For example, to set the region to `us-east-2` on Linux or macOS:
Using the AWS CLI to Specify Credentials and a Region

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

**Hello World Tutorial**

The typical workflow for creating a new app is:

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

And of course, keep your code under version control.

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

**Creating the App Directory**

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

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

**Note**

Be sure to use the name `hello-cdk` for your project directory. The AWS CDK project template uses the directory name to name things in the generated code, so if you use a different name, you'll need to change some of the code in this article.

**Initializing the App**

To initialize your new AWS CDK app, you use the `cdk init` command.
cdk init --language LANGUAGE [TEMPLATE]

Where:

- `LANGUAGE` is one of the supported programming languages: `csharp` (C#), `java` (Java), `javascript` (JavaScript), `python` (Python), or `typescript` (TypeScript)
- `TEMPLATE` is an optional template. If the desired template is `app`, the default, you may omit it.

The following table describes the templates available with the supported languages.

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>app</code> (default)</td>
<td>Creates an empty AWS CDK app.</td>
</tr>
<tr>
<td><code>sample-app</code></td>
<td>Creates an AWS CDK app with a stack containing an Amazon SQS queue and an Amazon SNS topic.</td>
</tr>
</tbody>
</table>

For Hello World, you can just use the default template.

TypeScript

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

Python

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

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

On Linux/MacOS:

```
source .env/bin/activate
```

On Windows:

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

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

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

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

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

Java

```
cdk init --language java
```
Compiling the App

Compile your program, as follows.

TypeScript

```bash
npm run build
```

Python

Nothing to compile.

Java

```bash
mvn compile
```

Tips

You can suppress the INFO messages in the build log by adding the `-q` option to your `mvn` commands. (Don't forget the one in `cdk.json`.)

C#

```bash
dotnet build src
```

or press F6 in Visual Studio

Listing the Stacks in the App

List the stacks in the app.

```bash
cdk ls
```

The result is just the name of the stack.

HelloCdkStack

Adding an Amazon S3 Bucket

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

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

TypeScript

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

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

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

Java

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

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

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. Amazon S3 buckets are represented by the `Bucket` class.

TypeScript

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

```typescript
import core = require('@aws-cdk/core');
import s3 = require('@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
    });
  }
}
```

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 (s3 as s3,
                     core)
```

Replace the comment with the following code.

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

Java

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

```java
package com.myorg;
import software.amazon.awscdk.core.Construct;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.s3.Bucket;

public class HelloStack extends Stack {
    public HelloStack(final Construct scope, final String id) {
        this(scope, id, null);
    }
    public HelloStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

C#

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

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.S3;
namespace HelloCdk
{
    public class HelloStack : Stack
    {
        public HelloStack(Construct scope, string id, IStackProps props) : base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps
            {
                Versioned = true
            });
        }
    }
}
```

Notice a few things:

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

Compile your program, as follows.
Synthesizing an AWS CloudFormation Template

Synthesize an AWS CloudFormation template for the app, as follows. If you get an error like "--app is required...", it's because you are running the command from a subdirectory of your project directory. Navigate to the project directory and try again.

cdk synth

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

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

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

Note
The AWS CDK CLI automatically adds the AWS::CDK::Metadata resource to your template. The AWS CDK uses metadata to gain insight into how the AWS CDK is used. One possible benefit
is that the CDK team could notify users if a construct is going to be deprecated. For details, including how to opt out (p. 170) of version reporting, see Version Reporting (p. 170).

**Deploying the Stack**

Deploy the app, as follows.

```bash
cdk deploy
```

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

**Modifying the App**

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

**TypeScript**

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

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

**Python**

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

**Java**

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

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

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

**C#**

Update `HelloStack.cs`.

```csharp
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true,
    Encryption = BucketEncryption.KMS_MANAGED
});
```
Compile your program, as follows.

**TypeScript**

```bash
npm run build
```

**Python**

Nothing to compile.

**Java**

```bash
mvn compile
```

**Tip**

You can suppress the `[INFO]` messages in the build log by adding the `-q` option to your `mvn` commands. (Don’t forget the one in `cdk.json`.)

**C#**

```bash
dotnet build src
```

**or press F6 in Visual Studio**

---

## Preparing for Deployment

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

```bash
cdk diff
```

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

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

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

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

Deploy the changes.

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

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

HelloCdkStack

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

## Destroying the App's Resources

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

```
cdk destroy
```

Enter `y` to approve the changes and delete any stack resources. In some cases this command fails, such as when a resource isn't empty and must be empty before it can be destroyed. See [Delete Stack Fails](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/delete-stack-fails.html) in the *AWS CloudFormation User Guide* for details.
This topic describes some of the concepts (the why and how) behind the AWS CDK. It also discusses the AWS Construct Library.

AWS CDK apps are composed of building blocks known as Constructs (p. 22), which are composed together to form stacks and apps.

** Constructs

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

A construct can represent a single resource, such as an Amazon Simple Storage Service (Amazon S3) bucket, or it can represent a higher-level component consisting of multiple AWS 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. These constructs represent all of the AWS resources that are available in AWS CloudFormation. CFN Resources are generated from the AWS CloudFormation Resource Specification on a regular basis. They are named CfnXyz, where Xyz represents the 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 resource model.

The next level of constructs also represent AWS resources, but with a higher-level, intent-based API. They provide the same functionality, but handle much of the details, boilerplate, and glue logic required by CFN constructs. 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. 33). Within the app, you typically define one or more stacks (p. 38), 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. 62) 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. 52) 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 s3 = require('@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
    });
  }
}
```
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. 43), 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

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

Python

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

Java

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

    public HelloCdkStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        // ...
    }
}
```

C#

```csharp
public class HelloCdkStack : Stack
{
    public HelloCdkStack(App scope, string id, StackProps props) : base(scope, id, props)
    {
        //...
    }
}
```

Using Constructs

Once you have defined a stack, you can populate it with resources. The following example imports the Amazon S3 module and uses it to define a new Amazon S3 bucket by creating an instance of the `Bucket` class within the current scope (`this` or, in Python, `self`) which, in our case is the `HelloCdkStack` instance.

TypeScript

```typescript
import s3 = require('@aws-cdk/aws-s3');
```
// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
    versioned: true
});

### Python

    from aws_cdk import aws_s3 as s3
    # "self" is HelloCdkStack
    s3.Bucket(self, "MyFirstBucket", versioned=True)

### Java

    import software.amazon.awscdk.services.s3.*;
    public class HelloCdkStack extends Stack {
        public HelloCdkStack(final Construct scope, final String id) {
            this(scope, id, null);
        }
        
        public HelloCdkStack(final Construct scope, final String id, final StackProps props) {
            super(scope, id, props);
            Bucket.Builder.create(this, "MyFirstBucket")
                .versioned(true).build();
        }
    }

### C#

    using Amazon.CDK.AWS.S3;
    // "this" is HelloCdkStack
    new Bucket(this, "MyFirstBucket", new BucketProps
    { Versioned = true
    });

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

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

### Configuration

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

**TypeScript**

    new s3.Bucket(this, 'MyEncryptedBucket', {
        encryption: s3.BucketEncryption.KMS,
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. 87) methods that you can use to grant AWS Identity and Access Management (IAM) permissions on that construct to a principal. The following example grants the IAM group `data-science` permission to read from the Amazon S3 bucket `raw-data`.

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

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

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

**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")
  .code(Code.fromAsset("./create-job-lambda-code"))
  .environment(new HashMap<String, String>() {{
    put("QUEUE_URL", jobsQueue.getQueueUrl());
  }}).build();
```

**C#**

```csharp
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
{
  Runtime = Runtime.NODEJS_10_X,
  Handler = "index.handler",
  Code = Code.FromAsset(@"./create-job-lambda-code"),
  Environment = new Dictionary<string, string>
  {
    ["QUEUE_URL"] = jobsQueue.QueueUrl
  }
})
```
For information about the most common API patterns in the AWS Construct Library, see Resources.

## 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(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")
        topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(topic, s3.NotificationKeyFilter(prefix=prefix))
```

**Java**

```java
public class NotifyingBucket extends Bucket {

    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) {
```
The `NotifyingBucket` constructors have 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');
```

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/' });
```

**Python**

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

**Java**

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

**C#**

```csharp
new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
```

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

**TypeScript**

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

    constructor(scope: Construct, id: string, props: NotifyingBucketProps) {
        super(scope, id);
        const bucket = new s3.Bucket(this, 'bucket');
        this.topic = new sns.Topic(this, 'topic');
        bucket.add_object_created_notification(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(self.topic, { prefix: props.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)
    {
        var bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        if (prefix != null)
            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));

Python

queue = qs.Queue(self, "NewImagesQueue")
images = NotifyingBucket(self, prefix="Images")
images.topic.add_subscription(sns_sub.SqsSubscription(queue))
As described in the section called “Constructs” (p. 22), 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');
  }
}
```

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) {
    this(scope, id, null);
  }

  public MyFirstStack(final Construct scope, final String id, final StackProps props) {
    super(scope, id, props);
    new Bucket(this, "MyFirstBucket");
  }
}
```
The App Construct

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

TypeScript

```typescript
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();

Python

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

MyApp().synth()

Java

// MyApp.java
package com.myorg;
import software.amazon.awscdk.core.App;

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

// Main.java
package com.myorg;

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

C#

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

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

These two methods are equivalent.

App Lifecycle

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

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

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

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

4. Synthesis
   This is the final stage of the execution of your AWS CDK app. It's triggered by a call to `app.synth()`, and it traverses the construct tree and invokes the `synthesize` method on all constructs. Constructs that implement `synthesize` can participate in synthesis and emit deployment artifacts to the resulting cloud assembly. These constructs include AWS CloudFormation templates, AWS Lambda application bundles, file and Docker image assets, and other deployment artifacts. The section called “Cloud Assemblies” (p. 37) 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. 104). 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. 65) 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.

```
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**

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

**Python**

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

**Java**

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

**C#**

```json
{
  "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`.

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

Stacks

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

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

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

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

**TypeScript**

```typescript
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

**Python**

```python
app = App()

MyFirstStack(app, 'stack1')
MySecondStack(app, 'stack2')

app.synth()
```

**Java**

```java
App app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.synth();
```
To list all the stacks in an AWS CDK app, run the `cdk ls` command, which for the previous AWS CDK app would have the following output.

```
stack1
stack2
```

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

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

```
cdk synth stack1
```

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

**Note**

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

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

**TypeScript**

```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) {
```

---

**C#**

```csharp
var app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.Synth();
```
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();
```java
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, "data");
        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();
}
```

```csharp
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, "data");
        new DataPlane(this, "data");
        new Monitoring(this, "mon");
    }
}

class Program {
    static void Main(string[] args)
    {
```
var app = new App();
new MyService(app, "beta");
new MyService(app, "prod", prod: true);
app.Synth();
}
}

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

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

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

**TypeScript**

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

**Python**

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

**Java**

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

**C#**

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

### Stack API

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

- **Stack.of(construct)** – A static method that returns the **Stack** in which a construct is defined. This is useful if you need to interact with a stack from within a reusable construct. The call fails if a stack cannot be found in scope.
- **stack.stackName (Python: stack_name)** – Returns the physical name of the stack. As mentioned previously, all AWS CDK stacks have a physical name that the AWS CDK can resolve during synthesis.
- **stack.region and stack.account** – Return the AWS Region and account, respectively, into which this stack will be deployed. These properties return either the account or Region explicitly specified when the stack was defined, or a string-encoded token that resolves to the AWS CloudFormation...
pseudo-parameters for account and Region to indicate that this stack is environment agnostic. See the section called “Environments” (p. 43) 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.

### 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. 166) 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' };
```
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 the process object.

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

**Note**

TypeScript users must install the DefinitelyTyped NodeJS module with NPM to be able to use process. cdk init now installs this module for you, but if you are working with a project created before it was added, or didn't set up your project using cdk init, install it manually.

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

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

**Python**

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

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

**Java**

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

```
public class MyApp {

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

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

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

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

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

45
Use `System.Environment.GetEnvironmentVariable()` to get the value of an environment variable.

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

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

The AWS CDK distinguishes between not specifying the `env` property at all and specifying it using `CDK_DEFAULT_ACCOUNT` and `CDK_DEFAULT_REGION`. The former implies that the stack should synthesize an environment-agnostic template. Constructs that are defined in such a stack cannot use any information about their environment. For example, you can't write code like `if (stack.region === 'us-east-1')` or use framework facilities like `Vpc.fromLookup` (Python: `from_lookup`), which need to query your AWS account.

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

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

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

```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#

```csharp
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")
    };
}
```

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.

Linux/Mac OS X

```bash
#!/bash
# cdk-deploy-to.sh
export CDK_DEPLOY_ACCOUNT=$1
shift
export CDK_DEPLOY_REGION=$1
shift
cdk deploy "$@"
```
Then you can write additional scripts that call that script to deploy to specific environments (even multiple environments per script):

**Linux/Mac OS X**

```bash
#!/bin/bash
# cdk-deploy-to-test.sh
bash cdk-deploy-to.sh 123456790 us-east-1 "" "$@"
```

**Windows**

```bash
@echo off
rem cdk-deploy-to.bat
set CDK_DEPLOY_ACCOUNT=%1
shift
set CDK_DEPLOY_REGION=%1
shift
cdk deploy %*
```

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.

**Linux/Mac OS X**

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

**Windows**

```bash
@echo off
rem cdk-deploy-to-prod.bat
cdk-deploy-to 135792469 us-west-1 %* || goto :eof
cdk-deploy-to 246813579 eu-west-1 %*
```

Developers would continue to use the normal `cdk deploy` command to deploy to their own AWS environments.

## Resources

As described in the section called “Constructs” (p. 22), 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 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 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
```
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 });
```

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

**Python**

```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**

```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#**

```csharp
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");
```
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',
});
```

**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.
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
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**

When you're working with 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, you'll often need to use unique identifiers. These identifiers can be accessed as attributes on the resources, like this:

**TypeScript**

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

**Python**

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

**Java**

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

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

**C#**

```csharp
var bucket = new Bucket(this, "Bucket");

new Function(this, "MyLambda", new FunctionProps {
  Environment = new Dictionary<string, string>
  {
    ["BUCKET_NAME"] = bucket.BucketName
  }
});
```

## Importing Existing External Resources

Sometimes you already have a resource in your AWS account and want to use it in your AWS CDK app, for example, a resource that was defined through the console, 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 the existing bucket with the ARN `arn:aws:s3:::my-bucket-name`, and a VPC based on the existing VPC with the resource name `booh`.

**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-1234567890abcdef',
});

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. 93) 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

c2.Vpc.fromLookup(this, 'DefaultVpc', {
    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.

### Note

You can use `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. 42), 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.IBucket` 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) {
    // ...
}
```

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. 50)). 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');
```

Python

```python
table.grant(func, "dynamodb:create_backup")
```
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_role_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 cw = require('@aws-cdk/aws-cloudwatch');
import sqs = require('@aws-cdk/aws-sqs');
import { Duration } from '@aws-cdk/core';

const queue = new sqs.Queue(this, 'MyQueue');
const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'MessagesVisible (Approx)',
  period: Duration.minutes(5),
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100,
  // ...
});
```

#### Python

```python
import aws_cdk.aws_cloudwatch as cw
import aws_cdk.aws_sqs as sqs
from aws_cdk.core import Duration

queue = sqs.Queue(self, "MyQueue")
metric = queue.metric_approximate_number_of_messages_not_visible(
  label="Messages Visible (Approx)",
  period=Duration.minutes(5),
  # ...
)
```
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.
**Connectable** 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 asg = require('@aws-cdk/aws-autoscaling');
import ec2 = require('@aws-cdk/aws-ec2');

const fleet: asg.AutoScalingGroup = /* ... */;

// Allow surfing the (secure) web
fleet.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443, toPort: 443 }));

const fleet2: asg.AutoScalingGroup = /* ... */;
fleet.connections.allowFrom(fleet, ec2.Port.AllTraffic());
```

**Python**

```python
import aws_cdk.aws_autoscaling as asg
import aws_cdk.aws_ec2 as ec2

def fleet = asg.AutoScalingGroup( ... )

# Allow surfing the (secure) web
fleet.connections.allow_to(ec2.Peer.any_ipv4(),
    ec2.Port(PortProps(from_port=443, to_port=443)))

fleet2 = asg.AutoScalingGroup( ... )
fleet.connections.allow_from(fleet, 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 fleet = AutoScalingGroup.Builder.create(this, "MyFleet")
    /* ... */ .build();

// Allow surfing the (secure) Web
fleet.getConnections().allowTo(Peer.anyIpv4(),
    Port.Builder.create().fromPort(443).toPort(443).build());

AutoScalingGroup fleet2 = AutoScalingGroup.Builder.create(this, "MyFleet2")
    /* ... */ .build();
fleet2.getConnections().allowFrom(fleet, 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 fleet = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
    { ... });
```
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
const listener = ec2.Listener(this, 'Listener', { ... });
const fleet = new asg.AutoScalingGroup(this, 'MyFleet', { ... });
fleet.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");
fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

**Python**

```python
listener.connections.allow_default_port_from_any_ipv4("Allow public access")
fleet.connections.allow_to_default_port(rds_database, "Fleet can access database")
```

**Java**

```java
listener.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");
fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

**C#**

```csharp
listener.Connections.AllowDefaultPortFromAnyIpv4("Allow public access");
fleet.Connections.AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

### Amazon CloudWatch Events

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 simplified way to register a handler for a common event type.

The following example shows how to trigger a Lambda function when an object is added to an Amazon S3 bucket.

**TypeScript**

```typescript
import 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
bucket.add_object_created_notification(lambda_function)
```
Identifiers

The AWS CDK deals with many types of identifiers and names. To use the AWS CDK effectively and avoid errors, you need to understand the types of identifiers.

Identifiers must be unique within the scope in which they are created; they do not need to be globally unique in your AWS CDK application.

If you attempt to create an identifier with the same value within the same scope, the AWS CDK throws an exception.

Construct IDs

The most common identifier, id, is the identifier passed as the second argument when instantiating a construct object. This identifier, like all identifiers, need only be unique within the scope in which it is created, which is the first argument when instantiating a construct object.

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

class MyStack extends Stack {
    constructor(scope: Construct, id: string, props: StackProps = {}) {
```
Construct IDs

```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");
    }
}
```

```csharp
using core = Amazon.CDK;
```
using s3 = Amazon.CDK.AWS.S3;

public class MyStack : core.Stack
{
    public MyStack(core.App scope, string id, core.IStackProps props) : base(scope, id, props)
    {
        new s3.Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new core.App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}

Paths

As the constructs in an AWS CDK application form a hierarchy, we refer to the collection of IDs from a given construct, then of its parent construct, then grandparent construct, and so on up to the root of the construct tree, which is an instance of the App class, as a path.

The AWS CDK typically displays paths in your templates as a string, with the IDs from the levels separated by slashes, starting at the node just below the root App instance, which is usually a stack. For example, the paths of the two Amazon S3 bucket resources in the previous code example are Stack1/MyBucket and Stack2/MyBucket.

You can access the path of any construct 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;
```

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 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;
```

**Python**

```python
uid = my_construct.node.unique_id
```

**Java**

```java
String uid = myConstruct.getNode().getUniqueId();
```

**C#**

```c#
string uid = 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. 35)). 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.

**TypeScript**

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

const fn = new lambda.Function(stack, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  }
});
```

**Python**

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

fn = lambda_.Function(stack, "MyLambda",
  environment=dict(BUCKET_NAME=bucket.bucket_name))
```

**Java**

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

Function fn = Function.Builder.create(this, "MyLambda")
  .environment(new HashMap<String, String>() {{
      put("BUCKET_NAME", bucket.getBucketName());
  }}).build();
```

**C#**

```csharp
var bucket = new s3.Bucket(this, "MyBucket");

var fn = new Function(this, "MyLambda", new FunctionProps {
  Environment = new Dictionary<string, string> {
    ["BUCKET_NAME"] = bucket.BucketName
  }
});
```

When the AWS CloudFormation template is finally synthesized, the token is rendered as the AWS CloudFormation intrinsic `{ "Ref": "MyBucket" }`. At deployment time, AWS CloudFormation replaces this intrinsic with the actual name of the bucket that was created.

### Tokens and Token Encodings

Tokens are objects that implement the `IResolvable` interface, which contains a single `resolve` method. The AWS CDK calls this method during synthesis to produce the final value for the AWS CloudFormation template. Tokens participate in the synthesis process to produce arbitrary values of any type.

**Note**

You’ll hardly ever work directly with the `IResolvable` interface. You will most likely only see string-encoded versions of tokens.
Other functions typically only accept arguments of basic types, such as `string` or `number`. To use tokens in these cases, you can encode them into one of three types using static methods on the `core.Token` class.

- Strings using `Token.asString` (Python: `as_string`)
- List of strings using `Token.asList` (Python: `as_list`)
- Number (float) using `Token.asNumber` (Python: `as_number`)

These take an arbitrary value, which can also be an `IResolvable` interface, and encode them into a primitive value of the appropriate 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`);
}
```

**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 the error could 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.

---

67
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';
```

**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`;
```

**Python**

```python
function_name = f"{bucket.bucket_name}Function"
```

**Java**

```java
String functionName = String.format("%sFunction". bucket.getBucketName());
```

**C#**

```csharp
string functionName = "{bucket.bucketName}Function";
```

Avoid manipulating the string in other ways. For example, taking a substring of a string is likely to break the string token.

### List-Encoded Tokens

List-encoded tokens look like the following

```json
["#{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 attributes, 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

```typescript
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

Python

```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

```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;

public class NumberProducer : INumberProducer {
    Func<Double> function;
    public NumberProducer(Func<Double> function)
    {
        this.function = function;
    }
    public Double Produce(IResolveContext context)
    {
        return function();
    }
}

double actualValue = 0;

new AutoScalingGroup(this, "Group", new AutoScalingGroupProps
{
    DesiredCapacity = Lazy.NumberValue(new NumberProducer(() => actualValue))
});

// At some later point
actualValue = 10;

## Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method `stack.toJsonString`, as shown in the following example.

**TypeScript**

```typescript
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))
```
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.

**Note**
Tagging is implemented using the section called “Aspects” (p. 97). 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');
```

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');
```
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 priority of 200. To change the priority of applying a tag, pass a `priority` property to `Tag.add()` or `Tag.remove()`.

The following applies a tag with a priority of 300 to a construct.

TypeScript

```typescript
Tag.add(myConstruct, 'key', 'value', {
  priority: 300
});
```

Python

```python
Tag.add(my_construct, "key", "value", priority=300)
```

Java

```java
Tag.add(myConstruct, "key", "value", TagProps.builder()
  .priority(300).build());
```

C#

```csharp
Tag.Add(myConstruct, "key", "value", new TagProps { Priority = 300 });
```

**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`.

TypeScript

```typescript
Tag.add(myConstruct, 'tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
});
```
Tag.remove()

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::Yzz in the construct, but not from resources of type AWS::Xxx::Zzz.

TypeScript

```typescript
Tag.remove(myConstruct, 'tagname', {
  priority: 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 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", TagProps.builder()
    .exclude_resource_types=["AWS::EC2::Subnet"]
    .build());
```

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#

```c#
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']});
```

Python

```python
Tag.add(the_best_stack, "StackType", "TheBest",
    { include_resource_types: ['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. 37) 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 and 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. 37) 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')
});
```

**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#

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.

```python
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 cdk = require('@aws-cdk/core');
import lambda = require('@aws-cdk/aws-lambda');
import path = require('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'
    });
  }
}
```
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
        {
            "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. 49) 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 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#

using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps {
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png")});

new Function(this, "myLambdaFunction", new FunctionProps {
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionary<string, string> {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_URL"] = imageAsset.S3Url
    })
});

Permissions

If you use Amazon S3 assets directly through the aws-s3-assets module, IAM roles, users, or groups, and need to read assets in runtime, grant those assets IAM permissions through the asset.grantRead method.

The following example grants an IAM group read permissions on a file asset.

TypeScript

import { Asset } from '@aws-cdk/aws-s3-assets';
import path = require('path');

const asset = new Asset(this, 'MyFile', {
    path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);

Python

from aws_cdk.aws_s3_assets import Asset
from aws_cdk import aws_iam as iam

import os.path
dirname = os.path.dirname(__file__)

    asset = Asset(self, "MyFile",
        path=os.path.join(dirname, "my-image.png"))

    group = iam.Group(self, "MyUserGroup")
    asset.grantRead(group)

Java

import java.io.File;

import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

    Asset asset = Asset.Builder.create(this, "SampleAsset")
        .path(new File(startDir, "images/my-image.png").toString()).build();

    Group group = new Group(this, "MyUserGroup");
    asset.grantRead(group);
    }
}

C#

using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @$"images\my-image.png")))
});

var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
Docker Image Assets

The AWS CDK supports bundling local Docker images as assets through the `aws-ecr-assets` module.

The following example defines a docker image that is built locally and pushed to Amazon ECR. Images are built from a local Docker context directory (with a Dockerfile) and uploaded to Amazon ECR by the AWS CDK CLI or your app’s CI/CD pipeline, and can be naturally referenced in your AWS CDK app.

TypeScript

```typescript
import { DockerImageAsset } from '@aws-cdk/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

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. 49) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command `cdk synth` displays asset properties as AWS CloudFormation parameters.

Amazon ECS Task Definition Example

A common use case is to create an Amazon ECS TaskDefinition to run docker containers. The following example specifies the location of a Docker image asset that the AWS CDK builds locally and pushes to Amazon ECR.
### TypeScript

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

TypeScript

```typescript
import ecs = require('@aws-cdk/aws-ecs');
import path = require('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)
});
```

Python

```python
import aws_cdk.aws_ecs as ecs
from aws_cdk.aws_ecr_assets import DockerImageAsset
import os.path
dirname = os.path.dirname(__file__)
asset = DockerImageAsset(self, 'my-image',
    directory=os.path.join(dirname, '..', "demo-image"))
task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024, cpu=512)
task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.fromEcrRepository(
        asset.repository, asset.image_uri))
```

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();

taskDefinition.addContainer("my-other-container",
    ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
        asset.getRepository(), asset.getImageUri())).build());

C# using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.Ecr.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
});

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)
});

Build Arguments Example

You can provide customized build arguments for the Docker build step through the buildArgs (Python: build_args) property option when the AWS CDK CLI builds the image during deployment.

TypeScript

const asset = new DockerImageAsset(this, 'MyBuildImage', {
    directory: path.join(__dirname, 'my-image'),
    buildArgs: {
        HTTP_PROXY: 'http://10.20.30.2:1234'
    }
});

Python

asset = DockerImageAsset(self, "MyBuildImage",
    directory=os.path.join(dirname, "my-image"),
    build_args=dict(HTTP_PROXY="http://10.20.30.2:1234"))

Java

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
    .directory(new File(startDir, "my-image").toString())
    .buildArgs(new HashMap<String, String>() {{
        put("HTTP_PROXY", "http://10.20.30.2:1234");
    }}).build();

C# var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps {

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. 171) 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.

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 with grant. For example, Amazon S3 buckets have the methods grantRead and grantReadWrite (Python: grant_read, grant_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 (bucket.grantRead(lambda), or `grant_read` in Python) instead of granting access to their role.

**Roles**

The IAM package contains a `Role` construct that represents IAM roles. The following code creates a new role, trusting the Amazon EC2 Service Principal.

TypeScript

```typescript
import 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
});
```

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**

```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',
  }}));
```

**Python**

```python
def 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**

```java
role.addPolicy(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#**

```csharp
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
  Effect = Effect.DENY,
  Resources = [bucket.BucketArn, otherRole.RoleArn],
  Actions = ["ec2:SomeAction", "s3:Anotheraction"],
  Conditions = new Dictionary<string, object>
  {
    ["StringEquals"] = new Dictionary<string, string>
    {
      ["ec2:AuthorizedService"] = "codebuild.amazonaws.com"
    }
  }
}));
```

In our example above, we've created a new `PolicyStatement` inline with the `addToPolicy` (Python: `add_to_policy`) call. You can also pass in an existing policy statement or one you've modified. The `PolicyStatement` object has numerous methods for adding principals, resources, conditions, and actions.

If you're using a construct that requires a role to function correctly, you can either pass in an existing role when instantiating the construct object, or let the construct create a new role for you, trusting the appropriate service principal. The following example uses such a construct: a CodeBuild project.
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:

```typescript
import 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: 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,
});
```

```python
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
  # if role is None, the Project creates a new default role,
  # trusting the codebuild.amazonaws.com service principal
  role=some_role)
```

```java
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
  .role(someRole).build();
```

```csharp
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 });
```
add_to_policy) method of the role property otherwise, saving you the trouble of handling the undefined case explicitly. The following example demonstrates:

**TypeScript**

```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
  }));
```

**Python**

```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(new iam.PolicyStatement(
    effect=iam.Effect.ALLOW,   # ... and so on defining the policy
))
```

**Java**

```java
// project is imported into the CDK application
Project project = Project.fromProjectName(this, "Project", "ProjectName");

// project is imported, so project.getRole() is null, and this call has no effect
project.addToRolePolicy(PolicyStatement.Builder.create()
    .effect(Effect.ALLOW)   // ... and so on defining the policy
    .build();
```

**C#**

```csharp
// project is imported into the CDK application
var project = Project.FromProjectName(this, "Project", "ProjectName");

// project is imported, so project.role is null, and this call has no effect
project.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW, // ... and so on defining the policy
}));
```

**Resource Policies**

A few resources in AWS, such as Amazon S3 buckets and IAM roles, also have a resource policy. These constructs have an `addToResourcePolicy` method (Python: `add_to_resource_policy`), which takes a `PolicyStatement` as its argument. Every policy statement added to a resource policy must specify at least one principal.

In the following example, the Amazon S3 bucket grants a role with the `s3:SomeAction` permission to itself.

**TypeScript**

```typescript
bucket.addToResourcePolicy(new iam.PolicyStatement({
```
Principals

The AWS CDK 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. Account principals (new iam.AccountPrincipal('0123456789012'))
4. Canonical user principals (new iam.CanonicalUserPrincipal('79a59d[...]7ef2be'))
5. AWS organizations principals (new iam.OrganizationPrincipal('org-id'))
6. Arbitrary ARN principals (new iam.ArnPrincipal(res.arn))
7. An iam.CompositePrincipal(principal1, principal2, ...) to trust multiple principals

Runtime Context

The AWS CDK uses context to retrieve information such as the Availability Zones in your account or Amazon Machine Image (AMI) IDs used to start your instances. Context entries are key-value pairs.

To avoid unexpected changes to your deployments when, for example, a new Amazon Linux AMI is released, thus changing your Auto Scaling group, the AWS CDK stores context values in the
cdk.context.json file within your project. This ensures that the AWS CDK uses the same context values the next time it synthesizes your app. Don't forget to put this file under version control.

**Construct Context**

Context values are made available to your AWS CDK app in five different ways:

- Automatically from the current AWS account.
- Through the `--context` option to the `cdk` command.
- In the `context` key of the project's `cdk.json` file.
- In the `context` key of a `~/.cdk.json` file.
- In code using the `construct.node.getContext` method.

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), either automatically 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. It will be refreshed on the next CDK synthesis run.
```

To remove a context value, run `cdk context --reset`, specifying the value's corresponding key number. The following example removes the value that corresponds to the key value of 2 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
```

Example

Below is an example of importing an existing Amazon VPC using AWS CDK context.

TypeScript

```typescript
import cdk = require('@aws-cdk/core');
import ec2 = require('@aws-cdk/aws-ec2');

export class ExistsVpcStack extends cdk.Stack {
```
```python
import aws_cdk.core as cdk
import aws_cdk.aws_ec2 as ec2

class ExistsVpcStack(cdk.Stack):
    def __init__(self, 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.from_lookup(this, "VPC",
                                      VpcLookupOptions.builder()
                                      .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.select_subnets(SubnetSelection.builder().
                                                          subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
                               .value(pubSubNets.getSubnetIds().toString()).build();
    }
}
```
C# using Amazon.CDK; using Amazon.CDK.AWS.EC2;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(App scope, string id, StackProps props) : base(scope, id, props)
    {
        var vpcId = (string)this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
        {
            VpcId = vpcId
        });

        SelectedSubnets pubSubNets = vpc.SelectSubnets([new SubnetSelection
        {
            SubnetType = SubnetType.PUBLIC
        }]);

        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
}

You can use cdk diff to see the effects of passing in a context value on the command line:

$ cdk diff -c vpcid=vpc-0cb9c31031d0d3e22

Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
   {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acbf38f3128f"}

The resulting context values can be viewed as shown here.

$ cdk context -j

{
    "vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-east-1":
    {
        "vpcId": "vpc-0cb9c31031d0d3e22",
        "availabilityZones": [
            "us-east-1a",
            "us-east-1b"
        ],
        "privateSubnetIds": [
            "subnet-03ecfc033225be285",
            "subnet-0ced5da53180ebfa"
        ],
        "privateSubnetNames": [
            "Private"
        ],
        "privateSubnetRouteTableIds": [
            "rtb-0e955393ced0ada04",
            ...
Aspects

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(...));
```

**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. 71), 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;
}
```
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. 35), 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());
```

Python

```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.isResolvable(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)
    {
        // See that we're dealing with a CfnBucket
        if (node is CfnBucket)
        {
            var bucket = (CfnBucket)node;
            if (bucket.VersioningConfiguration is null ||
                !Tokenization.IsResolvable(bucket.VersioningConfiguration) &&
                !bucket.VersioningConfiguration.ToString().Contains("Enabled")
            bucket.Node.AddError("Bucket versioning is not enabled");
        }
    }
}

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',
      // ...
    }
  ]
});
```

Python

```python
s3.CfnBucket(self, "MyBucket",
    analytics_configurations: [
        dict(id="Config",
            # ...
        )
    ]
)
```

Java

```java
CfnBucket.Builder.create(this, "MyBucket")
    .analyticsConfigurations(Arrays.asList(new HashMap<String, String>() {
        put("id", "Config");
        // ...
    }))).build();
```

C#

```csharp
new CfnBucket(this, 'MyBucket', new CfnBucketProps {
  AnalyticsConfigurations = new Dictionary<string, string> {
    ["id"] = "Config",
    // ...
  }
});
```
In the rare case where you want to define a resource that doesn't have a corresponding `CfnXxx` class, such as a new resource type that hasn't yet been published in the AWS CloudFormation resource specification, you can instantiate the `cdk.CfnResource` directly and specify the resource type and properties. This is shown in the following example.

TypeScript

```typescript
new cdk.CfnResource(this, 'MyBucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config',
        // ...
      }
    ]
  }
});
```

Python

```python
cdk.CfnResource(self, 'MyBucket',
  type="AWS::S3::Bucket",
  properties=dict(
    # Note the PascalCase here! These are CloudFormation identifiers.
    "AnalyticsConfigurations": [
      {
        "Id": "Config",
        # ...
      }
    ]
  )
)
```

Java

```java
CfnResource.Builder.create(this, "MyBucket")
  .type("AWS::S3::Bucket")
  .properties(new HashMap<String, Object>() {
    // Note the PascalCase here! These are CloudFormation identifiers
    put("AnalyticsConfigurations", Arrays.asList(
      new HashMap<String, String>() {
        put("Id", "Config");
        // ...
      })));
}).build();
```

C#

```csharp
new CfnResource(this, "MyBucket", new CfnResourceProps {
  Type = "AWS::S3::Bucket",
  Properties = new Dictionary<string, object>
  {
    // Note the PascalCase here! These are CloudFormation identifiers
    ["AnalyticsConfigurations"] = new List<Dictionary<string, string>>
```
Modifying the AWS CloudFormation Resource behind AWS Constructs

If an Construct is missing a feature or you are trying to work around an issue, you can modify the CFN Resource that is encapsulated by the Construct.

All Constructs contain within them the corresponding CFN Resource. For example, the high-level `Bucket` construct wraps the low-level `CfnBucket` construct. Because the `CfnBucket` corresponds directly to the AWS CloudFormation resource, it exposes all features that are available through AWS CloudFormation.

The basic approach to get access to the CFN Resource class is to use `construct.node.defaultChild` (Python: `default_child`), cast it to the right type (if necessary), and modify its properties. Again, let's take the example of a `Bucket`.

**TypeScript**

```typescript
// Get the AWS CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
    {
        id: 'Config',
        // ...
    }
];
```

**Python**

```python
# Get the AWS CloudFormation resources
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
    {
        "id": "Config",
        # ...
    }
]
```

**Java**

```java
cfnBucket.setAnalyticsConfigurations(
    Arrays.asList(
        new HashMap<String, String>() {
            put("Id", "Config");
            // ...
        })));
```
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_offset 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');

Python

```python
# Get the AWS CloudFormation resource
cfn_bucket = bucket.node.default_child

# Use dot notation to address inside the resource template fragment
cfn_bucket.add_deletion_override("Properties.VersioningConfiguration.Status")

# add_property_override is a convenience function, which implies the
# path starts with "Properties."
cfn_bucket.add_property_override("VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_property_deletion_override("VersioningConfiguration.Status")
```

C#

```csharp
// Get the AWS CloudFormation resource
var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment

// 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 and Stability Model

Version numbers consist of three numeric version parts: major.minor.patch, and adhere to the semantic versioning model. This means that breaking changes to stable APIs, which is described in the next section, 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.

AWS CDK Stability Index

However, certain APIs do not adhere to the semantic versioning model. The AWS CDK team has added a stability index to help developers determine which APIs deviate from the semantic versioning model.

There are three levels of stability in the AWS CDK Construct Library:

Stable

The API is subject to the semantic versioning model. We will not introduce non-backward-compatible changes or remove the API in a subsequent patch or feature release.

CloudFormation Only

These APIs are automatically built from the AWS CloudFormation resource specification and are subject to any changes introduced by AWS CloudFormation.

Experimental

The API is still under active development and subject to non-backward-compatible changes or removal in any future version. We recommend that you do not use this API in production environments. Experimental APIs are not subject to the semantic versioning model.

Deprecated

The API may emit warnings. We do not guarantee backward compatibility.

Experimental and stable modules receive the same level of support from AWS. The only difference is that we might change experimental APIs within a major version. Although we don’t recommend using experimental APIs in production, we vet them the same way as we vet stable APIs before we include them in an AWS CDK release.

Identifying the Support Level of an API

Each module in the API Reference starts with a section outlining the module's stability index. The libraries that include only AWS CloudFormation resources, and no hand-curated constructs, are labeled with the maturity indicator CloudFormation-only.

The module level gives an indication of the stability of the majority of the APIs included in the module, however, individual APIs within the module can be annotated with different stability levels.
Language Binding Stability

In addition to modules of the AWS CDK Construct Library, language support is also subject to a stability indication. 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.

<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>
<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>

AWS CDK in Other Languages

In some cases the example code in the AWS CDK documentation is available only in TypeScript. This topic describes how to read TypeScript code and translate it into other languages. See Hello World Tutorial (p. 13) for an example of creating an AWS CDK app in a supported language.

Importing a Module

Both TypeScript and Python support 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). Replace the dashes in the TypeScript module name with underscores to get the Python module name.

The following is how you import the entire Amazon S3 module or just a `Stack` class in both languages.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>// Import entire module import s3 = require('@aws-cdk/aws-s3')</code></td>
<td><code># Import entire module import aws_cdk.aws_s3 as s3</code></td>
</tr>
<tr>
<td><code>// Selective import import { Stack } from '@aws-cdk/core';</code></td>
<td><code># Selective import from aws_cdk.core import Stack</code></td>
</tr>
</tbody>
</table>
Instantiating a Class

Classes have the same name in TypeScript and in Python. TypeScript uses `new` to instantiate classes, whereas in Python you call the class object directly, like a function. Props objects at the end of an argument list become keyword-only arguments in Python, and their names become `snake_case`. The keyword `this` in TypeScript translates to `self` in Python.

The following table shows how you can translate TypeScript class instantiations to Python class instantiations.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
</table>
| // Instantiate Bucket class
  const bucket = new s3.Bucket(this, 'Bucket'); | # Instantiate Bucket class
  bucket = s3.Bucket(self, 'Bucket') |
| // Instantiate Bucket with props
  const bucket = new s3.Bucket(this, 'Bucket', {
    bucketName: 'my-bucket',
    versioned: true,
  }); | # Instantiate Bucket with props
  bucket = s3.Bucket(self, 'Bucket',
    bucket_name='my-bucket',
    versioned=True) |

Methods

Methods and argument names in TypeScript are camelCased, whereas in Python they are snake_cased. Props objects at the end of an argument list in TypeScript are translated into keyword-only arguments in Python, and their names become `snake_case`.

The following table shows how you can translate TypeScript methods to Python methods.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
</table>
| // Call method
  bucket.addCorsRule({
    allowedOrigins: ['*'],
    allowedMethods: [],
  }); | # Call method
  bucket.add_cors_rule(
    allowed_origins=['*'],
    allowed_methods=[] |

Enum Constants

Enum constants are scoped to a class, and have uppercase names with underscores in both languages (sometimes referred to as `SCREAMING_SNAKE_CASE`). TypeScript enum constants and Python enum constants are identical.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
</table>
Defining Constructs

In TypeScript, a construct's props are defined with a shape-based interface, whereas Python takes keyword (or keyword-only, see PEP3102) arguments.

The following table shows how TypeScript construct definitions translate to Python construct definitions.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface MyConstructProps {</td>
<td>class MyConstruct(Construct):</td>
</tr>
<tr>
<td>prop1: number;</td>
<td>def <strong>init</strong>(scope, id, *, prop1,</td>
</tr>
<tr>
<td>prop2?: number;</td>
<td>prop2=10):</td>
</tr>
<tr>
<td>}</td>
<td>super().<strong>init</strong>(scope, id)</td>
</tr>
<tr>
<td>class MyConstruct extends Construct {</td>
<td># Construct contents here</td>
</tr>
<tr>
<td>constructor(scope: Construct, id: string, props: MyConstructProps) {</td>
<td></td>
</tr>
<tr>
<td>super(scope, id);</td>
<td></td>
</tr>
<tr>
<td>const prop2 = props.prop2 !== undefined ? props.prop2 : 10;</td>
<td></td>
</tr>
<tr>
<td>// Construct contents here</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Structs (Interfaces)

Structs are TypeScript interfaces that represent a set of values. You can recognize them because their name doesn't start with an `I`, and all of their fields are read-only.

In TypeScript, structs are passed as object literals. In Python, if the struct is the last argument to a method, its fields are lifted into the method call itself. If the argument list contains nested structs, wrap them in a class named after the struct.

The following table shows how to call a method with two levels of structs.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucket.addLifecycleRule({</td>
<td>bucket.add_lifecycle_rule(</td>
</tr>
<tr>
<td>transitions: [</td>
<td>transitions=[</td>
</tr>
<tr>
<td>{</td>
<td>Transition(</td>
</tr>
<tr>
<td>storageClass: StorageClass.GLACIER,</td>
<td>transition_after=Duration.days(10)</td>
</tr>
<tr>
<td>transitionAfter: Duration.days(10)</td>
<td>)</td>
</tr>
<tr>
<td>]</td>
<td>)</td>
</tr>
<tr>
<td>});</td>
<td>}</td>
</tr>
</tbody>
</table>

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`.  
Typically, Python users don't explicitly indicate that a class implements an interface. However, for the AWS CDK you can do this by decorating your class with `@jsii.implements(interface)`.

<table>
<thead>
<tr>
<th>TypeScript</th>
<th>Python</th>
</tr>
</thead>
</table>
| import {IAspect, IConstruct } from '@aws-cdk/core';
class MyAspect implements IAspect {
public visit(node: IConstruct) {
  console.log('Visited', node.node.path);
}
} | from aws_cdk.core import IAspect, IConstruct
@jsii.implements(IAspect)
class MyAspect():
def visit(self, node: IConstruct) -> None:
  print("Visited", node.node.path) |
Examples

This topic contains the following examples:

- Creating a Serverless Application Using the AWS CDK (p. 110) Creates a serverless application using Lambda, API Gateway, and Amazon S3.
- Creating an AWS Fargate Service Using the AWS CDK (p. 123) Creates an Amazon ECS Fargate service from an image on DockerHub.
- Creating a Code Pipeline Using the AWS CDK (p. 129) 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

```sh
mkdir MyWidgetService
cd MyWidgetService
cdk init --language typescript
```
Create a AWS CDK App

Python

```bash
mkdir MyWidgetService
cd MyWidgetService
cdk init --language python
source .env/bin/activate || "on Windows, use: .env\Scripts\activate.bat"
pip install -r requirements.txt
```

Java

```bash
mkdir MyWidgetService
cd MyWidgetService
cdk init --language java
```

You may now import the Maven project into your IDE.

C#

```bash
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

**Python**

- `app.py` – Main entry point for the application
- `my_widget_service/my_widget_service_stack.py` – Defines the widget service stack

**Java**

- `src/main/java/com/myorg/MyWidgetServiceApp.java` – Main entry point for the application
- `src/main/java/com/myorg/MyWidgetServiceStack.java` – Defines the widget service stack

**C#**

- `src/MyWidgetService/Program.cs` – Main entry point for the application
- `src/MyWidgetService/MyWidgetServiceStack.cs` – Defines the widget service stack

Build the app and note that it synthesizes an empty stack.

**TypeScript**

```bash
npm run build
cdk synth
```
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
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,
                    body: JSON.stringify(body)
                }
            }
        }
    } catch (err) {
        console.error(err.stack);
    }
};
```

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

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

Create a Lambda Function to List All Widgets

The next step is to create a Lambda function to list all of the widgets in our Amazon S3 bucket. 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
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,
                    body: JSON.stringify(body)
                }
            }
        }
    } catch (err) {
        console.error(err.stack);
    }
};
```
// 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.

TypeScript

```sh
npm run build
cdk synth
```

Python

```sh
cdk synth
```

Java

```sh
mvn compile
cdk synth
```

**Note**

Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.

C#

```sh
dotnet build src
cdk synth
```

**Note**

Instead of issuing `dotnet build`, you can instead press F6 in Visual Studio.

# Creating a Widget Service

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

TypeScript

```sh
npm install @aws-cdk/aws-apigateway @aws-cdk/aws-lambda @aws-cdk/aws-s3
```
Creating a Widget Service

Python

```
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:

```
apigateway
lambda
s3
```

C#

Choose Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio and add the following packages.

```
Amazon.CDK.AWS.ApiGateway
Amazon.CDK.AWS.Lambda
Amazon.CDK.AWS.S3
```

Tip

If you don't see these packages in the Browse tab of the Manage Packages for Solution page, make sure the Include prerelease checkbox is ticked.

For a better experience, also add the Amazon.Jsii.Analyzers package to provide compile-time checks for missing required properties.

Create a new source file to define the widget service with the source code shown below.

TypeScript

File: lib/widget_service.ts

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

export class WidgetService extends 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_8_10, // 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 /

```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

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 {
```
城县 a Widget Service

```java
@SuppressWarnings("serial")
public WidgetService(Construct scope, String id) {
    super(scope, id);

    Bucket bucket = new Bucket(this, "WidgetStore");

    Function handler = Function.Builder.create(this, "WidgetHandler")
        .runtime(Runtime.NODEJS_10_X)
        .code(Code.fromAsset("resources"))
        .handler("widgets.main")
        .environment(new HashMap<String, String>() {
            put("BUCKET", bucket.getBucketName());
        }).build();

    bucket.grantReadWrite(handler);

    RestApi api = RestApi.Builder.create(this, "Widgets-API")
        .restApiName("Widget Service").description("This service services widgets.")
        .build();

    LambdaIntegration getWidgetsIntegration =
        LambdaIntegration.Builder.create(handler)
        .requestTemplates(new HashMap<String, String>() {
            put("application/json", "\"statusCode\": \"200\"");
        }).build();

    api.getRoot().addMethod("GET", getWidgetsIntegration);
}
```

C#

```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
    {
        public WidgetService(Construct scope, string id) : base(scope, id)
        {
            var bucket = new Bucket(this, "WidgetStore");

            var handler = new Function(this, "WidgetHandler", new FunctionProps
            {
                Runtime = Runtime.NODEJS_10_X,
                Code = Code.FromAsset("resources"),
                Handler = "widgets.main",
                Environment = new Dictionary<string, string>
                {
                    ["BUCKET"] = bucket.BucketName
                }
            });

            bucket.GrantReadWrite(handler);

            var api = new RestApi(this, "Widgets-API", new RestApiProps
            {
```
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
RestApiName = "Widget Service",
Description = "This service services widgets."
});

var getWidgetsIntegration = new LambdaIntegration(handler, new
LambdaIntegrationOptions
{
    RequestTemplates = new Dictionary<string, string>
    {
        ["application/json"] = "{ "statusCode": "200" }
    }
});

api.Root.AddMethod("GET", getWidgetsIntegration);
```

Save the app and make sure it still synthesizes an empty stack.

**TypeScript**

```
npm run build
cdk synth
```

**Python**

```
cdk synth
```

**Java**

```
mvn compile
cdk synth
```

*Note*

Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.

**C#**

```
dotnet build src
cdk synth
```

*Note*

Instead of issuing `dotnet build`, you can instead press F6 in Visual Studio.

### Add the Service to the App

To add the widget service to our AWS CDK app, we'll need to modify the source file that defines the stack to instantiate the service construct.

**TypeScript**

File: lib/my_widget_service-stack.ts

Add the following line of code after the existing `import` statement.
import widget_service = require('../lib/widget_service');

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

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

Python

File: my_widget_service/my_widget_service_stack.py

Add the following line of code after the existing import statement.

from . import widget_service

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

widget_service.WidgetService(self, "Widgets")

Java

File: src/src/main/java/com/myorg/MyWidgetServiceStack.java

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

new WidgetService(this, "Widgets");

C#

File: src/MyWidgetService/MyWidgetServiceStack.cs

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

new WidgetService(this, "Widgets");

Be sure the app builds and synthesizes a stack (we won't show the stack here: it's over 250 lines).

TypeScript

npm run build
cdk synth

Python

cdk synth

Java

mvn compile
cdk synth

Note

Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.
### 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 bootstrap section of the AWS CDK Tools (p. 166) (if you've already bootstrapped, you'll get a warning and nothing will change).

```bash
cdk bootstrap
```

Now we're ready to deploy the app as follows.

```bash
cdk deploy
```

If the deployment succeeds, save the URL for your server. This URL appears in one of the last lines in the window, where `GUID` is an alphanumeric GUID and `REGION` is your AWS Region.

```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 existing `exports.main` function in `widgets.js` (in `resources`) with the following code.

```javascript
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
};

try {
    return {
        statusCode: 200,
        headers: {},
        body: body
    }
} 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

```typescript
const widget = api.root.addResource("{id}");

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

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

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

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

Python

File: my_widget_service/widget_service.py

```python
widget = api.root.add_resource("{id}")

# Add new widget to bucket with: POST /{id}
post_widget_integration = apigateway.LambdaIntegration(handler)

# Get a specific widget from bucket with: GET /{id}
```
Add the Individual Widget Functions

```python
get_widget_integration = apigateway.LambdaIntegration(handler)

# Remove a specific widget from the bucket with: DELETE /{id}
delete_widget_integration = apigateway.LambdaIntegration(handler)

widget.add_method("POST", post_widget_integration);      # POST /{id}
widget.add_method("GET", get_widget_integration);        # GET /{id}
widget.add_method("DELETE", delete_widget_integration); # DELETE /{id}
```

Java

File: src/src/main/java/com/myorg/WidgetService.java

```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}");

// Add new widget to bucket with: POST /{id}
var postWidgetIntegration = new LambdaIntegration(handler);

// Get a specific widget from bucket with: GET /{id}
var getWidgetIntegration = new LambdaIntegration(handler);

// Remove a specific widget from the bucket with: DELETE /{id}
var deleteWidgetIntegration = new LambdaIntegration(handler);

widget.AddMethod("POST", postWidgetIntegration);      // POST /{id}
widget.AddMethod("GET", getWidgetIntegration);        // GET /{id}
widget.AddMethod("DELETE", deleteWidgetIntegration); // DELETE /{id}
```

Save, build, and deploy the app.

TypeScript

```
npm run build
cdk deploy
```

Python

```
cdk deploy
```

Java

```
mvn compile
```
cdk deploy

**Note**
Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.

C#
dotnet build src
cdk deploy

**Note**
Instead of issuing `dotnet build`, you can instead press F6 in Visual Studio.

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/example'
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/example'
```

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
```

Python

```bash
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language python
source .env/bin/activate || "on Windows, use: .env\Scripts\activate.bat"
pip install -r requirements.txt
```
Add the Amazon EC2 and Amazon ECS Packages

Install the AWS construct library modules for Amazon EC2 and Amazon ECS.

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

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/

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

TypeScript

npm run build
cdk synth

Python

cdk synth

Java

mvn compile
cdk synth

Note
Instead of issuing mvn compile, you can instead press Control-B in Eclipse.

C#

dotnet build src
cdk synth

Note
Instead of issuing dotnet build, you can instead press F6 in Visual Studio.

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.)
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

```
import ec2 = require('@aws-cdk/aws-ec2');
import ecs = require('@aws-cdk/aws-ecs');
import ecs_patterns = require('@aws-cdk/aws-ecs-patterns');
```
Replace the comment at the end of the constructor with the following code.
Create a Fargate Service

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 512
        PublicLoadBalancer = true // Default is false
    })
;
```

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

TypeScript

```typescript
npm run build
cdk deploy
```

Python

```python
cdk deploy
```
Clean Up

Java

```java
mvn compile
cdk deploy
```

**Note**
Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.

C#

```csharp
dotnet build src
cdk deploy
```

**Note**
Instead of issuing `dotnet build`, you can instead press F6 in Visual Studio.

The stack is hundreds of lines, so we don't show it here. The stack should contain one default instance, a private subnet and a public subnet for the three Availability Zones, and a security group.

Deploy the stack.

```bash
cdk deploy
```

AWS CloudFormation displays information about the dozens of steps that it takes as it deploys your app.

That's how easy it is to create a Fargate service to run a Docker image.

**Clean Up**

To avoid unexpected AWS charges, destroy your AWS CDK stack after you're done with this exercise.

```bash
cdk destroy
```

Creating a Code Pipeline Using the AWS CDK

This example creates a code pipeline using the AWS CDK.

The AWS CDK enables you to easily create applications running in the AWS Cloud. But creating the application is just the start of the journey. You also want to make changes to it, test those changes, and finally deploy them to your stack. The AWS CDK enables this workflow by using the **Code** suite of AWS 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.

The following example shows how to deploy an AWS Lambda function in a pipeline. In this example, your AWS CDK code and your Lambda code are in the same project. The Lambda code is in the `Lambda` directory.

To set up a project like this from scratch, follow these instructions.

**TypeScript**

```typescript
mkdir pipeline
```

129
cd pipeline
cdk init --language typescript
mkdir Lambda

Python

mkdir pipeline
cd pipeline
cdk init --language python

```bash
source .env/bin/activate || "on Windows, use: .env\Scripts\activate.bat"
pip install -r requirements.txt
mkdir Lambda
pip install aws_cdk.aws_codedeploy aws_cdk.aws_lambda aws_cdk.aws_codebuild
pip install aws_cdk.aws_codecommit aws_cdk.aws_codepipeline_actions aws_cdk.aws_s3
```

Java

```bash
mkdir pipeline
cd pipeline
cdk init --language java

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.

```java
lambda
codedeploy
codebuild
codecommit
codepipeline-actions
s3
```

C#

```bash
mkdir pipeline
cd pipeline
cdk init --language csharp

C#

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.

```bash
Amazon.CDK.AWS.CodeDeploy
Amazon.CDK.AWS.Lambda
Amazon.CDK.AWS.CodeBuild
Amazon.CDK.AWS.CodeCommit
Amazon.CDK.AWS.CodePipeline.Actions
Amazon.CDK.AWS.S3
```

Tip

If you don’t see these packages in the Browse tab of the Manage Packages for Solution page, make sure the Include prerelease checkbox is ticked.
Lambda Stack

The first step is to define the AWS CloudFormation stack that will create the Lambda function. This is the stack that we'll deploy in our pipeline.

We'll create a new file to hold this stack.

This class includes the `lambdaCode (Python: lambda_code)` property, which is an instance of the `CfnParametersCode` class. This property represents the code that is supplied later by the pipeline. Because the pipeline needs access to the object, we expose it as a public property of our class.

The example also uses the CodeDeploy support for blue-green deployments to Lambda, and the deployment increases the 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, which is named after the date when the code executed. 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, declare those resources here.

TypeScript

```typescript
File: lib/lambda-stack.ts

import codedeploy = require('@aws-cdk/aws-codedeploy');
import lambda = require('@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.cfnParameters();

        const func = new lambda.Function(this, 'Lambda', {
            code: this.lambdaCode, 
            handler: 'index.handler', 
            runtime: lambda.Runtime.NODEJS_10_X,
        });

        const version = func.addVersion(new Date().toISOString());
        const alias = new lambda.Alias(this, 'LambdaAlias', {
            aliasName: 'Prod',
            version,
        });

        new codedeploy.LambdaDeploymentGroup(this, 'DeploymentGroup', {
            alias,
            deploymentConfig: codedeploy.LambdaDeploymentConfig.LINEAR_10PERCENT_EVERY_1MINUTE,
        });
    }
}
```
Python

File: pipeline/lambda_stack.py

```python
from aws_cdk import core, aws_codedeploy as codedeploy, aws_lambda as lambda_
from datetime import datetime

class LambdaStack(core.Stack):
    def __init__(self, app: core.App, id: str, **kwargs):
        super().__init__(app, id, **kwargs)

        self.lambda_code = lambda_.Code.cfn_parameters()

        func = lambda_.Function(self, "Lambda",
            code=self.lambda_code,
            handler="index.handler",
            runtime=lambda_.Runtime.NODEJS_10_X,
        )

        version = func.add_version(datetime.now().isoformat())

        alias = lambda_.Alias(self, "LambdaAlias",
            alias_name="Prod",
            version=version,
        )

        codedeploy.LambdaDeploymentGroup(self, "DeploymentGroup",
            alias=alias,
            deployment_config=codedeploy.LambdaDeploymentConfig.LINEAR_10_PERCENT_EVERY_1_MINUTE
        )
```

Java

File: src/main/java/com/myorg/LambdaStack.java

```java
package com.myorg;

import java.time.Instant;

import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.codedeploy.LambdaDeploymentConfig;
import software.amazon.awscdk.services.codedeploy.LambdaDeploymentGroup;
import software.amazon.awscdk.services.lambda.Alias;
import software.amazon.awscdk.services.lambda.CfnParametersCode;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Version;

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);
    }
```

132
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.handler")
        .runtime(Runtime.NODEJS_10_X).build();

    Version version = func.addVersion(Instant.now().toString());
    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 System;
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(App app, string id, StackProps props=null) : base(app, id, props)
        {
            lambdaCode = Code.FromCfnParameters();

            var func = new Function(this, "Lambda", new FunctionProps
            {
                Code = lambdaCode,
                Handler = "index.handler",
                Runtime = Runtime.NODEJS_10_X
            });

            var version = func.AddVersion(DateTime.UtcNow.ToString("s"));
            var alias = new Alias(this, "LambdaAlias", new AliasProps
            {
                AliasName = "Prod",
                Version = version
            });

            new LambdaDeploymentGroup(this, "DeploymentGroup", new LambdaDeploymentGroupProps
            {
                Alias = alias,
                DeploymentConfig = LambdaDeploymentConfig.LINEAR_10PERCENT_EVERY_1MINUTE
            });
        }
    }
}
Pipeline Stack

The second class, `PipelineStack`, is the stack that contains 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 isn't necessary in Python, where properties are instead passed as keyword arguments.) This extends the standard `StackProps` and is how clients of this class (including ourselves) pass the Lambda code that the class needs.

Then comes the Git repository used to store the source code. In the example, it's hosted by CodeCommit. The `Repository.fromRepositoryName` method (Python: `from_repository_name`) is a standard AWS CDK idiom for referencing a resource, such as a CodeCommit repository, that lives outside the AWS CDK code. Replace `NameOfYourCodeCommitRepository` with the name of your repository.

The example has two CodeBuild projects. The first project obtains the AWS CloudFormation template from the AWS CDK code. To do that, it calls the standard install and build targets for Node.js, and then calls the `cdk synth` command. This produces AWS CloudFormation templates in the target directory dist. Finally, it uses the `dist/LambdaStack.template.json` file as its output.

The second project does a similar thing, except for the Lambda code. Because of that, it starts by changing the current directory to `lambda`, which is where we said the Lambda code lives in the repository. It then invokes the same install and build Node.js targets as before. The output is the contents of the node_modules directory, plus the `index.js` file. Because `index.handler` is the entry point to the Lambda code, `index.js` must exist, and must export a `handler` function. This function is called by the Lambda runtime to handle requests. If your Lambda code uses more files than just `index.js`, add them here.

Finally, we create 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, same as the build specified), and then uses the Lambda code that was passed in its props to reference the output of the build of our Lambda function. The deployed Lambda function uses the output of that build as its code. We have to make sure that the Lambda build output is an input to the AWS CloudFormation action though, and that's why we pass it in the `extraInputs` property (Python: `extra_inputs`).

We also change the name of the stack that will be deployed, from `LambdaStack` to `LambdaDeploymentStack`. The name change isn't required. We could have left it the same.

TypeScript

```typescript
import codebuild = require('@aws-cdk/aws-codebuild');
import codecommit = require('@aws-cdk/aws-codecommit');
import codepipeline = require('@aws-cdk/aws-codepipeline');
import codepipeline_actions = require('@aws-cdk/aws-codepipeline-actions');
import lambda = require('@aws-cdk/aws-lambda');
import s3 = require('@aws-cdk/aws-s3');
import { App, Stack, StackProps } from '@aws-cdk/core';

export interface PipelineStackProps extends StackProps {
  readonly lambdaCode: lambda.CfnParametersCode;
}
```
export class PipelineStack extends Stack {

    constructor(app: App, id: string, props: PipelineStackProps) {
        super(app, id, props);

        const code = codecommit.Repository.fromRepositoryName(this, 'ImportedRepo', 
            'NameOfYourCodeCommitRepository');

        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'
                        ],
                    },
                },
            }),
            environment: {
                buildImage: codebuild.LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1,
            },
        });

        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',
                    },
                },
            }),
            environment: {
                buildImage: codebuild.LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1,
            },
        });

        const sourceOutput = new codepipeline.Artifact();
        const cdkBuildOutput = new codepipeline.Artifact('CdkBuildOutput');
        const lambdaBuildOutput = new codepipeline.Artifact('LambdaBuildOutput');

        new codepipeline.Pipeline(this, 'Pipeline', { 
            stages: [
                
            ]
        });
    }
}
Pipeline Stack

```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, *,
                 lambda_code: lambda_.CfnParametersCode = None, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        code = codecommit.Repository.from_repository_name(self, "ImportedRepo",
                                                        "NameOfYourCodeCommitRepository")
```
cdk_build = codebuild.PipelineProject(self, "CdkBuild",
    build_spec=codebuild.BuildSpec.from_object(dict(
        version="0.2",
        phases=dict(
            install=dict(
                commands="npm install"),
            build=dict(commands=[
                "npm run build",
                "npm run cdk synth -- -o dist"])),
        artifacts={
            "base-directory": "dist",
            "files": [
                "LambdaStack.template.json"],
        environment=dict(buildImage=
            codebuild.LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1))))

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"],
            build=dict(
                commands="npm run build")),
        artifacts={
            "base-directory": "lambda",
            "files": [
                "index.js",
                "node_modules/**/*"],
        environment=dict(buildImage=
            codebuild.LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1))))

source_output = codepipeline.Artifact()

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",
            actions=[
                codepipeline_actions.CloudFormationCreateUpdateStackAction(  
                    action_name="Lambda_CF_Deploy"},  
                    stack_parameters=[
                        "stack_name": "LambdaStack",  
                        "parameters": ["..."],  
                        "template_url": "LambdaStack.template.json"]
        ]
    ]
)
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

```java
package com.myorg;

import java.util.Arrays;
import java.util.List;
import java.util.HashMap;
import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.codebuild.BuildEnvironment;
import software.amazon.awscdk.services.codebuild.BuildSpec;
import software.amazon.awscdk.services.codebuild.LinuxBuildImage;
import software.amazon.awscdk.services.codebuild.PipelineProject;
import software.amazon.awscdk.services.codecommit.Repository;
import software.amazon.awscdk.services.codepipeline.Artifact;
import software.amazon.awscdk.services.codepipeline.StageProps;
import software.amazon.awscdk.services.codepipeline.Pipeline;
import software.amazon.awscdk.services.codepipeline.actions.CloudFormationCreateUpdateStackAction;
import software.amazon.awscdk.services.codepipeline.actions.CodeBuildAction;
import software.amazon.awscdk.services.codepipeline.actions.CodeCommitSourceAction;
import software.amazon.awscdk.services.lambda.CfnParametersCode;

public class PipelineStack extends Stack {
    // alternate constructor for calls without props. lambdaCode is required.
    public PipelineStack(final App scope, final String id, final CfnParametersCode lambdaCode) {
        this(scope, id, null, lambdaCode);
    }

    @SuppressWarnings("serial")
    public PipelineStack(final App scope, final String id, final StackProps props,
        final CfnParametersCode lambdaCode) {
        super(scope, id, props);
        Repository code = (Repository)Repository.fromRepositoryName(this,
            "ImportedRepo",
            "NameOfYourCodeCommitRepository");
        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");
});
put("build", new HashMap<String, Object>() {
    put("commands", Arrays.asList("npm run build", "npm run cdk synth -- o dist");
});
put("artifacts", new HashMap<String, String>() {
    put("base-directory", "dist");
});
put("files", Arrays.asList("LambdaStack.template.json"));
).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");
           });
           put("build", new HashMap<String, List<String>>() {
               put("commands", Arrays.asList("npm run build");
           });
       });
       put("artifacts", new HashMap<String, Object>() {
           put("base-directory", "lambda");
           put("files", Arrays.asList("index.js", "node_modules/**/*")
       });
   })
   .environment(BuildEnvironment.builder().buildImage( 
       LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1).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.Buildercreate()
                   .actionName("Lambda_Build")
                   .build()
           ).build())
           .build())
           .build();
Pipeline Stack

C#

File: src/pipeline/PipelineStack.cs

```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 class PipelineStack : Stack
    {
        public PipelineStack(App app, string id, PipelineStackProps props=null)
        {
            var code = Repository.FromRepositoryName(this, "ImportedRepo", "NameOfYourCodeCommitRepository");

            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, string>
                        {
                            ["commands"] = "npm install"
                        },
                        ["build"] = new Dictionary<string, object>
                        {
                            ["commands"] = new List<string> {
                                "npm run build",
                                "npm run cdk synth --o dist"
                            }
                        }
                }
            });
```
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 List<string>
                {
                    "cd lambda",
                    "npm install"
                }
            },
            ["build"] = new Dictionary<string, string>
            {
                ["commands"] = "npm run build"
            }
        },
        ["artifacts"] = new Dictionary<string, object>
        {
            ["base-directory"] = "lambda",
            ["files"] = new List<string>
            {
                "index.js",
                "node_modules/**/*"
            }
        }
    }
},
    Environment = new BuildEnvironment
    {
        BuildImage = LinuxBuildImage.UBUNTU_14_04_NODEJS_10_14_1
    }
});

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 StageProps
        {
            Actions = new []
            {
                new ActionProps
                {
                    Role = new Role_()
                    {
                        "AssumeRolePolicyDocument": new Policy_()
                        {
                            "Statement": new []
                            {
                                new StatementProps
                                {
                                    "Action": "sts:AssumeRole",
                                    "Principal": new Principal_()
                                    {
                                        "Service": "codepipeline.amazonaws.com"
                                    }
                                    "Effect": "Allow"
                                }
                            }
                        }
                    }
                    OutputArtifacts = new []
                    {
                        new OutputArtifactProps
                        {
                            Name = "sourceOutput",
                            Category = "SOURCE"
                        }
                    }
                    InputArtifacts = new []
                    {
                        new InputArtifactProps
                        {
                            Name = "cdkBuildOutput",
                            Category = "SOURCE"
                        }
                    }
                    ActionCommands = new []
                    {
                        new ActionCommandProps
                        {
                            Command = "aws cloudformation create-stack --stack-name LambdaBuild --template-url http://example.com/LambdaStack.template.json --parameters ParameterKey=BucketName,ParameterValue=my-bucket --capabilities 'CAPABILITY_NAMED_IAM' --region us-east-1"
                        }
                    }
                    Artifacts = new []
                    {
                        new ArtifactProps
                        {
                            Name = "lambdaBuildOutput",
                            Category = "TARGET"
                        }
                    }
                }
            }
        }
    }
});
Main Program

Finally, we have our main AWS CDK entry point file, which contains our app.

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 required Lambda code from the LambdaStack object.
TypeScript

File: bin/pipeline.ts

```typescript
#!/usr/bin/env node

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,
});

app.synth();
```

Python

File: app.py

```python
#!/usr/bin/env python3

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)

app.synth()
```

Java

File: src/main/java/com/myorg/PipelineApp.java

```java
package com.myorg;

import software.amazon.awscdk.core.App;

public class PipelineApp {
  public static void main(final String argv[]) {
    App app = new App();
    
    LambdaStack lambdaStack = new LambdaStack(app, "LambdaStack");
    new PipelineStack(app, "PipelineStack", lambdaStack.getLambdaCode());
    
    app.synth();
  }
}
```

C#

```csharp
using Amazon.CDK;
```
namespace Pipeline
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();
            var lambdaStack = new LambdaStack(app, "LambdaStack");
            new PipelineStack(app, "PipelineDeployingLambdaStack", new PipelineStackProps
            {
                LambdaCode = lambdaStack.lambdaCode
            });
            app.Synth();
        }
    }
}

Creating the Pipeline

The final steps are building the code and deploying the pipeline.

TypeScript

```javascript
npm run build
```

Python

No build step is necessary.

Java

```bash
mvn compile
```

**Note**

Instead of issuing `mvn compile`, you can instead press Control-B in Eclipse.

C#

```bash
dotnet build src
```

**Note**

Instead of issuing `dotnet build`, you can instead press F6 in Visual Studio.

```bash
cdk deploy PipelineDeployingLambdaStack
```

The name, **PipelineDeployingLambdaStack**, is the name we used when we instantiated `PipelineStack`.

After the deployment finishes, you should have a three-stage pipeline that looks something like the following.
Try making a change, such as to your LambdaStack AWS CDK code or 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 human intervention.

### Cleaning Up

To avoid unexpected AWS charges, destroy your AWS CDK stack after you’re done with this exercise.

```bash
cdk destroy
```
AWS CDK Examples

For more examples of AWS CDK stacks and apps in your favorite supported programming language, see:

- The CDK Examples repository on GitHub
- The AWS Code Sample Catalog.
AWS CDK HowTos

This section contains short code examples that show you how to accomplish a task using the AWS CDK.

Get a Value from an Environment Variable

To get the value of an environment variable, use code like the following. This code gets the value of the environment variable MYBUCKET.

TypeScript

```typescript
// Sets bucket_name to undefined if environment variable not set
const bucket_name = process.env.MYBUCKET;

// Sets bucket_name to a default if env var doesn't exist
const bucket_name = process.env.MYBUCKET || "DefaultName";
```

Python

```python
import os

# Throws error 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.
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 cdk = require('@aws-cdk/core');
import fs = require('fs');
new cdk.CfnInclude(this, 'ExistingInfrastructure', {
  template: JSON.parse(fs.readFileSync('my-template.json').toString())
});
```

**Python**

```python
import json
import json

cdk.CfnInclude(self, 'ExistingInfrastructure',
    template=json.load(open('my-template.json')))
```

**Java**

```java
import java.util.*;
import java.io.File;
import java.util.*;
import java.io.File;
import software.amazon.awscdk.core.CfnInclude;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
template = new ObjectMapper().readTree(new File("my-template.json")));
```

**C#**

```csharp
using Newtonsoft.Json.Linq;
new CfnInclude(this, "ExistingInfrastructure", new CfnIncludeProps
```

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. 148).
Get SSM Value

```json
    {        
        Template = JObject.Parse(File.ReadAllText("my-template.json"))        
    }
```

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

TypeScript
```typescript
    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");
```

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. 65) 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. 151)).

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. 65), not the actual value. The value is resolved by AWS CloudFormation during deployment.

TypeScript
```typescript
    import ssm = require('aws-cdk/aws-ssm');
    // Get latest version or specified version of plain string attribute
```
const latestStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name'); // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(this, 'my-plain-parameter-name', 1); // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(this, 'my-secure-parameter-name', 1); // must specify version

Python

```python
import aws_cdk.aws_ssm as ssm

# Get latest version or specified version of plain string attribute
latest_string_token = ssm.StringParameter.value_for_string_parameter(self, "my-plain-parameter-name")
latest_string_token = ssm.StringParameter.value_for_string_parameter(self, "my-plain-parameter-name", 1)

# Get specified version of secure string attribute
secure_string_token = ssm.StringParameter.value_for_secure_string_parameter(self, "my-secure-parameter-name", 1)  # must specify version
```

Java

```java
import software.amazon.awscdk.services.ssm.StringParameter;

// Get latest version or specified version of plain string attribute
String latestStringToken = StringParameter.valueForStringParameter(this, "my-plain-parameter-name"); // latest version
String versionOfStringToken = StringParameter.valueForStringParameter(this, "my-plain-parameter-name", 1); // version 1

// Get specified version of secure string attribute
String secureStringToken = StringParameter.valueForSecureStringParameter(this, "my-secure-parameter-name", 1); // must specify version
```

C#

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

// Get latest version or specified version of plain string attribute
var latestStringToken = StringParameter.ValueForStringParameter(this, "my-plain-parameter-name"); // latest version
var versionOfStringToken = StringParameter.ValueForStringParameter(this, "my-plain-parameter-name", 1); // version 1

// Get specified version of secure string attribute
var secureStringToken = StringParameter.ValueForSecureStringParameter(this, "my-secure-parameter-name", 1); // must specify version
```

Reading Systems Manager Values at Synthesis Time

It is sometimes useful to "bake in" a parameter at synthesis time, so that the resulting AWS CloudFormation template always uses the same value, rather than resolving the value during deployment.

To read a value from the Systems Manager parameter store at synthesis time, use the `valueFromLookup` method (Python: `value_from_lookup`). This method returns the actual value of the parameter as a
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 sm = require("@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", {
      // 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;

public class SecretsManagerStack extends Stack {
  public SecretsManagerStack(App scope, String id) {
    this(scope, id, null);
  }

  public SecretsManagerStack(App scope, String id, StackProps props) {
    super(scope, id, props);

    Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret",
                                     SecretAttributes.builder()
                                     .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(....)
                                     .build());
  }
}
```

C#

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

public class SecretsManagerStack : Stack
```
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 affect what resources the stack contains and their configuration, and how to instantiate multiple stacks from this class. The example uses a Boolean property, named encryptBucket (Python: encrypt_bucket), to indicate whether an Amazon S3 bucket should be encrypted. If so, the stack enables encryption using a key managed by AWS Key Management Service (AWS KMS). The app creates two instances of this stack, one with encryption and one without.

Before You Begin

First, install Node.js and the AWS CDK command line tools, if you haven’t already. See Getting Started With the AWS CDK (p. 8) for details.

Next, create an AWS CDK project by entering the following commands at the command line.

TypeScript

```bash
mkdir multistack
cd multistack
cdk init --language=typescript
```

Python

```bash
mkdir multistack
cd multistack
cdk init --language=python
source ./env/bin/activate
pip install -r requirements.txt
```
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.

---

Java

```java
mkdir multistack
cd multistack
cdk init --language=java
```

You can import the resulting Maven project into your Java IDE.

C#

```csharp
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

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

Python

```python
pip install aws_cdk.core aws_cdk.aws_s3
```

Java

```java
Using the Maven integration in your IDE (for example, in Eclipse, right-click the project and choose Maven > Add Dependency), add the following packages in the group software.amazon.awscdk.

core
s3
```

C#

```csharp
nuget install Amazon.CDK
nuget install Amazon.CDK.AWS.S3
```

Or Tools > NuGet Package Manager > Manage NuGet Packages for Solution in Visual Studio

Tip

If you don’t see these packages in the Browse tab of the Manage Packages for Solution page, make sure the Include prerelease checkbox is ticked.
For a better experience, also add the Amazon.Jsii.Analyzers package to provide compile-time checks for missing required properties.

Add Optional Parameter
So open the indicated source file in your IDE or editor and add the new interface, class, or argument. The code should look like this after the changes. The lines we added are shown in boldface.

**TypeScript**

File: `lib/multistack-stack.ts`

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

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

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
  }
}
```

**Python**

File: `multistack/multistack_stack.py`

```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`

```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) {
```

155
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 cdk = require('@aws-cdk/core');
import s3 = require('@aws-cdk/aws-s3');

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}
```

C#

```csharp
using Amazon.CDK;
using Amazon.CDK.AWS.S3;
namespace Multistack
{

  public class MultiStackProps : StackProps
  {
    public bool? EncryptBucket { get; set; }
  }

  public class MultistackStack : Stack
  {
    public MultistackStack(Construct scope, string id, MultiStackProps props) : base(scope, id, props)
    {
      // The code that defines your stack goes here
    }
  }
}
```

The new property is optional. If `encryptBucket` (Python: `encrypt_bucket`) is not present, its value is undefined, or the local equivalent. The bucket will be unencrypted by default.
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
      });
    }
  }
}

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) {
```
Define the Stack Class

```java
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) {
        Bucket.Builder.create(this, "MyGroovyBucket")
            .encryption(BucketEncryption.KMS_MANAGED)
            .removalPolicy(RemovalPolicy.DESTROY).build();
    } else {
        Bucket.Builder.create(this, "MyGroovyBucket")
            .removalPolicy(RemovalPolicy.DESTROY).build();
    }
}
```

C#

```csharp
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 {
                    Encryption = BucketEncryption.KMS_MANAGED,
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            } else {
                new Bucket(this, "MyGroovyBucket", new BucketProps {
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
        }
    }
}
```
Create Two Stack Instances

Now we'll add the code to instantiate two separate stacks. As before, the lines of code shown in boldface are the ones you need to add. Delete the existing MultistackStack definition.

**TypeScript**

File: bin/multistack.ts

```typescript
#!/usr/bin/env node
import 'source-map-support/register';
import cdk = require('@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
});
```

**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

using Amazon.CDK;

namespace Multistack {
    class Program {
        static void Main(string[] args) {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
                { Env = new Environment { Region = "us-west-1", EncryptBucket = false })
            ;

            new MultistackStack(app, "MyEastCdkStack", new MultiStackProps
                { Env = new Environment { Region = "us-east-1", EncryptBucket = true })
            ;

            app.Synth();
        }
    }
}

This code uses the new encryptBucket (Python: encrypt_bucket) property on the MultistackStack class to instantiate the following:

- One stack with an encrypted Amazon S3 bucket in the us-east-1 AWS Region.
- One stack with an unencrypted Amazon S3 bucket in the us-west-1 AWS Region.

Synthesize and Deploy the Stack

Now you can deploy stacks from the app. First, build the project, if necessary.
TypeScript

    npm run build

Python

No build step is necessary.

Java

    mvn compile

    Note
    Instead of issuing mvn compile, you can instead press Control-B in Eclipse.

C#

    dotnet build src

    Note
    Instead of issuing dotnet build, you can instead press F6 in Visual Studio.

Next, synthesize a AWS CloudFormation template for MyEastCdkStack—the stack in us-east-1. This is the stack with the encrypted S3 bucket.

$ cdk synth MyEastCdkStack

The output should look similar to the following AWS CloudFormation template (there might be slight differences).

Resources:
    MyGroovyBucketFD9882AC:
        Type: AWS::S3::Bucket
        Properties:
            BucketEncryption:
                ServerSideEncryptionConfiguration:
                    - ServerSideEncryptionByDefault:
                        SSEAlgorithm: aws:kms
            UpdateReplacePolicy: Retain
            DeletionPolicy: Retain
            Metadata:
                aws:cdk:path: MyEastCdkStack/MyGroovyBucket/Resource
            CDKMetadata:
                Type: AWS::CDK::Metadata
                Properties:
                    Modules: aws-cdk=1.10.0,@aws-cdk/aws-events=1.10.0,@aws-cdk/aws-iam=1.10.0,@aws-cdk/aws-kms=1.10.0,@aws-cdk/aws-s3=1.10.0,@aws-cdk/core=1.10.0,@aws-cdk/cx-api=1.10.0,@aws-cdk/region-info=1.10.0,jsii-runtime=node.js/v10.16.2

To deploy this stack to your AWS account, issue one of the following commands. The first command uses your default AWS profile to obtain the credentials to deploy the stack. The second uses a profile you specify: for PROFILE_NAME, substitute the name of an AWS CLI profile that contains appropriate credentials for deploying to the us-east-1 AWS Region.

cdk deploy MyEastCdkStack

cdk deploy MyEastCdkStack --profile=PROFILE_NAME
Clean Up

To avoid charges for resources that you deployed, destroy the stack using the following command.

```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. The syntax is as follows, where `METRIC` is a CloudWatch metric you have created, and the alarm is raised there are more than 100 of the measured metrics in two of the last three seconds:

**TypeScript**

```typescript
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric, // see below
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

**Python**

```python
alarm = cloudwatch.Alarm(self, "Alarm",
metric=metric,  # see below
threshold=100,
evaluation_periods=3,
datapoints_to_alarm=2
)
```

**Java**

```java
import software.amazon.awscdk.services.cloudwatch.Alarm;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
  .metric(metric)  // see below
  .threshold(100)
  .evaluationPeriods(3)
  .datapointsToAlarm(2).build();
```

**C#**

```csharp
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
  Metric = metric,  // see below
  Threshold = 100,
  EvaluationPeriods = 3,
  DatapointsToAlarm = 2
});
```
The syntax for creating a metric is as follows, where the namespace value should be something like `AWS/SQS` for an Amazon SQS queue.

**TypeScript**

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

**Python**

```python
metric = cloudwatch.Metric(
  namespace="MyNamespace",
  metric_name="MyMetric",
  dimensions=dict(MyDimension="MyDimensionValue")
)
```

**Java**

```java
Metric metric = Metric.Builder.create()
  .namespace("MyNamespace")
  .metricName("MyMetric")
  .dimensions(new HashMap<String, Object>() {{
    put("MyDimension", "MyDimensionValue");
  }}).build();
```

**C#**

```csharp
var metric = new Metric(this, "Metric", new MetricProps
{
  Namespace = "MyNamespace",
  MetricName = "MyMetric",
  Dimensions = new Dictionary<string, string>
  {
    ["MyDimension"]: "MyDimensionValue"
  }
});
```

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

**TypeScript**

```typescript
const qMetric = queue.metric("ApproximateNumberOfMessagesVisible");

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

**Python**

```python
q_metric = queue.metric("ApproximateNumberOfMessagesVisible")
```
Get Context Value

```typescript
cloudwatch.Alarm(self, "Alarm",
  metric=q_metric,
  threshold=100,
  evaluation_periods=3,
  datapoints_to_alarm=2
)
```

```java
Alarm.Builder.create(this, "Alarm")
  .metric(qMetric)
  .threshold(100)
  .evaluationPeriods(3)
  .datapointsToAlarm(2).build();
```

```csharp
var qMetric = queue.Metric("ApproximateNumberOfMessagesVisible");

new Alarm(this, "Alarm", new AlarmProps {
  Metric = qMetric,
  Threshold = 100,
  EvaluationPeriods = 3,
  DatapointsToAlarm = 2
});
```

Get a Value from a Context Variable

You can specify a context variable either as part of 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 code like the following in the context of a construct (that is, when `this`, or `self` in Python, is an instance of some construct). The example gets the value of the context variable `bucket_name`.

**TypeScript**

```typescript
const bucket_name = this.node.tryGetContext('bucket_name');
```

**Python**

```python
bucket_name = self.node.try_get_context("bucket_name")
```
Get Context Value

Java

```java
String bucketName = (String)this.getNode().tryGetContext("bucket_name");
```

C#

```c#
var bucketName = this.Node.TryGetContext("bucket_name");
```

Outside the context of a construct, you can access the context variable from the app object, like this.

TypeScript

```typescript
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#

```c#
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

For more details on working with context variables, see the section called “Context” (p. 92).
AWS CDK Tools

This section contains information about AWS CDK tools.

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.

AWS CDK Toolkit (cdk)

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

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

cdk --app "npx ts-node bin/hello-cdk.ts" ls

The second way is to add the following entry to the cdk.json file (if you use the cdk init command, the command does this for you).

```json
{
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

You can also use npx cdk instead of just cdk. npx cdk looks for a locally-installed copy of the AWS CDK CLI in the current project before falling back to a global installation.

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

<table>
<thead>
<tr>
<th>Usage: cdk -a &lt;cdk-app&gt; COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commands:</td>
</tr>
<tr>
<td>cdk list [STACKS..]</td>
</tr>
<tr>
<td>cdk synthesize [STACKS..]</td>
</tr>
<tr>
<td>cdk bootstrap [ENVIRONMENTS..]</td>
</tr>
<tr>
<td>cdk deploy [STACKS..]</td>
</tr>
<tr>
<td>cdk destroy [STACKS..]</td>
</tr>
<tr>
<td>cdk diff [STACKS..]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>cdk metadata [STACK]</td>
</tr>
<tr>
<td>cdk init [TEMPLATE]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
AWS CDK Toolkit Commands

The AWS CDK CLI supports several distinct commands. Help for each (including only the command-line options specific to the particular command) follows. Commands with no command-specific options are not listed. All commands additionally accept the options listed above.

**cdk list (ls)**

```
cdk list [STACKS..]
```
Lists all stacks in the app

Options:
--long, -l Display environment information for each stack
[boolean] [default: false]

cdk synthesize (synth)

cdk synthesize [STACKS..]
Synthesizes and prints the CloudFormation template for this stack

Options:
--exclusively, -e Only synthesize requested stacks, don’t include dependencies
[boolean]

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

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
--bootstrap-kms-key-id AWS KMS master key ID used for the
SSE-KMS encryption
--tags, -t Tags to add for the stack
(KEY=VALUE) [array] [default: []]
--execute Whether to execute ChangeSet
(--no-execute will NOT execute the ChangeSet) [boolean] [default: true]

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

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. Use --no-ci to disable CI autodetection. [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]
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 [boolean]
```

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

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]
```

Bootstrapping your AWS Environment

Before you can use the AWS CDK you must bootstrap your AWS environment to create the infrastructure that the AWS CDK CLI needs to deploy your AWS CDK app. Currently the `bootstrap` command creates only an Amazon S3 bucket.

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

Security-Related Changes

To protect you against unintended changes that affect your security posture, the AWS CDK toolkit prompts you to approve security-related changes before deploying them.
You change the level of changes that requires approval by specifying:

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

Where `LEVEL` can be one of the following:

- `never`
  Approval is never required.
- `any-change`
  Requires approval on any IAM or security-group related change.
- `broadening`
  (default) Requires approval when IAM statements or traffic rules are added. Removals don't require approval.

The setting can also be configured in the `cdk.json` file.

```json
{
  "app": "...",
  "requireApproval": "never"
}
```

**Version Reporting**

To gain insight into how the AWS CDK is used, the versions of libraries used by AWS CDK applications are collected and reported by using a resource identified as `AWS::CDK::Metadata`. This resource is added to AWS CloudFormation templates, and can easily be reviewed. This information can also be used to identify stacks using a package with known serious security or reliability issues, and to contact their users with important information.

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

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

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

**Opting Out from Version Reporting**

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

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

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

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

  ```json
  {
    "app": "...",
    "versionReporting": false
  }
  ```
SAM CLI

This topic describes how to use the SAM CLI with the AWS CDK to test a Lambda function locally. For further information, see Invoking Functions Locally. To install the SAM CLI, see Installing the AWS SAM CLI.

1. The first step is to create a AWS CDK application and add the Lambda package.

```
mkdir cdk-sam-example
cd cdk-sam-example
cdk init app --language typescript
npm install @aws-cdk/aws-lambda
```

2. Add a Lambda reference to `lib/cdk-sam-example-stack.ts`:

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

3. Replace the comment in `lib/cdk-sam-example-stack.ts` with the following Lambda function:

```
new lambda.Function(this, 'MyFunction', {
  runtime: lambda.Runtime.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. Compile your AWS CDK app and create a AWS CloudFormation template

```
npm run build
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)
```
Fetching lambci/lambda:python3.7 Docker container image......
2019-04-01 12:22:43 Mounting D:\cdk-sam-example\cdk.staging
\a57f59883918e662ab3c46b964d2faa5 as /var/task:ro,delegated inside runtime container
START RequestId: 52fdfe07-2182-154f-163f-5f0f9a621d72 Version: $LATEST
END RequestId: 52fdfe07-2182-154f-163f-5f0f9a621d72
REPORT RequestId: 52fdfe07-2182-154f-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/dead-letter-queue.ts`:

```typescript
import {Construct, Duration} from '@aws-cdk/core';
import sqs = require('@aws-cdk/aws-sqs');
import cloudwatch = require('@aws-cdk/aws-cloudwatch');

export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

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

    // Create a CloudWatch alarm that fires when a message arrives.
    const alarm = new cloudwatch.Alarm(this, 'MessagesInQueueAlarm', {
      threshold: 1,
      evaluationPeriods: 1,
      metric: new sqs.QueueCount(this, 'QueueCount', {queue: this}),
    });

    this.messagesInQueueAlarm = alarm;
  }
}
```
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 assertion module.

```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`.

```javascript
import { SynthUtils } from '@aws-cdk/assert';
import { Stack } from '@aws-cdk/core';
```
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.

```
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.

```json
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 `dead-letter-queue.ts`.

```typescript
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
```
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 with default options—along with literally everything else about the synthesized template. This problem is magnified when a project contains many constructs, each with a snapshot test.

Fine-Grained Assertions

To avoid needing to review every snapshot whenever you make a change, use the custom assertions in the @aws-cdk/assert/jest module to write fine-grained tests that verify only part of the construct's behavior. For example, the test we called "dlq creates an alarm" in our example really should assert only that an alarm is created with the appropriate metric.

The AWS::CloudWatch::Alarm resource specification reveals that we're interested in the properties Namespace, MetricName and Dimensions. We'll use the expect(stack).toHaveResource(...) assertion, which is in the @aws-cdk/assert/jest module, to make sure these properties have the appropriate values.

Replace the code in test/dead-letter-queue.test.ts with the following.

```typescript
import { Stack } from '@aws-cdk/core';
import '@aws-cdk/assert/jest';
import dlq = require('../lib/dead-letter-queue');

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
```
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.

```typescript
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`.

```typescript
test('retention period can be configured', () => {
  const stack = new Stack();

  new dlq.DeadLetterQueue(stack, 'DLQ', {
    retentionDays: 7
  });

  expect(stack).toHaveResource('AWS::SQS::Queue', {
    MessageRetentionPeriod: 604800
  });
});
```
test('configurable retention period cannot exceed 14 days', () => {
  const stack = new Stack();

  expect(() => {
    new dlq.DeadLetterQueue(stack, 'DLQ', {
      retentionDays: 15
    });
  }).toThrowError(/retentionDays may not exceed 14 days/);
});

Run the tests to confirm the construct behaves as expected.

cpym 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. 180)
- Compliance Validation for the AWS Cloud Development Kit (AWS CDK) (p. 181)
- Resilience for the AWS Cloud Development Kit (AWS CDK) (p. 181)
- Infrastructure Security for the AWS Cloud Development Kit (AWS CDK) (p. 182)

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. 183)
- After updating the AWS CDK, the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library (p. 184)
- When deploying my AWS CDK stack, I receive a NoSuchBucket error (p. 185)
- When deploying my AWS CDK stack, I receive a forbidden: null message (p. 186)
- 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. 186)
- When deploying an AWS CDK stack, I receive an error because the AWS CloudFormation template contains too many resources (p. 186)
- 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. 187)
- My S3 bucket, DynamoDB table, or other resource is not deleted when I issue cdk destroy (p. 187)

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.

Below, you’ll find details on managing the versions of your installed AWS Construct Library modules TypeScript, JavaScript, and Python

TypeScript/JavaScript

Install your project’s AWS Construct Library modules locally (the default). Use npm to install the modules and keep them up to date.

To see what needs to be updated:

```
  npm outdated
```

To actually update the modules to the latest version:

```
  npm update
```
If you are working with a specific older version of the AWS Construct Library, rather than the latest, first uninstall all of your project's `@aws-cdk` modules, then reinstall the specific version you want to use. For example, to install version 1.9.0 of the Amazon S3 module, use:

```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
```

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. 76) 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-*. 

When synthesizing an AWS CDK stack, I get the message --app is required either in command-line, in cdk.json or in ~/.cdk.json

This message usually means that you aren't in the main directory of your AWS CDK project when you issue cdk synth. The file cdk.json in this directory, created by the cdk init command, contains the command line needed to run (and thereby synthesize) your AWS CDK app. For a TypeScript app, for example, the default cdk.json looks something like this:

```json
{
  "app": "npx ts-node bin/my-cdk-app.ts"
}
```

We recommend issuing cdk commands only in your project's main directory, so the AWS CDK toolkit can find cdk.json there and successfully run your app.

If this isn't practical for some reason, the AWS CDK Toolkit looks for the app's command line in two other locations:

- in cdk.json in your home directory
- on the cdk synth command itself using the -a option

For example, you might synthesize a stack from a TypeScript app as follows.

```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. 123), 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

const fs = require('fs');
const path = process.argv[2];

if (path) fs.readFile(path, 'utf8', function(err, contents) {
  console.log(err ? `${err}` : `${Object.keys(JSON.parse(contents).Resources).length} resources defined in ${path}`);
}); else console.log("Please specify the path to the stack's output .json file");
```

As your stack's resource count approaches 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` construct.

(Back to list (p. 183))

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 on how to specify a stack's account and region at synthesis time, while retaining the flexibility to deploy to any region, see the section called "Environments" (p. 43).

(Back to list (p. 183))

**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 cdk = require('@aws-cdk/core');
import s3 = require('@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,
    });
  }
}
```

**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)
```

**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.

(back to list (p. 183))
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: 2022-06-19
User ID: AWS CDK Team <aws-cdk@amazon.com>
Key fingerprint: E88B E386 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

Select the “Copy” icon to copy the following OpenPGP key:

-----BEGIN PGP PUBLIC KEY BLOCK-----
mQINBFsovE8BEADFEVFChEvPoVqoqjYu9FFPczzxy9P+2sGIT/MLI3/vBPlULQmWy 1ZGoxyBNdcdOdTaNa/ftKw3EVvWTFp4V1h+uBoKZDj/p/dTmSDFRyB6CMI0g2Zj3U6G CthyiJ20f44as0L8gLdLtQnSLRF+ZrTF3zgpUlp1W7tvkLxr78jDpW4QDp8pD29 WNn3Jg855jpyPgaJKNqALnVdunl/1XKrG42nxrr7U1uUdZPFvP2ZELJ6a6n0/zaGj jg3le+xQh45aIs6PAgaAgy7JaNfwbkGTB8HjujITAY1DwQHSi513oaCM9n4JNPC xG2eJAVYTLiiznft2Ts/a50t+Zcnpq67Sp2j6QypluomoLoq3K/Ry/yFOFZ8SL 1Tu6NX00cXBptiMvUfEtIqrgSAnq18EPtLZQOYW+ZkbcVytKdFrqj7bMwA7m7Z6GCJ jga1TbcEmsmVdOyYS1G6Z2ptWtTrvrrgA6Afz2xXI1UXLR87tT/wRtAbfQKAh85FF a3U9W4c3C1MP5IyHvW1W8Zm0fLZ1zc012nojJtTS660bxcwNNL4Q8e08FWJhunjg yxsS1BnQ1Aeoj1N4Bz1I+n9iaXVvDUN7Ks1Q5EyS4P7poyUryUq+a9C5sRA7WFx+x IEOaBQpoABBt3LoAdT06AcwCd9+r2NlXhWC4/uh3YHQU1eqPqHmFWXwaAQAQB tCPFV1MqG0LPFRLyWqyPgf3cy1j2QeIYhm9uLmNvb76JAJ8EEwEIAcKAlso vsECDy8fQFezgAHCcWk1whCAQVCAIJCg8EPfariDAEaAIQXgAACKCAFZg8E4K8c cLgXd/0XHnHe02xvz385HG8QlQ127zW9vIv9WvQKmnDQmUavw8Z7+x1R3m7nq3x7Q7Q BDbcK5g11VLKSq6G6h2V6rPysokhSh1Hn2d08Tv5KIPcxKx48+1x75m0+kSSs/+ oo1UVmOmTxx2oItxY13h09H6XKX/1IL0tF2TQ2y2sS51M4HcJ8sK1nThs561Mh 0J3e6Z2EZyBkskTu/ytRJ236BP2XKIEFzFavhmTytuXWUXEFtxOx6cfIAcY1KTha aofGfWYrOvfb15gNLCb55sQXxa23Nzdcs2ZH7468WEW15GJa6A75kxsvsPCC 2YPvd+VU0Z1J1juu6s5frkiEMHRkTlkkvqE5tHtxUcP71T0o6TBcMtAmAT4q7r7 htLt2mgUyKgR8PkmAfX6G+aqBz1/UY2qk8RCzcxcm1PlAhNmeefh1m7FQZ8eNN WL0nMh88gVcNNh5h8WM9jExoQCA0d3xX3Ne1522ZDYBvPoEqzZBB8BqUPL6nWPzA2 Ek5lco/B0BqZ9P5Bq3cyy1F6wWV/2Ld96U07qLg0YoW57jS5aDmyurktsxkJ6d 4gdPeBtgmTqCecuUPv25aEuvvXEEwaI deceived780FqDaFpXgVTX+YCLlZ3zf+DwQvqFQf 80RE?ayh78iaLzF8vux/zz/Wyvd2X108r8tDi9Qb5l1Ormqws== 0wuQ
-----END PGP PUBLIC KEY BLOCK-----
JSII OpenPGP Key

<table>
<thead>
<tr>
<th>Key ID:</th>
<th>0x1C7ACE4CB2A1B93A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>RSA</td>
</tr>
<tr>
<td>Size:</td>
<td>4096/4096</td>
</tr>
<tr>
<td>Created:</td>
<td>2018-08-06</td>
</tr>
<tr>
<td>Expires:</td>
<td>2022-08-05</td>
</tr>
<tr>
<td>User ID:</td>
<td>AWS JSII Team <a href="mailto:aws-jsii@amazon.com">aws-jsii@amazon.com</a></td>
</tr>
<tr>
<td>Key fingerprint:</td>
<td>85EF 6522 4CE2 1E8C 72DB 28EC 1C7A CE4C B2A1 B93A</td>
</tr>
</tbody>
</table>

Select the “Copy” icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
mQINBFtoSs08EAD6WweLDQB26h0P7J09iR6tVQ4PGpQK1Va5H/eP+A2Iqw79UyxZ
WkXHYzhsQ5mjYY11ycgPeVY5/LV1N8HJ7Q0yzKasybnLYpNTPYxARWE8ZM9Z3ejv1R
p1GzwvBGQfoOlxyeuetE9T52kAn4d5TS5j1no4unj4gHJnwxF2nFp1APUUCzfdfK
8vDpLoGjqjLeeGlerYNbx+7xtY/1+csFjV9K09FPLNNWQLkhhY0r6Rt9ZQG+63
tJn+AuymM37w0UIX1tgySc1OIXu8h1kPGu0NuYx9AY/63Ak2cyfj0W/PLvueQ
noNQ3j0nkOE5sTOE4CyaLWqiy1BkpxvLnm5RjMSGODCkj8c9uu0Lhr7J4EotgtZs1
pem7Y/c+/2Akg9fPf8FVTfYwRPvd11x2asIncDfLoq4998dtrN5WpfI4sG404sI
x7Al8yF/ObjJAg2rD4Jm7/g/0Vb95sQAC4A63t3ofeP14g9rS0eYodEXZ+1ammxFg1xF
Sn88RC4JFhM/KX5uAtNgudvPF/9F0D69 FRNT8CnFnmn0G0cPhN5037PaC2L/Buq2
3+K68PtkKc/CHYFC/SwIt//XIDAQgsGc7l1sFDYxrxjsk7uOuwC5j1A9X6WZ/jjX
4umRRJBAV1aNhB1+yfayYCO2ZFX06capbOLv8IV7Pc4leC2DogyLy3Kl9QARQaB
7CNV1MgS4NJSBBZWPT1dxh3M4amNPaUBhhWFB6b24uY2t9P0kCPwQTAGaKQUC
W2hKXQ1hBnUw61+0Gxh+AE6dJgGk6Ahw5Aw0D3rV5jXv7dLw5G/Mw9kYgA0YuMw
v8f6V27Xl/5d0ZG5f0/pPn5xStDzAx/3k9yUyXOZxWQP30/wBhG8SoEvzoRbUUr
ok6B34P/lNbSjKyhT0gl1qiwK7tudBDRpR6f/c/sp67d/GHqN04BzlDbUPSJW5
905V7/waqhHXI/ma/QV781Rztf3W7y7m8eOodjiu77y/JprhV9n0cX/nLAdcNoH
BvabKdRkWNNWISrg7w2bcfM1TjwxyjHPiqs6ktoozjp2985CDS/KTszjw6g/fGMm
DlDhsSBdM347Egew7912aQzL7cvm/N59k+AGyEMHZDZXkew/8Ge5Ov5050Onsp
6hi8H4BPRj7uWuq1lk pajvkmWxxk02wV1jDfb1gybLbyjfs5sM5spxT2GTwA/ex/r2tAU6
lsQg0QmLMs45btDavSojymxMAAG1iTG8ECZ/cOuW9k2PChqGvYu63lxtkPP6
MyCQRMXh9aXAd4+1elteXM9F19enLyGz3BHmxioXZv02s0zlu1PuhPoApekwJ
/umv84bF6a6gPoACHQ7zr/+lk28m9Y70G0n2Z76f/J7k0e8g/mdiXTQmSct/pG
pGm1T8x31IXqQyIhlwLaM0tHNGby3M6Vx17zalnuB8Pj+07bhCwV0UVPtNh2dJ
ZsaLa98UMtqQ8UHmX3t06jdwA7mdbr2at1tttLH5WNPoCwbhJXLe/03ENCOp
nj270sBsbKx2nVvCNOH3v5ey0Y1tLY936oR7xOwCbwBp5skTCR0Ub
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
```
Document History for the AWS CDK Developer Guide

This document reflects the following release of the AWS Cloud Development Kit (AWS CDK).

- API version: 1.18.0
- Latest documentation update: November 25, 2019

See Releases for a list of the AWS CDK releases.

**Note**
The table below represents significant 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>Version 1.18.0 (p. 191)</td>
<td>Add Java code snippets throughout. Designate Java and C# bindings stable.</td>
<td>November 25, 2019</td>
</tr>
<tr>
<td>Version 1.17.0 (p. 191)</td>
<td>Add C# code snippets throughout.</td>
<td>November 19, 2019</td>
</tr>
<tr>
<td>Version 1.16.0 (p. 191)</td>
<td>Add Python code snippets throughout. Add Troubleshooting and Testing topics.</td>
<td>November 14, 2019</td>
</tr>
<tr>
<td>Version 1.8.0 (p. 191)</td>
<td>Updates to reflect improvements to ECS Patterns module.</td>
<td>September 17, 2019</td>
</tr>
<tr>
<td>Version 1.3.0 (p. 191)</td>
<td>Update tagging topic to use new API.</td>
<td>August 13, 2019</td>
</tr>
<tr>
<td>Version 1.0.0 (p. 191)</td>
<td>The AWS CDK Developer Guide is released.</td>
<td>July 11, 2019</td>
</tr>
</tbody>
</table>