# Development Guide

Best practices, testing, and development workflows for AWS CDK applications.

## 🛠️ Development Essentials

### Core Development

* [Development Chapter](chapter-develop.md) - Development workflow overview
* [Projects](projects.md) - Organizing CDK projects
* [Versioning](versioning.md) - Managing CDK and dependency versions

### Assets and Resources

* [Assets](assets.md) - Working with files, Docker images, and assets
* [Build Containers](build-containers.md) - Custom build environments
* [Tagging](tagging.md) - Resource tagging strategies

### Testing and Quality

* [Testing](testing.md) - Unit testing, integration testing
* [Testing Locally](testing-locally.md) - Local development and testing
* [Testing with SAM CLI](testing-locally-with-sam-cli.md) - Using SAM for local testing
* [Testing Build with SAM](testing-locally-build-with-sam-cli.md) - Build testing with SAM

### Best Practices

* [Best Practices](best-practices.md) - General CDK best practices
* [Security Best Practices](best-practices-security.md) - Security guidelines and practices

### Getting Started with Local Testing

* [Testing Locally - Getting Started](testing-locally-getting-started.md) - Quick start for local testing

## 🎯 Recommended Learning Path

### For New Developers

1. [**Development Chapter**](chapter-develop.md) - Overview of development process
2. [**Projects**](projects.md) - Project organization
3. [**Best Practices**](best-practices.md) - Follow established patterns

### For Testing

1. [**Testing Overview**](testing.md) - Comprehensive testing guide
2. [**Testing Locally**](testing-locally.md) - Local development setup
3. [**SAM CLI Integration**](testing-locally-with-sam-cli.md) - Advanced local testing

### For Production

1. [**Security Best Practices**](best-practices-security.md) - Security guidelines
2. [**Tagging**](tagging.md) - Resource organization
3. [**Versioning**](versioning.md) - Dependency management

## 🔗 Related Sections

* [Core Concepts](../02-core-concepts/) - Understand fundamentals first
* [Deployment](../04-deployment/) - Deploy your applications
* [Advanced](../05-advanced/) - Advanced patterns and techniques

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Assets :keywords: {aws} CDK, Assets, Amazon S3, Amazon ECR

[#assets] = Assets and the {aws} CDK

== [abstract]

## Assets are local files, directories, or Docker images.

// Content start

Assets are local files, directories, or Docker images that can be bundled into {aws} CDK libraries and apps. For example, an asset might be a directory that contains the handler code for an {aws} Lambda function. Assets can represent any artifact that the app needs to operate.

The following tutorial video provides a comprehensive overview of CDK assets, and explains how you can use them in your infrastructure as code (IaC).

video::jHNtXQmkKfw[youtube,align = center,height = 390,fileref = https://www.youtube.com/embed/jHNtXQmkKfw,width = 480]

You add assets through APIs that are exposed by specific {aws} constructs. For example, when you define a https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_lambda.Function.html[lambda.Function] construct, the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_lambda.Function.html#code[code] property lets you pass an https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_lambda.Code.html#static-fromwbrassetpath-options[asset] (directory). Function uses assets to bundle the contents of the directory and use it for the function’s code. Similarly, https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecs.ContainerImage.html#static-fromwbrassetdirectory-props[ecs.ContainerImage.fromAsset] uses a Docker image built from a local directory when defining an Amazon ECS task definition.

[#assets-details] == Assets in detail

When you refer to an asset in your app, the xref:deploy-how-synth-assemblies[cloud assembly] that’s synthesized from your application includes metadata information with instructions for the {aws} CDK CLI. The instructions include where to find the asset on the local disk and what type of bundling to perform based on the asset type, such as a directory to compress (zip) or a Docker image to build.

The {aws} CDK generates a source hash for assets. This 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 way, the cloud assembly is self-contained, so if it moved over to a different host for deployment, it can still be deployed. See xref:deploy-how-synth-assemblies[Cloud assemblies] for details.

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 the assets to an Amazon S3 bucket or Amazon ECR repository. (The S3 bucket or repository is created during bootstrapping.) Only then are the resources defined in the stack deployed.

This section describes the low-level APIs available in the framework.

[#assets-types] == 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.

[#assets-types-s3] == 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 https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3\_assets-readme.html[aws-s3-assets] module.

The following example defines a local directory asset and a file asset.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Asset } from ‘aws-cdk-lib/aws-s3-assets’;

// Archived and uploaded to Amazon S3 as a .zip file const directoryAsset = new Asset(this, “SampleZippedDirAsset”, { path: path.join(\_\_dirname, “sample-asset-directory”) });

// Uploaded to Amazon S3 as-is const fileAsset = new Asset(this, ‘SampleSingleFileAsset’, { path: path.join(\_\_dirname, ‘file-asset.txt’) }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Asset } = require(‘aws-cdk-lib/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:: + [source,python,subs=“verbatim,attributes”] — 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:: + [source,java,subs=“verbatim,attributes”] — 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#:: + [source,csharp,subs=“verbatim,attributes”] — 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”) }); —

Go:: + [source,go,subs=“verbatim,attributes”] — dirName, err := os.Getwd() if err != nil { panic(err) }

awss3assets.NewAsset(stack, jsii.String(“SampleZippedDirAsset”), &awss3assets.AssetProps{ Path: jsii.String(path.Join(dirName, “sample-asset-directory”)), })

awss3assets.NewAsset(stack, jsii.String(“SampleSingleFileAsset”), &awss3assets.AssetProps{ Path: jsii.String(path.Join(dirName, “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 so that you can use assets. For Lambda functions, the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_lambda.Code.html#static-fromwbrassetpath-options[fromAsset()] static method enables you to specify a directory or a .zip file in the local file system.

[#assets-types-s3-lambda] === Lambda function example

A common use case is creating Lambda functions with the handler code as an Amazon S3 asset.

The following example uses an Amazon S3 asset to define a Python handler in the local directory handler. It also creates a Lambda function with the local directory asset as the code property. Following is the Python code for the handler.

== [source,python,subs=“verbatim,attributes”]

def lambda\_handler(event, context): message = ‘Hello World!’ return { ‘message’: message } —

The code for the main {aws} CDK app should look like the following.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Constructs } from ‘constructs’; import \* as lambda from ‘aws-cdk-lib/aws-lambda’; import \* as path from ‘path’;

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

new lambda.Function(this, ‘myLambdaFunction’, { code: lambda.Code.fromAsset(path.join(\_\_dirname, ‘handler’)), runtime: lambda.Runtime.PYTHON\_3\_6, handler: ‘index.lambda\_handler’ }); } } —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const lambda = require(‘aws-cdk-lib/aws-lambda’); const path = require(‘path’);

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

new lambda.Function(this, ‘myLambdaFunction’, { code: lambda.Code.fromAsset(path.join(\_\_dirname, ‘handler’)), runtime: lambda.Runtime.PYTHON\_3\_6, handler: ‘index.lambda\_handler’ }); } }

== module.exports = { HelloAssetStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import Stack from constructs import 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:: + [source,java,subs=“verbatim,attributes”] — import java.io.File;

import software.amazon.awscdk.Stack; import software.amazon.awscdk.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#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.Lambda; using System.IO;

public class HelloAssetStack : Stack { public HelloAssetStack(Construct scope, string id, StackProps props) : base(scope, id, props) { new Function(this, “myLambdaFunction”, new FunctionProps { Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), “handler”)), Runtime = Runtime.PYTHON\_3\_6, Handler = “index.lambda\_handler” }); } } —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awslambda” “github.com/aws/aws-cdk-go/awscdk/v2/awss3assets” “github.com/aws/constructs-go/constructs/v10” “github.com/aws/jsii-runtime-go” )

func HelloAssetStack(scope constructs.Construct, id string, props \*HelloAssetStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } stack := awscdk.NewStack(scope, &id, &sprops)

dirName, err := os.Getwd() if err != nil { panic(err) }

awslambda.NewFunction(stack, jsii.String(“myLambdaFunction”), &awslambda.FunctionProps{ Code: awslambda.AssetCode\_FromAsset(jsii.String(path.Join(dirName, “handler”)), &awss3assets.AssetOptions{}), Runtime: awslambda.Runtime\_PYTHON\_3\_6(), Handler: jsii.String(“index.lambda\_handler”), })

return stack } — ====

The Function method uses assets to bundle the contents of the directory and use it for the function’s code.

= [TIP]

Java .jar files are ZIP files with a different extension. These are uploaded as-is to Amazon S3, but when deployed as a Lambda function, the files they contain are extracted, which you might not want. To avoid this, place the .jar file in a directory and specify that directory as the asset.

====

[#assets-types-s3-deploy] === Deploy-time attributes example

Amazon S3 asset types also expose xref:resources-attributes[deploy-time attributes] that can be referenced in {aws} CDK libraries and apps. The {aws} CDK CLI command cdk synth displays asset properties as {aws} CloudFormation parameters.

The following example uses deploy-time attributes to pass the location of an image asset into a Lambda function as environment variables. (The kind of file doesn’t matter; the PNG image used here is only an example.)

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Asset } from ‘aws-cdk-lib/aws-s3-assets’; import \* as path from ‘path’;

const imageAsset = new Asset(this, “SampleAsset”, { path: path.join(\_\_dirname, “images/my-image.png”) });

new lambda.Function(this, “myLambdaFunction”, { code: lambda.Code.asset(path.join(\_\_dirname, “handler”)), runtime: lambda.Runtime.PYTHON\_3\_6, handler: “index.lambda\_handler”, environment: { ‘S3\_BUCKET\_NAME’: imageAsset.s3BucketName, ‘S3\_OBJECT\_KEY’: imageAsset.s3ObjectKey, ‘S3\_OBJECT\_URL’: imageAsset.s3ObjectUrl } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Asset } = require(‘aws-cdk-lib/aws-s3-assets’); const path = require(‘path’);

const imageAsset = new Asset(this, “SampleAsset”, { path: path.join(\_\_dirname, “images/my-image.png”) });

new lambda.Function(this, “myLambdaFunction”, { code: lambda.Code.asset(path.join(\_\_dirname, “handler”)), runtime: lambda.Runtime.PYTHON\_3\_6, handler: “index.lambda\_handler”, environment: { ‘S3\_BUCKET\_NAME’: imageAsset.s3BucketName, ‘S3\_OBJECT\_KEY’: imageAsset.s3ObjectKey, ‘S3\_OBJECT\_URL’: imageAsset.s3ObjectUrl } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — import os.path

import aws\_cdk.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\_OBJECT\_URL=image\_asset.s3\_object\_url)) —

Java:: + [source,java,subs=“verbatim,attributes”] — import java.io.File;

import software.amazon.awscdk.Stack; import software.amazon.awscdk.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(java.util.Map.of( // Java 9 or later  
 "S3\_BUCKET\_NAME", imageAsset.getS3BucketName(),  
 "S3\_OBJECT\_KEY", imageAsset.getS3ObjectKey(),  
 "S3\_OBJECT\_URL", imageAsset.getS3ObjectUrl()))  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — 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-image.png”) });

new Function(this, “myLambdaFunction”, new FunctionProps { Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), “handler”)), Runtime = Runtime.PYTHON\_3\_6, Handler = “index.lambda\_handler”, Environment = new Dictionary<string, string> { [“S3\_BUCKET\_NAME”] = imageAsset.S3BucketName, [“S3\_OBJECT\_KEY”] = imageAsset.S3ObjectKey, [“S3\_OBJECT\_URL”] = imageAsset.S3ObjectUrl } }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awslambda” “github.com/aws/aws-cdk-go/awscdk/v2/awss3assets” )

dirName, err := os.Getwd() if err != nil { panic(err) }

imageAsset := awss3assets.NewAsset(stack, jsii.String(“SampleAsset”), &awss3assets.AssetProps{ Path: jsii.String(path.Join(dirName, “images/my-image.png”)), })

awslambda.NewFunction(stack, jsii.String(“myLambdaFunction”), &awslambda.FunctionProps{ Code: awslambda.AssetCode\_FromAsset(jsii.String(path.Join(dirName, “handler”))), Runtime: awslambda.Runtime\_PYTHON\_3\_6(), Handler: jsii.String(“index.lambda\_handler”), Environment: &map[string]\*string{ “S3\_BUCKET\_NAME”: imageAsset.S3BucketName(), “S3\_OBJECT\_KEY”: imageAsset.S3ObjectKey(), “S3\_URL”: imageAsset.S3ObjectUrl(), }, }) — ====

[#assets-types-s3-permissions] === Permissions

If you use Amazon S3 assets directly through the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3\_assets-readme.html[aws-s3-assets] module, IAM roles, users, or groups, and you need to read assets in runtime, then grant those assets IAM permissions through the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3\_assets.Asset.html#grantwbrreadgrantee[asset.grantRead] method.

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Asset } from ‘aws-cdk-lib/aws-s3-assets’; import \* as path from ‘path’;

const asset = new Asset(this, ‘MyFile’, { path: path.join(\_\_dirname, ‘my-image.png’) });

const group = new iam.Group(this, ‘MyUserGroup’); asset.grantRead(group); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Asset } = require(‘aws-cdk-lib/aws-s3-assets’); const path = require(‘path’);

const asset = new Asset(this, ‘MyFile’, { path: path.join(\_\_dirname, ‘my-image.png’) });

const group = new iam.Group(this, ‘MyUserGroup’); asset.grantRead(group); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk.aws\_s3\_assets import Asset import aws\_cdk.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.grant\_read(group) ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — import java.io.File;

import software.amazon.awscdk.Stack; import software.amazon.awscdk.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#:: + [source,csharp,subs=“verbatim,attributes”] — 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-image.png”)) });

var group = new Group(this, “MyUserGroup”); asset.GrantRead(group); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awsiam” “github.com/aws/aws-cdk-go/awscdk/v2/awss3assets” )

dirName, err := os.Getwd() if err != nil { panic(err) }

asset := awss3assets.NewAsset(stack, jsii.String(“MyFile”), &awss3assets.AssetProps{ Path: jsii.String(path.Join(dirName, “my-image.png”)), })

group := awsiam.NewGroup(stack, jsii.String(“MyUserGroup”), &awsiam.GroupProps{})

== asset.GrantRead(group)

====

[#assets-types-docker] == Docker image assets

The {aws} CDK supports bundling local Docker images as assets through the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecr\_assets-readme.html[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. The images can be naturally referenced in your {aws} CDK app.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { DockerImageAsset } from ‘aws-cdk-lib/aws-ecr-assets’;

const asset = new DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’) }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { DockerImageAsset } = require(‘aws-cdk-lib/aws-ecr-assets’);

const asset = new DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’) }); —

Python:: + [source,python,subs=“verbatim,attributes”] — 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:: + [source,java,subs=“verbatim,attributes”] — 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#:: + [source,csharp,subs=“verbatim,attributes”] — using System.IO; using Amazon.CDK.{aws}.ECR.Assets;

var asset = new DockerImageAsset(this, “MyBuildImage”, new DockerImageAssetProps { Directory = Path.Combine(Directory.GetCurrentDirectory(), “my-image”) }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets” )

dirName, err := os.Getwd() if err != nil { panic(err) }

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String(“MyBuildImage”), &awsecrassets.DockerImageAssetProps{ Directory: jsii.String(path.Join(dirName, “my-image”)), }) — ====

The my-image directory must include a Dockerfile. The {aws} CDK CLI builds a Docker image from my-image, pushes it to an Amazon ECR repository, and specifies the name of the repository as an {aws} CloudFormation parameter to your stack. Docker image asset types expose xref:resources-attributes[deploy-time attributes] that can be referenced in {aws} CDK libraries and apps. The {aws} CDK CLI command cdk synth displays asset properties as {aws} CloudFormation parameters.

[#assets-types-docker-ecs] === Amazon ECS task definition example

A common use case is to create an Amazon ECS https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecs.TaskDefinition.html[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.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as ecs from ‘aws-cdk-lib/aws-ecs’; import \* as ecr\_assets from ‘aws-cdk-lib/aws-ecr-assets’; import \* as path from ‘path’;

const taskDefinition = new ecs.FargateTaskDefinition(this, “TaskDef”, { memoryLimitMiB: 1024, cpu: 512 });

const asset = new ecr\_assets.DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’) });

taskDefinition.addContainer(“my-other-container”, { image: ecs.ContainerImage.fromDockerImageAsset(asset) }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const ecs = require(‘aws-cdk-lib/aws-ecs’); const ecr\_assets = require(‘aws-cdk-lib/aws-ecr-assets’); const path = require(‘path’);

const taskDefinition = new ecs.FargateTaskDefinition(this, “TaskDef”, { memoryLimitMiB: 1024, cpu: 512 });

const asset = new ecr\_assets.DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’) });

taskDefinition.addContainer(“my-other-container”, { image: ecs.ContainerImage.fromDockerImageAsset(asset) }); —

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_ecs as ecs import aws\_cdk.aws\_ecr\_assets as ecr\_assets

import os.path dirname = os.path.dirname(*file*)

task\_definition = ecs.FargateTaskDefinition(self, “TaskDef”, memory\_limit\_mib=1024, cpu=512)

asset = ecr\_assets.DockerImageAsset(self, ‘MyBuildImage’, directory=os.path.join(dirname, ‘my-image’))

task\_definition.add\_container(“my-other-container”, image=ecs.ContainerImage.from\_docker\_image\_asset(asset)) —

Java:: + [source,java,subs=“verbatim,attributes”] — 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;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

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

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create( this, “TaskDef”).memoryLimitMiB(1024).cpu(512).build();

DockerImageAsset asset = DockerImageAsset.Builder.create(this, “MyBuildImage”) .directory(new File(startDir, “my-image”).toString()).build();

taskDefinition.addContainer(“my-other-container”, ContainerDefinitionOptions.builder() .image(ContainerImage.fromDockerImageAsset(asset)) .build(); ) —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using System.IO; using Amazon.CDK.{aws}.ECS; using Amazon.CDK.{aws}.Ecr.Assets;

var taskDefinition = new FargateTaskDefinition(this, “TaskDef”, new FargateTaskDefinitionProps { MemoryLimitMiB = 1024, Cpu = 512 });

var asset = new DockerImageAsset(this, “MyBuildImage”, new DockerImageAssetProps { Directory = Path.Combine(Directory.GetCurrentDirectory(), “my-image”) });

taskDefinition.AddContainer(“my-other-container”, new ContainerDefinitionOptions { Image = ContainerImage.FromDockerImageAsset(asset) }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets” “github.com/aws/aws-cdk-go/awscdk/v2/awsecs” )

dirName, err := os.Getwd() if err != nil { panic(err) }

taskDefinition := awsecs.NewTaskDefinition(stack, jsii.String(“TaskDef”), &awsecs.TaskDefinitionProps{ MemoryMiB: jsii.String(“1024”), Cpu: jsii.String(“512”), })

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String(“MyBuildImage”), &awsecrassets.DockerImageAssetProps{ Directory: jsii.String(path.Join(dirName, “my-image”)), })

taskDefinition.AddContainer(jsii.String(“MyOtherContainer”), &awsecs.ContainerDefinitionOptions{ Image: awsecs.ContainerImage\_FromDockerImageAsset(asset), }) — ====

[#assets-types-docker-deploy] === Deploy-time attributes example

The following example shows how to use the deploy-time attributes repository and imageUri to create an Amazon ECS task definition with the {aws} Fargate launch type. Note that the Amazon ECR repo lookup requires the image’s tag, not its URI, so we snip it from the end of the asset’s URI.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as ecs from ‘aws-cdk-lib/aws-ecs’; import \* as path from ‘path’; import { DockerImageAsset } from ‘aws-cdk-lib/aws-ecr-assets’;

const asset = new DockerImageAsset(this, ‘my-image’, { directory: path.join(\_\_dirname, “..”, “demo-image”) });

const taskDefinition = new ecs.FargateTaskDefinition(this, “TaskDef”, { memoryLimitMiB: 1024, cpu: 512 });

taskDefinition.addContainer(“my-other-container”, { image: ecs.ContainerImage.fromEcrRepository(asset.repository, asset.imageUri.split(“:”).pop()) }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const ecs = require(‘aws-cdk-lib/aws-ecs’); const path = require(‘path’); const { DockerImageAsset } = require(‘aws-cdk-lib/aws-ecr-assets’);

const asset = new DockerImageAsset(this, ‘my-image’, { directory: path.join(\_\_dirname, “..”, “demo-image”) });

const taskDefinition = new ecs.FargateTaskDefinition(this, “TaskDef”, { memoryLimitMiB: 1024, cpu: 512 });

taskDefinition.addContainer(“my-other-container”, { image: ecs.ContainerImage.fromEcrRepository(asset.repository, asset.imageUri.split(“:”).pop()) }); —

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_ecs as ecs from aws\_cdk.aws\_ecr\_assets import DockerImageAsset

import os.path dirname = os.path.dirname(*file*)

asset = DockerImageAsset(self, ‘my-image’, directory=os.path.join(dirname, “..”, “demo-image”))

task\_definition = ecs.FargateTaskDefinition(self, “TaskDef”, memory\_limit\_mib=1024, cpu=512)

task\_definition.add\_container(“my-other-container”, image=ecs.ContainerImage.from\_ecr\_repository( asset.repository, asset.image\_uri.rpartition(“:”)[-1])) —

Java:: + [source,java,subs=“verbatim,attributes”] — import java.io.File;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition; import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions; import software.amazon.awscdk.services.ecs.ContainerImage;

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

DockerImageAsset asset = DockerImageAsset.Builder.create(this, “my-image”) .directory(new File(startDir, “demo-image”).toString()).build();

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create( this, “TaskDef”).memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset’s image URI for use in ECR repo lookup String imageUri = asset.getImageUri(); String imageTag = imageUri.substring(imageUri.lastIndexOf(“:”) + 1);

taskDefinition.addContainer(“my-other-container”, ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository( asset.getRepository(), imageTag)).build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using System.IO; using Amazon.CDK.{aws}.ECS; using Amazon.CDK.{aws}.ECR.Assets;

var asset = new DockerImageAsset(this, “my-image”, new DockerImageAssetProps { Directory = Path.Combine(Directory.GetCurrentDirectory(), “demo-image”) });

var taskDefinition = new FargateTaskDefinition(this, “TaskDef”, new FargateTaskDefinitionProps { MemoryLimitMiB = 1024, Cpu = 512 });

taskDefinition.AddContainer(“my-other-container”, new ContainerDefinitionOptions { Image = ContainerImage.FromEcrRepository(asset.Repository, asset.ImageUri.Split(“:”).Last()) }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “os” “path”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets” “github.com/aws/aws-cdk-go/awscdk/v2/awsecs” )

dirName, err := os.Getwd() if err != nil { panic(err) }

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String(“MyImage”), &awsecrassets.DockerImageAssetProps{ Directory: jsii.String(path.Join(dirName, “demo-image”)), })

taskDefinition := awsecs.NewFargateTaskDefinition(stack, jsii.String(“TaskDef”), &awsecs.FargateTaskDefinitionProps{ MemoryLimitMiB: jsii.Number(1024), Cpu: jsii.Number(512), })

taskDefinition.AddContainer(jsii.String(“MyOtherContainer”), &awsecs.ContainerDefinitionOptions{ Image: awsecs.ContainerImage\_FromEcrRepository(asset.Repository(), asset.ImageTag()), }) — ====

[#assets-types-docker-build] === 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.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const asset = new DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’), buildArgs: { HTTP\_PROXY: ‘http://10.20.30.2:1234’ } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const asset = new DockerImageAsset(this, ‘MyBuildImage’, { directory: path.join(\_\_dirname, ‘my-image’), buildArgs: { HTTP\_PROXY: ‘http://10.20.30.2:1234’ } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — asset = DockerImageAsset(self, “MyBuildImage”, directory=os.path.join(dirname, “my-image”), build\_args=dict(HTTP\_PROXY=“http://10.20.30.2:1234”)) —

Java:: + [source,java,subs=“verbatim,attributes”] — DockerImageAsset asset = DockerImageAsset.Builder.create(this, “my-image”), .directory(new File(startDir, “my-image”).toString()) .buildArgs(java.util.Map.of( // Java 9 or later “HTTP\_PROXY”, “http://10.20.30.2:1234”)) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var asset = new DockerImageAsset(this, “MyBuildImage”, new DockerImageAssetProps { Directory = Path.Combine(Directory.GetCurrentDirectory(), “my-image”), BuildArgs = new Dictionary<string, string> { [“HTTP\_PROXY”] = “http://10.20.30.2:1234” } }); —

Go:: + [source,go,subs=“verbatim,attributes”] — dirName, err := os.Getwd() if err != nil { panic(err) }

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String(“MyBuildImage”), &awsecrassets.DockerImageAssetProps{ Directory: jsii.String(path.Join(dirName, “my-image”)), BuildArgs: &map[string]\*string{ “HTTP\_PROXY”: jsii.String(“http://10.20.30.2:1234”), }, }) — ====

[#assets-types-docker-permissions] === Permissions

If you use a module that supports Docker image assets, such as https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecs-readme.html[aws-ecs], the {aws} CDK manages permissions for you when you use assets directly or through https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecs.ContainerImage.html#static-fromwbrecrwbrrepositoryrepository-tag[ContainerImage.fromEcrRepository] (Python: from\_ecr\_repository). If you use Docker image assets directly, make sure that the consuming principal has permissions to pull the image.

In most cases, you should use https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecr.Repository.html#grantwbrpullgrantee[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 if it’s an {aws} service that doesn’t assume a role in your account (such as {aws} CodeBuild), you must grant pull permissions on the resource policy and not on the principal’s policy. Use the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecr.Repository.html#addwbrtowbrresourcewbrpolicystatement[asset.repository.addToResourcePolicy] method (Python: add\_to\_resource\_policy) to grant the appropriate principal permissions.

[#assets-cfn] == {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 xref:sam[{aws} SAM integration] 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.

include::attributes.txt[]

// Attributes

[.topic] [[best-practices-security,best-practices-security.title]] = {aws} CDK security best practices :info\_titleabbrev: Security :keywords: {aws} CDK, IAM, security, permissions, infrastructure, {aws} CloudFormation, {aws} CDK deployments

== [abstract]

## The {aws} Cloud Development Kit ({aws} CDK) is a powerful tool that developers can use to configure {aws} services and provision infrastructure on {aws}. With any tool that provides such control and capabilities, organizations will need to establish policies and practices to ensure that the tool is being used in safe and secure ways. For example, organizations may want to restrict developer access to specific services to ensure that they can’t tamper with compliance or cost control measures that are configured in the account.

// Content start

The {aws} Cloud Development Kit ({aws} CDK) is a powerful tool that developers can use to configure {aws} services and provision infrastructure on {aws}. With any tool that provides such control and capabilities, organizations will need to establish policies and practices to ensure that the tool is being used in safe and secure ways. For example, organizations may want to restrict developer access to specific services to ensure that they can’t tamper with compliance or cost control measures that are configured in the account.

Often, there can be a tension between security and productivity, and each organization needs to establish the proper balance for themselves. This topic provides security best practices for the {aws} CDK that you can consider as you create and implement your own security policies. The following best practices are general guidelines and don`’t represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

[#best-practices-security-iam] == Follow IAM security best practices

{aws} Identity and Access Management (IAM) is a web service that helps you securely control access to {aws} resources. Organizations, individuals, and the {aws} CDK use IAM to manage permissions that determine the actions that can be performed on {aws} resources. When using IAM, follow the IAM security best practices. For more information, see https://docs.aws.amazon.com/IAM/latest/UserGuide/IAMBestPracticesAndUseCases.html[Security best practices and use cases in {aws} Identity and Access Management] in the *IAM User Guide*.

[#best-practices-security-permissions] == Manage permissions for the {aws} CDK

When you use the {aws} CDK across your organization to develop and manage your infrastructure, you’ll want to consider the following scenarios where managing permissions will be important:

* *Permissions for {aws} CDK deployments* – These permissions determine who can make changes to your {aws} resources and what changes they can make.
* *Permissions between resources* – These are the permissions that allow interactions between the {aws} resources that you create and manage with the {aws} CDK.

[#best-practices-security-permissions-deployments] === Manage permissions for {aws} CDK deployments

Developers use the {aws} CDK to define infrastructure locally on their development machines. This infrastructure is implemented in {aws} environments through deployments that typically involve using the {aws} CDK Command Line Interface ({aws} CDK CLI). With deployments, you may want to control what changes developers can make in your environments. For example, you might have an Amazon Virtual Private Cloud (Amazon VPC) resource that you don’t want developers to modify.

By default, the CDK CLI uses a combination of the actor’s security credentials and IAM roles that are created during bootstrapping to receive permissions for deployments. The actor’s security credentials are first used for authentication and IAM roles are then assumed to perform various actions during deployment, such as using the {aws} CloudFormation service to create resources. For more information on how CDK deployments work, including the IAM roles that are used, see xref:deploy[Deploy {aws} CDK applications].

To restrict who can perform deployments and the actions that can be performed during deployment, consider the following:

* The actor’s security credentials are the first set of credentials used to authenticate to {aws}. From here, the permissions used to perform actions during deployment are granted to the IAM roles that are assumed during the deployment workflow. You can restrict who can perform deployments by limiting who can assume these roles. You can also restrict the actions that can be performed during deployment by replacing these IAM roles with your own.
* Permissions for performing deployments are given to the DeploymentActionRole. You can control permissions for who can perform deployments by limiting who can assume this role. By using a role for deployments, you can perform cross-account deployments since the role can be assumed by {aws} identities in a different account. By default, all identities in the same {aws} account with the appropriate AssumeRole policy statement can assume this role.
* Permissions for creating and modifying resources through {aws} CloudFormation are given to the CloudFormationExecutionRole. This role also requires permission to read from the bootstrap resources. You control the permissions that CDK deployments have by using a managed policy for the CloudFormationExecutionRole and optionally by configuring a permissions boundary. By default, this role has AdministratorAccess permissions with no permission boundary.
* Permissions for interacting with bootstrap resources are given to the FilePublishingRole and ImagePublishingRole. The actor performing deployments must have permission to assume these roles. By default, all identities in the same {aws} account with the appropriate AssumeRole policy statement can assume this role.
* Permissions for accessing bootstrap resources to perform lookups are given to the LookupRole. The actor performing deployments must have permission to assume this role. By default, this role has readOnly access to the bootstrap resources. By default, all identities in the same {aws} account with the appropriate AssumeRole policy statement can assume this role.

To configure the IAM identities in your {aws} account with permission to assume these roles, add a policy with the following policy statement to the identities:

== [source,json,subs=“verbatim,attributes”]

{ “Version”: “2012-10-17”, “Statement”: [{ “Sid”: “AssumeCDKRoles”, “Effect”: “Allow”, “Action”: “sts:AssumeRole”, “Resource”: “\*“,”Condition”: { “StringEquals”: { “iam:ResourceTag/aws-cdk:bootstrap-role”: [ “image-publishing”, “file-publishing”, “deploy”, “lookup” ] } } }] } —

[#best-practices-security-permissions-deployments-roles] ==== Modify the permissions for the roles assumed during deployment

By modifying permissions for the roles assumed during deployment, you can manage the actions that can be performed during deployment. To modify permissions, you create your own IAM roles and specify them when bootstrapping your environment. When you customize bootstrapping, you will have to customize synthesis. For general instructions, see xref:bootstrapping-customizing[Customize {aws} CDK bootstrapping].

[#best-practices-security-permissions-deployments-creds] ==== Modify the security credentials and roles used during deployment

The roles and bootstrap resources that are used during deployments are determined by the CDK stack synthesizer that you use. To modify this behavior, you can customize synthesis. For more information, see xref:configure-synth[Configure and perform CDK stack synthesis].

[#best-practices-security-permissions-deployments-least] ==== Considerations for granting least privilege access

Granting least privilege access is a security best practice that we recommend that you consider as you develop your security strategy. For more information, see link:https://docs.aws.amazon.com/wellarchitected/latest/security-pillar/sec\_permissions\_least\_privileges.html[SEC03-BP02 Grant least privilege access] in the *{aws} Well-Architected Framework Guide*.

Often, granting least privilege access involves restricting IAM policies to the minimum access necessary to perform a given task. Attempting to grant least privilege access through fine-grained permissions with the CDK using this approach can impact CDK deployments and cause you to have to create wider-scoped permissions than you`’d like. The following are a few things to consider when using this approach:

* Determining an exhaustive list of permissions that allow developers to use the {aws} CDK to provision infrastructure through CloudFormation is difficult and complex.
* If you want to be fine-grained, permissions may become too long to fit within the maximum length of IAM policy documents.
* Providing an incomplete set of permissions can severely impact developer productivity and deployments.

With the CDK, deployments are performed using CloudFormation. CloudFormation initiates a set of {aws} API calls in order using the permissions that are provided. The permissions necessary at any point in time depends on many factors:

* The {aws} services that are being modified. Specifically, the resources and properties that are being used and changed.
* The current state of the CloudFormation stack.
* Issues that may occur during deployments and if rollbacks are needed, which will require Delete permissions in addition to Create.

When the provided permissions are incomplete, manual intervention will be required. The following are a few examples:

* If you discover incomplete permissions during roll forward, you’ll need to pause deployment, and take time to discuss and provision new permissions before continuing.
* If deployment rolls back and the permissions to apply the roll back are missing, it may leave your CloudFormation stack in a state that will require a lot of manual work to recover from.

Since this approach can result in complications and severely limit developer productivity, we don’t recommend it. Instead, we recommend implementing guardrails and preventing bypass.

[#best-practices-security-permissions-deployments-guardrails] ==== Implementing guardrails and preventing bypass

You can implement guardrails, compliance rules, auditing, and monitoring by using services such as {aws} Control Tower, {aws} Config, {aws} CloudTrail, {aws} Security Hub, and others. With this approach, you grant developers permission to do everything, except tampering with the existing validation mechanisms. Developers have the freedom to implement changes quickly, as long as they stay within policy. This is the approach we recommend when using the {aws} CDK. For more information on guardrails, see https://docs.aws.amazon.com/wellarchitected/latest/management-and-governance-guide/controls.html[Controls] in the *Management and Governance Cloud Environment Guide*.

We also recommend using *permissions boundaries* or *service control policies (SCPs)* as a way of implementing guardrails. For more information on implementing permissions boundaries with the {aws} CDK, see xref:customize-permissions-boundaries[Create and apply permissions boundaries for the {aws} CDK].

If you are using any compliance control mechanisms, set them up during the bootstrapping phase. Make sure that the CloudFormationExecutionRole or developer-accessible identities have policies or permissions boundaries attached that prevent bypass of the mechanisms that you put in place. The appropriate policies depends on the specific mechanisms that you use.

[#best-practices-security-permissions-resources] === Manage permissions between resources provisioned by the {aws} CDK

How you manage permissions between resources that are provisioned by the {aws} CDK depends on whether you allow the CDK to create roles and policies.

When you use L2 constructs from the {aws} Construct Library to define your infrastructure, you can use the provided grant methods to provision permissions between resources. With grant methods, you specify the type of access you want between resources and the {aws} CDK provisions least privilege IAM roles to accomplish your intent. This approach meets security requirements for most organizations while being efficient for developers. For more information, see xref:define-iam-l2[Define permissions for L2 constructs with the {aws} CDK].

If you want to work around this feature by replacing the automatically generated roles with manually created ones, consider the following:

* Your IAM roles will need to be manually created, slowing down application development.
* When IAM roles need to be manually created and managed, people will often combine multiple logical roles into a single role to make them easier to manage. This runs counter to the least privilege principle.
* Since these roles will need to be created before deployment, the resources that need to be referenced will not yet exist. Therefore, you`’ll need to use wildcards, which runs counter to the least privilege principle.
* A common workaround to using wildcards is to mandate that all resources be given a predictable name. However, this interferes with CloudFormation`’s ability to replace resources when necessary and may slow down or block development. Because of this, we recommend that you allow CloudFormation to create unique resource names for you.
* It will be impossible to perform continuous delivery since manual actions must be performed prior to every deployment.

When organizations want to prevent the CDK from creating roles, it is usually to prevent developers from being able to create IAM roles. The concern is that by giving developers permission to create IAM roles using the {aws} CDK, they could possibly elevate their own privileges. To mitigate against this, we recommend using *permission boundaries* or *service control policies (SCPs)*. With permission boundaries, you can set limits for what developers and the CDK are allowed to do. For more information on using permission boundaries with the CDK, see xref:customize-permissions-boundaries[Create and apply permissions boundaries for the {aws} CDK].

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#best-practices] = Best practices for developing and deploying cloud infrastructure with the {aws} CDK :info\_titleabbrev: Best practices

// Content start

With the {aws} CDK, developers or administrators can define their cloud infrastructure by using a supported programming language. CDK applications should be organized into logical units, such as API, database, and monitoring resources, and optionally have a pipeline for automated deployments. The logical units should be implemented as constructs including the following:

* Infrastructure (such as Amazon S3 buckets, Amazon RDS databases, or an Amazon VPC network)
* Runtime code (such as {aws} Lambda functions)
* Configuration code

Stacks define the deployment model of these logical units. For a more detailed introduction to the concepts behind the CDK, see xref:getting-started[Getting started with the {aws} CDK].

The {aws} CDK reflects careful consideration of the needs of our customers and internal teams and of the failure patterns that often arise during the deployment and ongoing maintenance of complex cloud applications. We discovered that failures are often related to “out-of-band” changes to an application that aren’t fully tested, such as configuration changes. Therefore, we developed the {aws} CDK around a model in which your entire application is defined in code, not only business logic but also infrastructure and configuration. That way, proposed changes can be carefully reviewed, comprehensively tested in environments resembling production to varying degrees, and fully rolled back if something goes wrong.

image::./images/all-in-one.jpg[“Software development lifecycle icons representing infrastructure, application, source code, configuration, and deployment.”,scaledwidth=100%]

At deployment time, the {aws} CDK synthesizes a cloud assembly that contains the following:

* {aws} CloudFormation templates that describe your infrastructure in all target environments
* File assets that contain your runtime code and their supporting files

With the CDK, every commit in your application’s main version control branch can represent a complete, consistent, deployable version of your application. Your application can then be deployed automatically whenever a change is made.

The philosophy behind the {aws} CDK leads to our recommended best practices, which we have divided into four broad categories.

* xref:best-practices-organization[Organization best practices]
* xref:best-practices-code[Coding best practices]
* xref:best-practices-constructs[Construct best practices]
* xref:best-practices-apps[Application best practices]

= [TIP]

Also consider https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html[best practices for {aws} CloudFormation] and the individual {aws} services that you use, where applicable to CDK-defined infrastructure.

====

[#best-practices-organization] == Organization best practices

In the beginning stages of {aws} CDK adoption, it’s important to consider how to set up your organization for success. It’s a best practice to have a team of experts responsible for training and guiding the rest of the company as they adopt the CDK. The size of this team might vary, from one or two people at a small company to a full-fledged Cloud Center of Excellence (CCoE) at a larger company. This team is responsible for setting standards and policies for cloud infrastructure at your company, and also for training and mentoring developers.

The CCoE might provide guidance on what programming languages should be used for cloud infrastructure. Details will vary from one organization to the next, but a good policy helps make sure that developers can understand and maintain the company’s cloud infrastructure.

The CCoE also creates a “landing zone” that defines your organizational units within {aws}. A landing zone is a pre-configured, secure, scalable, multi-account {aws} environment based on best practice blueprints. To tie together the services that make up your landing zone, you can use https://aws.amazon.com/controltower/[{aws} Control Tower], which configures and manages your entire multi-account system from a single user interface.

Development teams should be able to use their own accounts for testing and deploy new resources in these accounts as needed. Individual developers can treat these resources as extensions of their own development workstation. Using xref:cdk-pipeline[CDK Pipelines], the {aws} CDK applications can then be deployed via a CI/CD account to testing, integration, and production environments (each isolated in its own {aws} Region or account). This is done by merging the developers’ code into your organization’s canonical repository.

image::./images/best-practice-deploy-to-multiple-accounts.png[“Diagram showing deployment process from developer accounts to multiple target accounts via CI/CD pipeline.”,scaledwidth=100%]

[#best-practices-code] == Coding best practices

This section presents best practices for organizing your {aws} CDK code. The following diagram shows the relationship between a team and that team’s code repositories, packages, applications, and construct libraries.

image::./images/code-organization.jpg[“Diagram showing team’s code organization: repository, package, CDK app or construct library.”,scaledwidth=100%]

[#best-practices-code-kiss] *Start simple and add complexity only when you need it*:: + The guiding principle for most of our best practices is to keep things simple as possible–but no simpler. Add complexity only when your requirements dictate a more complicated solution. With the {aws} CDK, you can refactor your code as necessary to support new requirements. You don’t have to architect for all possible scenarios upfront.

[#best-practices-code-well-architected] *Align with the {aws} Well-Architected Framework*:: + The https://aws.amazon.com/architecture/well-architected/[{aws} Well-Architected] Framework defines a *component* as the code, configuration, and {aws} resources that together deliver against a requirement. A component is often the unit of technical ownership, and is decoupled from other components. The term *workload* is used to identify a set of components that together deliver business value. A workload is usually the level of detail that business and technology leaders communicate about. + An {aws} CDK application maps to a component as defined by the {aws} Well-Architected Framework. {aws} CDK apps are a mechanism to codify and deliver Well-Architected cloud application best practices. You can also create and share components as reusable code libraries through artifact repositories, such as {aws} CodeArtifact.

[#best-practices-code-package] *Every application starts with a single package in a single repository*:: + A single package is the entry point of your {aws} CDK app. Here, you define how and where to deploy the different logical units of your application. You also define the CI/CD pipeline to deploy the application. The app’s constructs define the logical units of your solution. + Use additional packages for constructs that you use in more than one application. (Shared constructs should also have their own lifecycle and testing strategy.) Dependencies between packages in the same repository are managed by your repo’s build tooling. + Although it’s possible, we don’t recommend putting multiple applications in the same repository, especially when using automated deployment pipelines. Doing this increases the “blast radius” of changes during deployment. When there are multiple applications in a repository, changes to one application trigger deployment of the others (even if the others haven’t changed). Furthermore, a break in one application prevents the other applications from being deployed.

[#best-practices-code-repo] *Move code into repositories based on code lifecycle or team ownership*:: + When packages begin to be used in multiple applications, move them to their own repository. This way, the packages can be referenced by application build systems that use them, and they can also be updated on cadences independent of the application lifecycles. However, at first it might make sense to put all shared constructs in one repository. + Also, move packages to their own repository when different teams are working on them. This helps enforce access control. + To consume packages across repository boundaries, you need a private package repository–similar to NPM, PyPi, or Maven Central, but internal to your organization. You also need a release process that builds, tests, and publishes the package to the private package repository. https://docs.aws.amazon.com/codeartifact/latest/ug/[CodeArtifact] can host packages for most popular programming languages. + Dependencies on packages in the package repository are managed by your language’s package manager, such as NPM for TypeScript or JavaScript applications. Your package manager helps to make sure that builds are repeatable. It does this by recording the specific versions of every package that your application depends on. It also lets you upgrade those dependencies in a controlled manner. + Shared packages need a different testing strategy. For a single application, it might be good enough to deploy the application to a testing environment and confirm that it still works. But shared packages must be tested independently of the consuming application, as if they were being released to the public. (Your organization might choose to actually release some shared packages to the public.) + Keep in mind that a construct can be arbitrarily simple or complex. A Bucket is a construct, but CameraShopWebsite could be a construct, too.

[#best-practices-code-all] *Infrastructure and runtime code live in the same package*:: + In addition to generating {aws} CloudFormation templates for deploying infrastructure, the {aws} CDK also bundles runtime assets like Lambda functions and Docker images and deploys them alongside your infrastructure. This makes it possible to combine the code that defines your infrastructure and the code that implements your runtime logic into a single construct. It’s a best practice to do this. These two kinds of code don’t need to live in separate repositories or even in separate packages. + To evolve the two kinds of code together, you can use a self-contained construct that completely describes a piece of functionality, including its infrastructure and logic. With a self-contained construct, you can test the two kinds of code in isolation, share and reuse the code across projects, and version all the code in sync.

[#best-practices-constructs] == Construct best practices

This section contains best practices for developing constructs. Constructs are reusable, composable modules that encapsulate resources. They’re the building blocks of {aws} CDK apps.

[#best-practices-constructs-model] *Model with constructs, deploy with stacks*:: + Stacks are the unit of deployment: everything in a stack is deployed together. So when building your application’s higher-level logical units from multiple {aws} resources, represent each logical unit as a https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct], not as a https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack]. Use stacks only to describe how your constructs should be composed and connected for your various deployment scenarios. + For example, if one of your logical units is a website, the constructs that make it up (such as an Amazon S3 bucket, API Gateway, Lambda functions, or Amazon RDS tables) should be composed into a single high-level construct. Then that construct should be instantiated in one or more stacks for deployment. + By using constructs for building and stacks for deploying, you improve reuse potential of your infrastructure and give yourself more flexibility in how it’s deployed.

[#best-practices-constructs-config] *Configure with properties and methods, not environment variables*:: + Environment variable lookups inside constructs and stacks are a common anti-pattern. Both constructs and stacks should accept a properties object to allow for full configurability completely in code. Doing otherwise introduces a dependency on the machine that the code will run on, which creates yet more configuration information that you have to track and manage. + In general, environment variable lookups should be limited to the top level of an {aws} CDK app. They should also be used to pass in information that’s needed for running in a development environment. For more information, see xref:environments[Environments for the {aws} CDK].

[#best-practices-constructs-test] *Unit test your infrastructure*:: + To consistently run a full suite of unit tests at build time in all environments, avoid network lookups during synthesis and model all your production stages in code. (These best practices are covered later.) If any single commit always results in the same generated template, you can trust the unit tests that you write to confirm that the generated templates look the way you expect. For more information, see xref:testing[Test {aws} CDK applications].

[#best-practices-constructs-logicalid] *Don’t change the logical ID of stateful resources*:: + Changing the logical ID of a resource results in the resource being replaced with a new one at the next deployment. For stateful resources like databases and S3 buckets, or persistent infrastructure like an Amazon VPC, this is seldom what you want. Be careful about any refactoring of your {aws} CDK code that could cause the ID to change. Write unit tests that assert that the logical IDs of your stateful resources remain static. The logical ID is derived from the id you specify when you instantiate the construct, and the construct’s position in the construct tree. For more information, see xref:identifiers-logical-ids[Logical IDs].

[#best-practices-constructs-compliance] *Constructs aren’t enough for compliance*:: + Many enterprise customers write their own wrappers for L2 constructs (the “curated” constructs that represent individual {aws} resources with built-in sane defaults and best practices). These wrappers enforce security best practices such as static encryption and specific IAM policies. For example, you might create a MyCompanyBucket that you then use in your applications in place of the usual Amazon S3 Bucket construct. This pattern is useful for surfacing security guidance early in the software development lifecycle, but don’t rely on it as the sole means of enforcement. + Instead, use {aws} features such as link:https://docs.aws.amazon.com/organizations/latest/userguide/orgs\_manage\_policies\_scps.html[service control policies] and link:https://docs.aws.amazon.com/IAM/latest/UserGuide/access\_policies\_boundaries.html[permission boundaries] to enforce your security guardrails at the organization level. Use xref:aspects[Aspects and the {aws} CDK] or tools like https://github.com/aws-cloudformation/cloudformation-guard[CloudFormation Guard] to make assertions about the security properties of infrastructure elements before deployment. Use {aws} CDK for what it does best. + Finally, keep in mind that writing your own “L2+” constructs might prevent your developers from taking advantage of {aws} CDK packages such as https://docs.aws.amazon.com/solutions/latest/constructs/welcome.html[{aws} Solutions Constructs] or third-party constructs from Construct Hub. These packages are typically built on standard {aws} CDK constructs and won’t be able to use your wrapper constructs.

[#best-practices-apps] == Application best practices

In this section we discuss how to write your {aws} CDK applications, combining constructs to define how your {aws} resources are connected.

[#best-practices-apps-synth] *Make decisions at synthesis time*:: + Although {aws} CloudFormation lets you make decisions at deployment time (using Conditions, { Fn::If }, and Parameters), and the {aws} CDK gives you some access to these mechanisms, we recommend against using them. The types of values that you can use and the types of operations you can perform on them are limited compared to what’s available in a general-purpose programming language. + Instead, try to make all decisions, such as which construct to instantiate, in your {aws} CDK application by using your programming language’s if statements and other features. For example, a common CDK idiom, iterating over a list and instantiating a construct with values from each item in the list, simply isn’t possible using {aws} CloudFormation expressions. + Treat {aws} CloudFormation as an implementation detail that the {aws} CDK uses for robust cloud deployments, not as a language target. You’re not writing {aws} CloudFormation templates in TypeScript or Python, you’re writing CDK code that happens to use CloudFormation for deployment.

[#best-practices-apps-names] *Use generated resource names, not physical names*:: + Names are a precious resource. Each name can only be used once. Therefore, if you hardcode a table name or bucket name into your infrastructure and application, you can’t deploy that piece of infrastructure twice in the same account. (The name we’re talking about here is the name specified by, for example, the bucketName property on an Amazon S3 bucket construct.) + What’s worse, you can’t make changes to the resource that require it to be replaced. If a property can only be set at resource creation, such as the KeySchema of an Amazon DynamoDB table, then that property is immutable. Changing this property requires a new resource. However, the new resource must have the same name in order to be a true replacement. But it can’t have the same name while the existing resource is still using that name. + A better approach is to specify as few names as possible. If you omit resource names, the {aws} CDK will generate them for you in a way that won’t cause problems. Suppose you have a table as a resource. You can then pass the generated table name as an environment variable into your {aws} Lambda function. In your {aws} CDK application, you can reference the table name as table.tableName. Alternatively, you can generate a configuration file on your Amazon EC2 instance on startup, or write the actual table name to the {aws} Systems Manager Parameter Store so your application can read it from there. + If the place you need it is another {aws} CDK stack, that’s even more straightforward. Supposing that one stack defines the resource and another stack needs to use it, the following applies: +

* If the two stacks are in the same {aws} CDK app, pass a reference between the two stacks. For example, save a reference to the resource’s construct as an attribute of the defining stack (this.stack.uploadBucket = amzn-s3-demo-bucket). Then, pass that attribute to the constructor of the stack that needs the resource.
* When the two stacks are in different {aws} CDK apps, use a static from method to use an externally defined resource based on its ARN, name, or other attributes. (For example, use Table.fromArn() for a DynamoDB table). Use the CfnOutput construct to print the ARN or other required value in the output of cdk deploy, or look in the {aws} Management Console. Alternatively, the second app can read the CloudFormation template generated by the first app and retrieve that value from the Outputs section.

[#best-practices-apps-removal-logs] *Define removal policies and log retention*:: + The {aws} CDK attempts to keep you from losing data by defaulting to policies that retain everything you create. For example, the default removal policy on resources that contain data (such as Amazon S3 buckets and database tables) is not to delete the resource when it is removed from the stack. Instead, the resource is orphaned from the stack. Similarly, the CDK’s default is to retain all logs forever. In production environments, these defaults can quickly result in the storage of large amounts of data that you don’t actually need, and a corresponding {aws} bill. + Consider carefully what you want these policies to be for each production resource and specify them accordingly. Use xref:aspects[Aspects and the {aws} CDK] to validate the removal and logging policies in your stack.

[#best-practices-apps-separate] *Separate your application into multiple stacks as dictated by deployment requirements*:: + There is no hard and fast rule to how many stacks your application needs. You’ll usually end up basing the decision on your deployment patterns. Keep in mind the following guidelines: +

* It’s typically more straightforward to keep as many resources in the same stack as possible, so keep them together unless you know you want them separated.
* Consider keeping stateful resources (like databases) in a separate stack from stateless resources. You can then turn on termination protection on the stateful stack. This way, you can freely destroy or create multiple copies of the stateless stack without risk of data loss.
* Stateful resources are more sensitive to construct renaming–renaming leads to resource replacement. Therefore, don’t nest stateful resources inside constructs that are likely to be moved around or renamed (unless the state can be rebuilt if lost, like a cache). This is another good reason to put stateful resources in their own stack.

[#best-practices-apps-context] *Commit cdk.context.json to avoid non-deterministic behavior*:: + Determinism is key to successful {aws} CDK deployments. An {aws} CDK app should have essentially the same result whenever it is deployed to a given environment. + Since your {aws} CDK app is written in a general-purpose programming language, it can execute arbitrary code, use arbitrary libraries, and make arbitrary network calls. For example, you could use an {aws} SDK to retrieve some information from your {aws} account while synthesizing your app. Recognize that doing so will result in additional credential setup requirements, increased latency, and a chance, however small, of failure every time you run cdk synth. + Never modify your {aws} account or resources during synthesis. Synthesizing an app should not have side effects. Changes to your infrastructure should happen only in the deployment phase, after the {aws} CloudFormation template has been generated. This way, if there’s a problem, {aws} CloudFormation can automatically roll back the change. To make changes that can’t be easily made within the {aws} CDK framework, use link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.custom\_resources-readme.html[custom resources] to execute arbitrary code at deployment time. + Even strictly read-only calls are not necessarily safe. Consider what happens if the value returned by a network call changes. What part of your infrastructure will that impact? What will happen to already-deployed resources? Following are two example situations in which a sudden change in values might cause a problem. + –

* If you provision an Amazon VPC to all available Availability Zones in a specified Region, and the number of AZs is two on deployment day, then your IP space gets split in half. If {aws} launches a new Availability Zone the next day, the next deployment after that tries to split your IP space into thirds, requiring all subnets to be recreated. This probably won’t be possible because your Amazon EC2 instances are still running, and you’ll have to clean this up manually.
* {blank}
* == If you query for the latest Amazon Linux machine image and deploy an Amazon EC2 instance, and the next day a new image is released, a subsequent deployment picks up the new AMI and replaces all your instances. This might not be what you expected to happen.
* These situations can be pernicious because the {aws}-side change might occur after months or years of successful deployments. Suddenly your deployments are failing “for no reason” and you long ago forgot what you did and why.
* Fortunately, the {aws} CDK includes a mechanism called *context providers* to record a snapshot of non-deterministic values. This allows future synthesis operations to produce exactly the same template as they did when first deployed. The only changes in the new template are the changes that *you* made in your code. When you use a construct’s .fromLookup() method, the result of the call is cached in cdk.context.json. You should commit this to version control along with the rest of your code to make sure that future executions of your CDK app use the same value. The CDK Toolkit includes commands to manage the context cache, so you can refresh specific entries when you need to. For more information, see xref:context[Context values and the {aws} CDK].
* If you need some value (from {aws} or elsewhere) for which there is no native CDK context provider, we recommend writing a separate script. The script should retrieve the value and write it to a file, then read that file in your CDK app. Run the script only when you want to refresh the stored value, not as part of your regular build process.

[#best-practices-apps-roles] *Let the {aws} CDK manage roles and security groups*:: + With the {aws} CDK construct library’s grant() convenience methods, you can create {aws} Identity and Access Management roles that grant access to one resource by another using minimally scoped permissions. For example, consider a line like the following: + [source,javascript,subs=“verbatim,attributes”] — amzn-s3-demo-bucket.grantRead(myLambda) — + This single line adds a policy to the Lambda function’s role (which is also created for you). That role and its policies are more than a dozen lines of CloudFormation that you don’t have to write. The {aws} CDK grants only the minimal permissions required for the function to read from the bucket. + If you require developers to always use predefined roles that were created by a security team, {aws} CDK coding becomes much more complicated. Your teams could lose a lot of flexibility in how they design their applications. A better alternative is to use link:https://docs.aws.amazon.com/organizations/latest/userguide/orgs\_manage\_policies\_scps.html[service control policies] and link:https://docs.aws.amazon.com/IAM/latest/UserGuide/access\_policies\_boundaries.html[permission boundaries] to make sure that developers stay within the guardrails.

[#best-practices-apps-stages] *Model all production stages in code*:: + In traditional {aws} CloudFormation scenarios, your goal is to produce a single artifact that is parameterized so that it can be deployed to various target environments after applying configuration values specific to those environments. In the CDK, you can, and should, build that configuration into your source code. Create a stack for your production environment, and create a separate stack for each of your other stages. Then, put the configuration values for each stack in the code. Use services like https://aws.amazon.com/secrets-manager/[Secrets Manager] and https://aws.amazon.com/systems-manager/[Systems Manager] Parameter Store for sensitive values that you don’t want to check in to source control, using the names or ARNs of those resources. + When you synthesize your application, the cloud assembly created in the cdk.out folder contains a separate template for each environment. Your entire build is deterministic. There are no out-of-band changes to your application, and any given commit always yields the exact same {aws} CloudFormation template and accompanying assets. This makes unit testing much more reliable.

[#best-practices-apps-measure] *Measure everything*:: + Achieving the goal of full continuous deployment, with no human intervention, requires a high level of automation. That automation is only possible with extensive amounts of monitoring. To measure all aspects of your deployed resources, create metrics, alarms, and dashboards. Don’t stop at measuring things like CPU usage and disk space. Also record your business metrics, and use those measurements to automate deployment decisions like rollbacks. Most of the L2 constructs in {aws} CDK have convenience methods to help you create metrics, such as the metricUserErrors() method on the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_dynamodb.Table.html[dynamodb.Table] class.

include::best-practices-security.adoc[leveloffset=+1]

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#build-containers] = Build and deploy container image assets in CDK apps :info\_titleabbrev: Build and deploy container image assets :keywords: Docker, Dockerfile, Finch, Container, Container image, Container image asset, Amazon ECR, {aws} CDK, {aws} CDK CLI, Deploy, Build

== [abstract]

## When you build container image assets with the {aws} Cloud Development Kit ({aws} CDK), Docker is utilized by default to perform these actions. If you want to use a different container management tool, you can replace Docker through the CDK\_DOCKER environment variable.

// Content start

When you build container image assets with the {aws} Cloud Development Kit ({aws} CDK), Docker is utilized by default to perform these actions. If you want to use a different container management tool, you can replace Docker through the CDK\_DOCKER environment variable.

[#build-containers-intro-example] == Example: Build and publish a container image asset with the {aws} CDK

The following is a simple example of an {aws} CDK app that builds and publishes a container asset to Amazon Elastic Container Registry (Amazon ECR) using Docker by default:

*Project structure*:: + [source,none] — my-cdk-app/ ├── lib/ │ ├── my-stack.ts │ └── docker/ │ ├── Dockerfile │ └── app/ │ └── index.js ├── bin/ │ └── my-cdk-app.ts ├── package.json ├── tsconfig.json └── cdk.json —

*Dockerfile*:: + [source,auto] — FROM public.ecr.aws/lambda/nodejs:16

= Copy application code

COPY app/ /var/task/

= (Optional) Install dependencies

= RUN npm install

= The Lambda Node.js base image looks for index.handler by default

’’’

*Application code*:: + In lib/docker/app/index.js: + [source,javascript] — console.log(“Hello from inside the container!”); —

*CDK stack*:: + [source,javascript] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import \* as ecr\_assets from ‘aws-cdk-lib/aws-ecr-assets’;

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

…. // Define a Docker image asset const dockerImageAsset = new ecr\_assets.DockerImageAsset(this, ‘MyDockerImage’, { directory: ‘lib/docker’, // Path to the directory containing the Dockerfile });

// Output the ECR URI new cdk.CfnOutput(this, ‘ECRImageUri’, { value: dockerImageAsset.imageUri, }); } } —- ….

*CDK app*:: + [source,javascript] — #!/usr/bin/env node import \* as cdk from ‘aws-cdk-lib’; import { MyStack } from ‘../lib/my-stack’;

const app = new cdk.App(); new MyStack(app, ‘MyStack’); —

When we run cdk deploy, the {aws} Cloud Development Kit ({aws} CDK) Command Line Interface (CLI) does the following:

. *Build the Docker image* – Run docker build locally based on the Dockerfile in the specified directory (lib/docker). . *Tag the image* – Run docker tag to tag the built image with a unique hash, based on the image contents. . *Publish to Amazon ECR* – Run docker push to publish the container image to an Amazon ECR repository. This repository must already exist. It is created during the default bootstrapping process. . *Output the Image URI* – After a successful deployment, the Amazon ECR URI of the published container image is output in your command prompt. This is the URI of our Docker image in Amazon ECR.

[#build-container-replace] == How to replace Docker with another container management tool

Use the CDK\_DOCKER environment variable to specify the path to the binary of your replacement container management tool. The following is an example of replacing Docker with [noloc]Finch:

== [source,none,subs=“verbatim,attributes”]

$ which finch /usr/local/bin/finch # Locate the path to the binary

$ export CDK\_DOCKER=‘/usr/local/bin/finch’ # Set the environment variable

== $ cdk deploy # Deploy using the replacement

Aliasing or linking is not supported. To replace Docker, you must use the CDK\_DOCKER environment variable.

[#build-container-supported] == Supported Docker drop-in replacement engines

[noloc]Finch is supported, although there may be some Docker features that are unavailable or may work differently as the tool evolves. For more information on Finch, see https://aws.amazon.com/blogs/opensource/ready-for-flight-announcing-finch-1-0-ga/[Ready for Flight: Announcing Finch 1.0 GA!] in the *{aws} Open Source Blog*.

Other container management tools may work. The CDK doesn’t check which Docker replacement you are using to determine if it’s supported. If the tool has equivalent Docker commands and behaves similarly, it should work.

include::attributes.txt[]

// Attributes [.topic] [#develop] = Develop {aws} CDK applications :keywords: {aws} CDK, IaC, Infrastructure as code, {aws}, {aws} Cloud, serverless, modern applications

== [abstract]

## Manage your infrastructure on {aws} by developing applications using the {aws} Cloud Development Kit ({aws} CDK).

// Content start

Manage your infrastructure on {aws} by developing applications using the {aws} Cloud Development Kit ({aws} CDK).

[#develop-prerequisites] == Prerequisites

Before you can start developing applications, complete all set up steps in xref:getting-started[Getting started with the {aws} CDK].

[#develop-overview] == Developing {aws} CDK applications overview

At a high-level, developing CDK applications involves the following steps:

. *Create a CDK project* – A CDK xref:projects[project] consists of the files and folders that store and organize your CDK code. . *Create a CDK app* – Within a CDK project, you use the App construct to define a CDK xref:apps[application]. . *Create a CDK stack* – Within the scope of your CDK app, you define one or more CDK xref:stacks[stacks]. . *Define your infrastructure* – Within the scope of a CDK stack, you use xref:constructs[constructs] from the {aws} Construct Library to define the {aws} resources and properties that will become your infrastructure. Using a general-purpose programming xref:languages[language] of your choice, you can use logic, such as conditional statements and loops, to define your infrastructure based on certain conditions.

[#develop-gs] == Get started with developing CDK applications

To get started, you can use the {aws} CDK Command Line Interface ({aws} CDK CLI) cdk init command. Provide the --language option to specify the programming language to use. This command creates a starting CDK project and imports the main {aws} Construct Library and core modules.

[#develop-library] == Import and use the {aws} CDK Library

After you create a CDK project, import and use constructs from the {aws} CDK Library to begin defining your infrastructure. For instructions, see xref:work-with[Work with the {aws} CDK library].

[#develop-next] == Next steps

When ready to deploy your application, use the CDK CLI cdk deploy command. For instructions, see xref:deploy[Deploy {aws} CDK applications].

include::usage-data.adoc[leveloffset=+1]

include::cfn-layer.adoc[leveloffset=+1]

include::get-env-var.adoc[leveloffset=+1]

include::get-cfn-val.adoc[leveloffset=+1]

include::use-cfn-template.adoc[leveloffset=+1]

include::get-ssm-val.adoc[leveloffset=+1]

include::get-sm-val.adoc[leveloffset=+1]

include::set-cw-alarm.adoc[leveloffset=+1]

include::get-context-var.adoc[leveloffset=+1]

include::use-cfn-public-registry.adoc[leveloffset=+1]

include::define-iam-l2.adoc[leveloffset=+1]

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Projects :keywords: {aws} CDK, {aws} CloudFormation, IaC, Infrastructure as code, CDK projects, {aws} CDK concepts

[[projects,projects.title]] = {aws} CDK projects

== [abstract]

## An {aws} Cloud Development Kit ({aws} CDK) project represents the files and folders that contain your CDK code. Contents will vary based on your programming language.

// Content start

An {aws} Cloud Development Kit ({aws} CDK) project represents the files and folders that contain your CDK code. Contents will vary based on your programming language.

You can create your {aws} CDK project manually or with the {aws} CDK Command Line Interface ({aws} CDK CLI) cdk init command. In this topic, we will refer to the project structure and naming conventions of files and folders created by the {aws} CDK CLI. You can customize and organize your CDK projects to fit your needs.

= [NOTE]

Project structure created by the {aws} CDK CLI may vary across versions over time.

====

[#projects-universal] == Universal files and folders

[[projects-universal-git]] .git:: If you have git installed, the {aws} CDK CLI automatically initializes a [.noloc]Git repository for your project. The .git directory contains information about the repository.

[[projects-universal-gitignore]] .gitignore:: Text file used by [.noloc]Git to specify files and folders to ignore.

[[projects-universal-readme]] README.md:: Text file that provides you with basic guidance and important information for managing your {aws} CDK project. Modify this file as necessary to document important information regarding your CDK project.

[[projects-universal-cdk]] cdk.json:: Configuration file for the {aws} CDK. This file provides instruction to the {aws} CDK CLI regarding how to run your app.

[#projects-specific] == Language-specific files and folders

The following files and folders are unique to each supported programming language.

==== [role=“tablist”] TypeScript:: The following is an example project created in the my-cdk-ts-project directory using the cdk init --language typescript command: + [source,none,subs=“verbatim,attributes”] — my-cdk-ts-project ├── .git ├── .gitignore ├── .npmignore ├── README.md ├── bin │ └── my-cdk-ts-project.ts ├── cdk.json ├── jest.config.js ├── lib │ └── my-cdk-ts-project-stack.ts ├── node\_modules ├── package-lock.json ├── package.json ├── test │ └── my-cdk-ts-project.test.ts └── tsconfig.json —

.npmignore::: File that specifies which files and folders to ignore when publishing a package to the npm registry. This file is similar to .gitignore, but is specific to npm packages.

bin/my-cdk-ts-project.ts::: The *application file* defines your CDK app. CDK projects can contain one or more application files. Application files are stored in the bin folder. + The following is an example of a basic application file that defines a CDK app: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node import ‘source-map-support/register’; import \* as cdk from ‘aws-cdk-lib’; import { MyCdkTsProjectStack } from ‘../lib/my-cdk-ts-project-stack’;

const app = new cdk.App(); new MyCdkTsProjectStack(app, ‘MyCdkTsProjectStack’); —

jest.config.js::: Configuration file for [.noloc]Jest. [.noloc]*Jest* is a popular JavaScript testing framework.

lib/my-cdk-ts-project-stack.ts::: The *stack file* defines your CDK stack. Within your stack, you define {aws} resources and properties using constructs. + The following is an example of a basic stack file that defines a CDK stack: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’;

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

// code that defines your resources and properties go here } } —

node\_modules::: Common folder in [.noloc]Node.js projects that contain dependencies for your project.

package-lock.json::: Metadata file that works with the package.json file to manage versions of dependencies.

package.json::: Metadata file that is commonly used in [.noloc]Node.js projects. This file contains information about your CDK project such as the project name, script definitions, dependencies, and other import project-level information.

test/my-cdk-ts-project.test.ts::: A test folder is created to organize tests for your CDK project. A sample test file is also created. + You can write tests in TypeScript and use [.noloc]Jest to compile your TypeScript code before running tests.

tsconfig.json::: Configuration file used in TypeScript projects that specifies compiler options and project settings.

JavaScript:: The following is an example project created in the my-cdk-js-project directory using the cdk init --language javascript command: + [source,none,subs=“verbatim,attributes”] — my-cdk-js-project ├── .git ├── .gitignore ├── .npmignore ├── README.md ├── bin │ └── my-cdk-js-project.js ├── cdk.json ├── jest.config.js ├── lib │ └── my-cdk-js-project-stack.js ├── node\_modules ├── package-lock.json ├── package.json └── test └── my-cdk-js-project.test.js —

.npmignore::: File that specifies which files and folders to ignore when publishing a package to the npm registry. This file is similar to .gitignore, but is specific to npm packages.

bin/my-cdk-js-project.js::: The *application file* defines your CDK app. CDK projects can contain one or more application files. Application files are stored in the bin folder. + The following is an example of a basic application file that defines a CDK app: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node

const cdk = require(‘aws-cdk-lib’); const { MyCdkJsProjectStack } = require(‘../lib/my-cdk-js-project-stack’);

const app = new cdk.App(); new MyCdkJsProjectStack(app, ‘MyCdkJsProjectStack’); —

jest.config.js::: Configuration file for [.noloc]Jest. [.noloc]*Jest* is a popular JavaScript testing framework.

lib/my-cdk-js-project-stack.js::: The *stack file* defines your CDK stack. Within your stack, you define {aws} resources and properties using constructs. + The following is an example of a basic stack file that defines a CDK stack: + [source,javascript,subs=“verbatim,attributes”] — const { Stack, Duration } = require(‘aws-cdk-lib’);

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

// code that defines your resources and properties go here } }

== module.exports = { MyCdkJsProjectStack }

node\_modules::: Common folder in [.noloc]Node.js projects that contain dependencies for your project.

package-lock.json::: Metadata file that works with the package.json file to manage versions of dependencies.

package.json::: Metadata file that is commonly used in [.noloc]Node.js projects. This file contains information about your CDK project such as the project name, script definitions, dependencies, and other import project-level information.

test/my-cdk-js-project.test.js::: A test folder is created to organize tests for your CDK project. A sample test file is also created. + You can write tests in JavaScript and use [.noloc]Jest to compile your JavaScript code before running tests.

Python:: The following is an example project created in the my-cdk-py-project directory using the cdk init --language python command: + [source,none,subs=“verbatim,attributes”] — my-cdk-py-project ├── .git ├── .gitignore ├── .venv ├── README.md ├── app.py ├── cdk.json ├── my\_cdk\_py\_project │ ├── *init*.py │ └── my\_cdk\_py\_project\_stack.py ├── requirements-dev.txt ├── requirements.txt ├── source.bat └── tests ├── *init*.py └── unit —

.venv::: The CDK CLI automatically creates a virtual environment for your project. The .venv directory refers to this virtual environment.

app.py::: The *application file* defines your CDK app. CDK projects can contain one or more application files. + The following is an example of a basic application file that defines a CDK app: + [source,python,subs=“verbatim,attributes”] — #!/usr/bin/env python3 import os

import aws\_cdk as cdk

from my\_cdk\_py\_project.my\_cdk\_py\_project\_stack import MyCdkPyProjectStack

app = cdk.App() MyCdkPyProjectStack(app, “MyCdkPyProjectStack”)

== app.synth()

my\_cdk\_py\_project::: Directory that contains your *stack files*. The CDK CLI creates the following here: + –

* +*init*.py+ – An empty Python package definition file.
* {blank}
* == my\_cdk\_py\_project – File that defines your CDK stack. You then define {aws} resources and properties within the stack using constructs.
* The following is an example of a stack file:

## [source,python,subs=“verbatim,attributes”]

* from aws\_cdk import Stack

from constructs import Construct

class MyCdkPyProjectStack(Stack): def *init*(self, scope: Construct, construct\_id: str, **kwargs) -> None: super().*init*(scope, construct\_id,** kwargs)

== # code that defines your resources and properties go here

requirements-dev.txt::: File similar to requirements.txt, but used to manage dependencies specifically for development purposes rather than production.

requirements.txt::: Common file used in Python projects to specify and manage project dependencies.

source.bat::: Batch file for [.noloc]Windows that is used to set up the Python virtual environment.

tests::: Directory that contains tests for your CDK project. + The following is an example of a unit test: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as core import aws\_cdk.assertions as assertions

from my\_cdk\_py\_project.my\_cdk\_py\_project\_stack import MyCdkPyProjectStack

def test\_sqs\_queue\_created(): app = core.App() stack = MyCdkPyProjectStack(app, “my-cdk-py-project”) template = assertions.Template.from\_stack(stack)

template.has\_resource\_properties(“{aws}::SQS::Queue”, { “VisibilityTimeout”: 300 }) —

Java:: The following is an example project created in the my-cdk-java-project directory using the cdk init --language java command: + [source,none,subs=“verbatim,attributes”] — my-cdk-java-project ├── .git ├── .gitignore ├── README.md ├── cdk.json ├── pom.xml └── src ├── main └── test —

pom.xml::: File that contains configuration information and metadata about your CDK project. This file is a part of [.noloc]Maven.

src/main::: Directory containing your *application* and *stack* files. + The following is an example application file: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.amazon.awscdk.App; import software.amazon.awscdk.Environment; import software.amazon.awscdk.StackProps;

import java.util.Arrays;

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

new MyCdkJavaProjectStack(app, “MyCdkJavaProjectStack”, StackProps.builder() .build());

app.synth(); } } — + The following is an example stack file: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps;

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

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

// code that defines your resources and properties go here } } —

src/test::: Directory containing your test files. The following is an example: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.amazon.awscdk.App; import software.amazon.awscdk.assertions.Template; import java.io.IOException;

import java.util.HashMap;

import org.junit.jupiter.api.Test;

public class MyCdkJavaProjectTest {

@Test public void testStack() throws IOException { App app = new App(); MyCdkJavaProjectStack stack = new MyCdkJavaProjectStack(app, “test”);

Template template = Template.fromStack(stack);

template.hasResourceProperties(“{aws}::SQS::Queue”, new HashMap<String, Number>() {{ put(“VisibilityTimeout”, 300); }}); } } —

C#:: The following is an example project created in the my-cdk-csharp-project directory using the cdk init --language csharp command: + [source,none,subs=“verbatim,attributes”] — my-cdk-csharp-project ├── .git ├── .gitignore ├── README.md ├── cdk.json └── src ├── MyCdkCsharpProject └── MyCdkCsharpProject.sln —

src/MyCdkCsharpProject::: Directory containing your *application* and *stack* files. + The following is an example application file: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using System; using System.Collections.Generic; using System.Linq;

namespace MyCdkCsharpProject { sealed class Program { public static void Main(string[] args) { var app = new App(); new MyCdkCsharpProjectStack(app, “MyCdkCsharpProjectStack”, new StackProps{}); app.Synth(); } } } — + The following is an example stack file: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs;

namespace MyCdkCsharpProject { public class MyCdkCsharpProjectStack : Stack { internal MyCdkCsharpProjectStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // code that defines your resources and properties go here } } } — + This directory also contains the following: + –

* GlobalSuppressions.cs – File used to suppress specific compiler warnings or errors across your project.
* {blank}
* == .csproj – XML-based file used to define project settings, dependencies, and build configurations.

src/MyCdkCsharpProject.sln::: [.noloc]Microsoft Visual Studio Solution File used to organize and manage related projects.

[.noloc]Go:: The following is an example project created in the my-cdk-go-project directory using the cdk init --language go command: + [source,none,subs=“verbatim,attributes”] — my-cdk-go-project ├── .git ├── .gitignore ├── README.md ├── cdk.json ├── go.mod ├── my-cdk-go-project.go └── my-cdk-go-project\_test.go —

go.mod::: File that contains module information and is used to manage dependencies and versioning for your [.noloc]Go project.

my-cdk-go-project.go::: File that defines your CDK application and stacks. + The following is an example: + [source,go,subs=“verbatim,attributes”] — package main import ( “github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/constructs-go/constructs/v10” “github.com/aws/jsii-runtime-go” )

type MyCdkGoProjectStackProps struct { awscdk.StackProps }

func NewMyCdkGoProjectStack(scope constructs.Construct, id string, props \*MyCdkGoProjectStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } stack := awscdk.NewStack(scope, &id, &sprops) // The code that defines your resources and properties go here

return stack }

func main() { defer jsii.Close() app := awscdk.NewApp(nil) NewMyCdkGoProjectStack(app, “MyCdkGoProjectStack”, &MyCdkGoProjectStackProps{ awscdk.StackProps{ Env: env(), }, }) app.Synth(nil) }

func env() \*awscdk.Environment {

return nil } —

my-cdk-go-project\_test.go::: File that defines a sample test. + The following is an example: + [source,go,subs=“verbatim,attributes”] — package main

import ( “testing”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/assertions” “github.com/aws/jsii-runtime-go” )

func TestMyCdkGoProjectStack(t \*testing.T) {

// GIVEN app := awscdk.NewApp(nil)

// WHEN stack := NewMyCdkGoProjectStack(app, “MyStack”, nil)

// THEN template := assertions.Template\_FromStack(stack, nil) template.HasResourceProperties(jsii.String(“{aws}::SQS::Queue”), map[string]interface{}{ “VisibilityTimeout”: 300, }) } — ====

:doctype: book

include::attributes.txt[]

// Attributes :https–docs-aws-amazon-com-cdk-api-v2-docs-aws-cdk-lib-Tags-html-removekey-props: https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tags.html#removekey-props :https–docs-aws-amazon-com-cdk-api-v2-docs-aws-cdk-lib-Tags-html-addkey-value-props: https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tags.html#addkey-value-props

[.topic] :info\_titleabbrev: Tags :keywords: {aws} CDK, Tags

[#tagging] = Tags and the {aws} CDK

== [abstract]

## Tags are informational key-value elements that you can add to constructs in your {aws} CDK app. A tag applied to a given construct also applies to all of its taggable children. Tags are included in the {aws} CloudFormation template synthesized from your app and are applied to the {aws} resources it deploys. You can use tags to identify and categorize resources for the following purposes:

// Content start

Tags are informational key-value elements that you can add to constructs in your {aws} CDK app. A tag applied to a given construct also applies to all of its taggable children. Tags are included in the {aws} CloudFormation template synthesized from your app and are applied to the {aws} resources it deploys. You can use tags to identify and categorize resources for the following purposes:

* Simplifying management
* Cost allocation
* Access control
* Any other purposes that you devise

= [TIP]

For more information about how you can use tags with your {aws} resources, see https://docs.aws.amazon.com/whitepapers/latest/tagging-best-practices/tagging-best-practices.html[Best Practices for Tagging {aws} Resources] in the *{aws} Whitepaper*.

====

[#tagging-use] == Using tags

The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tags.html[Tags] class includes the static method of(), through which you can add tags to, or remove tags from, the specified construct.

* https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tags.html#addkey-value-props[Tags.of(<SCOPE>).add()] applies a new tag to the given construct and all of its children.
* https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tags.html#removekey-props[Tags.of(<SCOPE>).remove()] removes a tag from the given construct and any of its children, including tags a child construct may have applied to itself.

= [NOTE]

Tagging is implemented using xref:aspects[Aspects and the {aws} CDK]. Aspects are a way to apply an operation (such as tagging) to all constructs in a given scope.

====

The following example applies the tag *key* with the value *value* to a construct.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘key’, ‘value’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘key’, ‘value’); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(my\_construct).add(“key”, “value”) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(“key”, “value”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(myConstruct).Add(“key”, “value”); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(myConstruct).Add(jsii.String(“key”), jsii.String(“value”), &awscdk.TagProps{}) — ====

The following example deletes the tag *key* from a construct.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).remove(‘key’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).remove(‘key’); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(my\_construct).remove(“key”) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of(myConstruct).remove(“key”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(myConstruct).Remove(“key”); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(myConstruct).Remove(jsii.String(“key”), &awscdk.TagProps{}) — ====

If you are using Stage constructs, apply the tag at the Stage level or below. Tags are not applied across Stage boundaries.

[#tagging-priorities] == Tag priorities

The {aws} CDK applies and removes tags recursively. If there are conflicts, the tagging operation with the highest priority wins. (Priorities are set using the optional priority property.) If the priorities of two operations are the same, the tagging operation closest to the bottom of the construct tree wins. By default, applying a tag has a priority of 100 (except for tags added directly to an {aws} CloudFormation resource, which has a priority of 50). The default priority for removing a tag is 200.

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘key’, ‘value’, { priority: 300 }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘key’, ‘value’, { priority: 300 }); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(my\_construct).add(“key”, “value”, priority=300) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(“key”, “value”, TagProps.builder() .priority(300).build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(myConstruct).Add(“key”, “value”, new TagProps { Priority = 300 }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(myConstruct).Add(jsii.String(“key”), jsii.String(“value”), &awscdk.TagProps{ Priority: jsii.Number(300), }) — ====

[#tagging-props] == Optional properties

Tags support https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.TagProps.html[properties] that fine-tune how tags are applied to, or removed from, resources. All properties are optional.

applyToLaunchedInstances (Python: apply\_to\_launched\_instances):: + Available for add() only. By default, tags are applied to instances launched in an Auto Scaling group. Set this property to *false* to ignore instances launched in an Auto Scaling group.

includeResourceTypes/excludeResourceTypes (Python: include\_resource\_types/exclude\_resource\_types):: + Use these to manipulate tags only on a subset of resources, based on {aws} CloudFormation resource types. By default, the operation is applied to all resources in the construct subtree, but this can be changed by including or excluding certain resource types. Exclude takes precedence over include, if both are specified.

priority:: + Use this to set the priority of this operation with respect to other Tags.add() and Tags.remove() operations. Higher values take precedence over lower values. The default is 100 for add operations (50 for tags applied directly to {aws} CloudFormation resources) and 200 for remove operations.

The following example applies the tag *tagname* with the value *value* and priority *100* to resources of type *{aws}::Xxx::Yyy* in the construct. It doesn’t apply the tag to instances launched in an Amazon EC2 Auto Scaling group or to resources of type *{aws}::Xxx::Zzz*. (These are placeholders for two arbitrary but different {aws} CloudFormation resource types.)

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘tagname’, ‘value’, { applyToLaunchedInstances: false, includeResourceTypes: [‘{aws}::Xxx::Yyy’], excludeResourceTypes: [‘{aws}::Xxx::Zzz’], priority: 100, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(‘tagname’, ‘value’, { applyToLaunchedInstances: false, includeResourceTypes: [‘{aws}::Xxx::Yyy’], excludeResourceTypes: [‘{aws}::Xxx::Zzz’], priority: 100 }); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(my\_construct).add(“tagname”, “value”, apply\_to\_launched\_instances=False, include\_resource\_types=[“{aws}::Xxx::Yyy”], exclude\_resource\_types=[“{aws}::Xxx::Zzz”], priority=100) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of(myConstruct).add(“tagname”, “value”, TagProps.builder() .applyToLaunchedInstances(false) .includeResourceTypes(Arrays.asList(“{aws}::Xxx::Yyy”)) .excludeResourceTypes(Arrays.asList(“{aws}::Xxx::Zzz”)) .priority(100).build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(myConstruct).Add(“tagname”, “value”, new TagProps { ApplyToLaunchedInstances = false, IncludeResourceTypes = [“{aws}::Xxx::Yyy”], ExcludeResourceTypes = [“{aws}::Xxx::Zzz”], Priority = 100 }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(myConstruct).Add(jsii.String(“tagname”), jsii.String(“value”), &awscdk.TagProps{ ApplyToLaunchedInstances: jsii.Bool(false), IncludeResourceTypes: &[]\_\_string{jsii.String(“{aws}::Xxx:Yyy”)}, ExcludeResourceTypes: &[]\_\_string{jsii.String(“{aws}::Xxx:Zzz”)}, Priority: jsii.Number(100), }) — ====

The following example removes the tag *tagname* with priority *200* from resources of type *{aws}::Xxx::Yyy* in the construct, but not from resources of type *{aws}::Xxx::Zzz*.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).remove(‘tagname’, { includeResourceTypes: [‘{aws}::Xxx::Yyy’], excludeResourceTypes: [‘{aws}::Xxx::Zzz’], priority: 200, }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(myConstruct).remove(‘tagname’, { includeResourceTypes: [‘{aws}::Xxx::Yyy’], excludeResourceTypes: [‘{aws}::Xxx::Zzz’], priority: 200 }); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(my\_construct).remove(“tagname”, include\_resource\_types=[“{aws}::Xxx::Yyy”], exclude\_resource\_types=[“{aws}::Xxx::Zzz”], priority=200,) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of((myConstruct).remove(“tagname”, TagProps.builder() .includeResourceTypes(Arrays.asList(“{aws}::Xxx::Yyy”)) .excludeResourceTypes(Arrays.asList(“{aws}::Xxx::Zzz”)) .priority(100).build()); ) —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(myConstruct).Remove(“tagname”, new TagProps { IncludeResourceTypes = [“{aws}::Xxx::Yyy”], ExcludeResourceTypes = [“{aws}::Xxx::Zzz”], Priority = 100 }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(myConstruct).Remove(jsii.String(“tagname”), &awscdk.TagProps{ IncludeResourceTypes: &[]\_\_string{jsii.String(“{aws}::Xxx:Yyy”)}, ExcludeResourceTypes: &[]\_\_string{jsii.String(“{aws}::Xxx:Zzz”)}, Priority: jsii.Number(200), }) — ====

[#tagging-example] == 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.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { App, Stack, Tags } from ‘aws-cdk-lib’;

const app = new App(); const theBestStack = new Stack(app, ‘MarketingSystem’);

// Add a tag to all constructs in the stack Tags.of(theBestStack).add(‘StackType’, ‘TheBest’);

// Remove the tag from all resources except subnet resources Tags.of(theBestStack).remove(‘StackType’, { excludeResourceTypes: [‘{aws}::EC2::Subnet’] }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { App, Stack, Tags } = require(‘aws-cdk-lib’);

const app = new App(); const theBestStack = new Stack(app, ‘MarketingSystem’);

// Add a tag to all constructs in the stack Tags.of(theBestStack).add(‘StackType’, ‘TheBest’);

// Remove the tag from all resources except subnet resources Tags.of(theBestStack).remove(‘StackType’, { excludeResourceTypes: [‘{aws}::EC2::Subnet’] }); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import App, Stack, Tags

app = App(); the\_best\_stack = Stack(app, ‘MarketingSystem’)

= Add a tag to all constructs in the stack

Tags.of(the\_best\_stack).add(“StackType”, “TheBest”)

= Remove the tag from all resources except subnet resources

Tags.of(the\_best\_stack).remove(“StackType”, exclude\_resource\_types=[“{aws}::EC2::Subnet”]) —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.App; import software.amazon.awscdk.Tags;

// Add a tag to all constructs in the stack Tags.of(theBestStack).add(“StackType”, “TheBest”);

// Remove the tag from all resources except subnet resources Tags.of(theBestStack).remove(“StackType”, TagProps.builder() .excludeResourceTypes(Arrays.asList(“{aws}::EC2::Subnet”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

var app = new App(); var theBestStack = new Stack(app, ‘MarketingSystem’);

// Add a tag to all constructs in the stack Tags.Of(theBestStack).Add(“StackType”, “TheBest”);

// Remove the tag from all resources except subnet resources Tags.Of(theBestStack).Remove(“StackType”, new TagProps { ExcludeResourceTypes = [“{aws}::EC2::Subnet”] }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import “github.com/aws/aws-cdk-go/awscdk/v2” app := awscdk.NewApp(nil) theBestStack := awscdk.NewStack(app, jsii.String(“MarketingSystem”), &awscdk.StackProps{})

// Add a tag to all constructs in the stack awscdk.Tags\_Of(theBestStack).Add(jsii.String(“StackType”), jsii.String(“TheBest”), &awscdk.TagProps{})

// Remove the tag from all resources except subnet resources awscdk.Tags\_Of(theBestStack).Add(jsii.String(“StackType”), jsii.String(“TheBest”), &awscdk.TagProps{ ExcludeResourceTypes: &[]\*string{jsii.String(“{aws}::EC2::Subnet”)}, }) — ====

The following code achieves the same result. Consider which approach (inclusion or exclusion) makes your intent clearer.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(theBestStack).add(‘StackType’, ‘TheBest’, { includeResourceTypes: [‘{aws}::EC2::Subnet’]}); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — Tags.of(theBestStack).add(‘StackType’, ‘TheBest’, { includeResourceTypes: [‘{aws}::EC2::Subnet’]}); —

Python:: + [source,python,subs=“verbatim,attributes”] — Tags.of(the\_best\_stack).add(“StackType”, “TheBest”, include\_resource\_types=[“{aws}::EC2::Subnet”]) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tags.of(theBestStack).add(“StackType”, “TheBest”, TagProps.builder() .includeResourceTypes(Arrays.asList(“{aws}::EC2::Subnet”)) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — Tags.Of(theBestStack).Add(“StackType”, “TheBest”, new TagProps { IncludeResourceTypes = [“{aws}::EC2::Subnet”] }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.Tags\_Of(theBestStack).Add(jsii.String(“StackType”), jsii.String(“TheBest”), &awscdk.TagProps{ IncludeResourceTypes: &[]\*string{jsii.String(“{aws}::EC2::Subnet”)}, }) — ====

[#tagging-single] == Tagging single constructs

Tags.of(scope).add(key, value) is the standard way to add tags to constructs in the {aws} CDK. Its tree-walking behavior, which recursively tags all taggable resources under the given scope, is almost always what you want. Sometimes, however, you need to tag a specific, arbitrary construct (or constructs).

One such case involves applying tags whose value is derived from some property of the construct being tagged. The standard tagging approach recursively applies the same key and value to all matching resources in the scope. However, here the value could be different for each tagged construct.

Tags are implemented using xref:aspects[aspects], and the CDK calls the tag’s visit() method for each construct under the scope you specified using Tags.of(scope). We can call Tag.visit() directly to apply a tag to a single construct.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new cdk.Tag(key, value).visit(scope); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new cdk.Tag(key, value).visit(scope); —

Python:: + [source,python,subs=“verbatim,attributes”] — cdk.Tag(key, value).visit(scope) —

Java:: + [source,java,subs=“verbatim,attributes”] — Tag.Builder.create(key, value).build().visit(scope); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new Tag(key, value).Visit(scope); —

Go:: + [source,go,subs=“verbatim,attributes”] — awscdk.NewTag(key, value, &awscdk.TagProps{}).Visit(scope) — ====

You can tag all constructs under a scope but let the values of the tags derive from properties of each construct. To do so, write an aspect and apply the tag in the aspect’s visit() method as shown in the preceding example. Then, add the aspect to the desired scope using Aspects.of(scope).add(aspect).

The following example applies a tag to each resource in a stack containing the resource’s path.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — class PathTagger implements cdk.IAspect { visit(node: IConstruct) { new cdk.Tag(“aws-cdk-path”, node.node.path).visit(node); } }

stack = new MyStack(app); cdk.Aspects.of(stack).add(new PathTagger()) —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class PathTagger { visit(node) { new cdk.Tag(“aws-cdk-path”, node.node.path).visit(node); } }

stack = new MyStack(app); cdk.Aspects.of(stack).add(new PathTagger()) —

Python:: + [source,python,subs=“verbatim,attributes”] — @jsii.implements(cdk.IAspect) class PathTagger: def visit(self, node: IConstruct): cdk.Tag(“aws-cdk-path”, node.node.path).visit(node)

stack = MyStack(app) cdk.Aspects.of(stack).add(PathTagger()) —

Java:: + [source,java,subs=“verbatim,attributes”] — final class PathTagger implements IAspect { public void visit(IConstruct node) { Tag.Builder.create(“aws-cdk-path”, node.getNode().getPath()).build().visit(node); } }

stack stack = new MyStack(app); Aspects.of(stack).add(new PathTagger()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — public class PathTagger : IAspect { public void Visit(IConstruct node) { new Tag(“aws-cdk-path”, node.Node.Path).Visit(node); } }

var stack = new MyStack(app); Aspects.Of(stack).Add(new PathTagger); — ====

= [TIP]

The logic of conditional tagging, including priorities, resource types, and so on, is built into the Tag class. You can use these features when applying tags to arbitrary resources; the tag is not applied if the conditions aren’t met. Also, the Tag class only tags taggable resources, so you don’t need to test whether a construct is taggable before applying a tag.

====

include::attributes.txt[]

// Attributes [.topic] [#testing-locally-build-with-sam-cli] = Building {aws} CDK applications with {aws} SAM :info\_titleabbrev: Building :keywords: {aws} SAM, build, {aws} CDK, Lambda

== [abstract]

## Use the sam build command to build {aws} CDK applications.

// Content start

The {aws} SAM CLI provides support for building Lambda functions and layers defined in your {aws} CDK application with +link:https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-build.html[sam build]+.

For Lambda functions that use zip artifacts, run cdk synth before you run sam local commands. sam build isn’t required.

If your {aws} CDK application uses functions with the image type, run cdk synth and then run sam build before you run sam local commands. When you run sam build, {aws} SAM doesn’t build Lambda functions or layers that use runtime-specific constructs, for example, +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_lambda\_nodejs.NodejsFunction.html[NodejsFunction]+. sam build doesn’t support https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.BundlingOptions.html[bundled assets].

[#testing-locally-build-with-sam-cli-examples] == Example

Running the following command from the {aws} CDK project root directory builds the application.

== [source,bash,subs=“verbatim,attributes”]

## $ sam build -t <./cdk.out/CdkSamExampleStack.template.json>

include::attributes.txt[]

// Attributes

[.topic] [#testing-locally-getting-started] = Getting started with locally testing :info\_titleabbrev: Getting started :keywords: CLI, {aws} CDK, {aws} SAM

== [abstract]

## Getting Started documentation that provides the prerequisites for using {aws} CDK with {aws} SAM, and tutorial for building, locally testing, and deploying a simple {aws} CDK application.

// Content start

This topic describes what you need to use the {aws} SAM CLI with {aws} CDK applications, and it provides instructions for building and locally testing a simple {aws} CDK application.

[#testing-locally-getting-started-prerequisites] == Prerequisites

To test locally, you must install the {aws} SAM CLI. see link:https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/getting\_started.html[Install the {aws} SAM CLI] for installation instructions.

[#testing-locally-getting-started-tutorial] == Creating and locally testing an {aws} CDK application

== [abstract]

## Learn how to use {aws} SAM to download, build, and locally test a {aws} CDK application.

To locally test an {aws} CDK application using the {aws} SAM CLI, you must have an {aws} CDK application that contains a Lambda function. Use the following steps to create a basic {aws} CDK application with a Lambda function. For more information, see link:https://docs.aws.amazon.com/cdk/latest/guide/serverless\_example.html[Creating a serverless application using the {aws} CDK] in the *{aws} Cloud Development Kit ({aws} CDK) Developer Guide*.

[#testing-locally-getting-started-tutorial-init] *Step 1: Create an {aws} CDK application*:: + For this tutorial, initialize an {aws} CDK application that uses TypeScript. + Command to run: + [source,none,subs=“verbatim,attributes”] — $ mkdir cdk-sam-example $ cd cdk-sam-example $ cdk init app –language typescript —

[#testing-locally-getting-started-tutorial-lambda] *Step 2: Add a Lambda function to your application*:: + Replace the code in lib/cdk-sam-example-stack.ts with the following: + [source,typescript,subs=“verbatim,attributes”] — import { Stack, StackProps } from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import \* as lambda from ‘aws-cdk-lib/aws-lambda’;

export class CdkSamExampleStack extends Stack { constructor(scope: Construct, id: string, props?: StackProps) { super(scope, id, props);

new lambda.Function(this, ‘MyFunction’, { runtime: lambda.Runtime.PYTHON\_3\_12, handler: ‘app.lambda\_handler’, code: lambda.Code.fromAsset(‘./my\_function’), }); } } —-

[#testing-locally-getting-started-tutorial-code] *Step 3: Add your Lambda function code*:: + Create a directory named my\_function. In that directory, create a file named app.py. + Command to run: + ==== [role=“tablist”] macOS / Linux:: + [source,none,subs=“verbatim,attributes”] — $ mkdir my\_function $ cd my\_function $ touch app.py —

Windows:: + [source,none,subs=“verbatim,attributes”] — $ mkdir my\_function $ cd my\_function $ type nul > app.py —

PowerShell:: + [source,none,subs=“verbatim,attributes”] — $ mkdir my\_function $ cd my\_function $ New-Item -Path “app.py" --- + Add the following code toapp.py`:

== [source,none,subs=“verbatim,attributes”]

def lambda\_handler(event, context): return “Hello from SAM and the CDK!” — ====

[#testing-locally-getting-started-tutorial-function] *Step 4: Test your Lambda function*:: + You can use the {aws} SAM CLI to locally invoke a Lambda function that you define in an {aws} CDK application. To do this, you need the function construct identifier and the path to your synthesized {aws} CloudFormation template. + Run the following command to go back to the lib directory: + [source,none,subs=“verbatim,attributes”] — $ cd .. — + *Command to run:* + [source,none,subs=“verbatim,attributes”] — $ cdk synth –no-staging — + [source,none,subs=“verbatim,attributes”] — $ sam local invoke MyFunction –no-event -t ./cdk.out/CdkSamExampleStack.template.json — + *Example output:* + — Invoking app.lambda\_handler (python3.9)

START RequestId: 5434c093-7182-4012-9b06-635011cac4f2 Version: $LATEST “Hello from SAM and the CDK!” END RequestId: 5434c093-7182-4012-9b06-635011cac4f2 REPORT RequestId: 5434c093-7182-4012-9b06-635011cac4f2 Init Duration: 0.32 ms Duration: 177.47 ms Billed Duration: 178 ms Memory Size: 128 MB Max Memory Used: 128 MB —

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#testing-locally-with-sam-cli] = Local testing {aws} CDK applications with {aws} SAM :info\_titleabbrev: Local testing :keywords: sam local, test, {aws} CDK, {aws} SAM

== [abstract]

## Use the sam local command to locally test {aws} CDK applications.

// Content start

You can use the {aws} SAM CLI to locally test your {aws} CDK applications by running the following commands from the project root directory of your {aws} CDK application:

* +link:https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-local-invoke.html[sam local invoke]+
* +link:https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-local-start-api.html[sam local start-api]+
* +link:https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-cli-command-reference-sam-local-start-lambda.html[sam local start-lambda]+

Before you run any of the sam local commands with a {aws} CDK application, you must run cdk synth.

When running sam local invoke you need the function construct identifier that you want to invoke, and the path to your synthesized {aws} CloudFormation template. If your application uses nested stacks, to resolve naming conflicts, you also need the stack name where the function is defined.

*Usage*:: + [source,none,subs=“verbatim,attributes”] —

= Invoke the function FUNCTION\_IDENTIFIER declared in the stack STACK\_NAME

$ sam local invoke ++++++++++++

= Start all APIs declared in the {aws} CDK application

$ sam local start-api -t <./cdk.out/CdkSamExampleStack.template.json> ++++++++++++

= Start a local endpoint that emulates {aws} Lambda

$ sam local start-lambda -t <./cdk.out/CdkSamExampleStack.template.json> ++++++—-++++++

[#testing-cdk-applications-examples] == Example

Consider stacks and functions that are declared with the following sample:

== [source,javascript,subs=“verbatim,attributes”]

app = new HelloCdkStack(app, “HelloCdkStack”, … ) class HelloCdkStack extends cdk.Stack { constructor(scope: Construct, id: string, props?: cdk.StackProps) { … new lambda.Function(this, ‘MyFunction’, { … });

new HelloCdkNestedStack(this, ‘HelloNestedStack’ ,{ … }); } }

class HelloCdkNestedStack extends cdk.NestedStack { constructor(scope: Construct, id: string, props?: cdk.NestedStackProps) { … new lambda.Function(this, ‘MyFunction’, { … }); new lambda.Function(this, ‘MyNestedFunction’, { … }); } } —

The following commands locally invokes the Lambda functions defined in example presented above:

== [source,none,subs=“verbatim,attributes”]

= Invoke MyFunction from the HelloCdkStack

$ sam local invoke -t <./cdk.out/HelloCdkStack.template.json> ++++++—-++++++

== [source,none,subs=“verbatim,attributes”]

= Invoke MyNestedFunction from the HelloCdkNestedStack

$ sam local invoke -t <./cdk.out/HelloCdkStack.template.json> ++++++—-++++++

== [source,none,subs=“verbatim,attributes”]

= Invoke MyFunction from the HelloCdkNestedStack

## $ sam local invoke -t <./cdk.out/HelloCdkStack.template.json>

include::attributes.txt[]

// Attributes

[.topic] [#testing-locally] = Locally test and build {aws} CDK applications with the {aws} SAM CLI :info\_titleabbrev: Local testing :keywords: build, test, {aws} CDK, {aws} SAM

== [abstract]

## You can use {aws} SAM to build and locally test your {aws} CDK applications.

// Content start

You can use the {aws} SAM CLI to locally test and build serverless applications defined using the {aws} Cloud Development Kit ({aws} CDK). Because the {aws} SAM CLI works within the {aws} CDK project structure, you can still use the xref:cli[{aws} CDK CLI reference] for creating, modifying, and deploying your {aws} CDK applications.

For details on using {aws} SAM, see https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started.html[Getting started with {aws} SAM] in the *{aws} Serverless Application Model Developer Guide*.

[.topiclist] [[Topic List]]

include::testing-locally-getting-started.adoc[leveloffset=+1]

include::testing-locally-with-sam-cli.adoc[leveloffset=+1]

include::testing-locally-build-with-sam-cli.adoc[leveloffset=+1]

:doctype: book

include::attributes.txt[]

// Attributes

:https–docs-aws-amazon-com-cdk-api-v2-docs-aws-cdk-lib-assertions-Match-html-methods: https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.assertions.Match.html#methods [.topic] [#testing] = Test {aws} CDK applications

// Content start

With the {aws} CDK, your infrastructure can be as testable as any other code you write. You can test in the cloud and locally. This topic addresses how to test in the cloud. For guidance on local testing see xref:testing-locally[Locally test and build {aws} CDK applications with the {aws} SAM CLI]. The standard approach to testing {aws} CDK apps uses the {aws} CDK’s https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.assertions-readme.html[assertions] module and popular test frameworks like https://jestjs.io/[Jest] for TypeScript and JavaScript or https://docs.pytest.org/en/6.2.x/[Pytest] for Python.

There are two categories of tests that you can write for {aws} CDK apps.

* *Fine-grained assertions* test specific aspects of the generated {aws} CloudFormation template, such as “this resource has this property with this value.” These tests can detect regressions. They’re also useful when you’re developing new features using test-driven development. (You can write a test first, then make it pass by writing a correct implementation.) Fine-grained assertions are the most frequently used tests.
* *Snapshot tests* test the synthesized {aws} CloudFormation template against a previously stored baseline template. Snapshot tests let you refactor freely, since you can be sure that the refactored code works exactly the same way as the original. If the changes were intentional, you can accept a new baseline for future tests. However, CDK upgrades can also cause synthesized templates to change, so you can’t rely only on snapshots to make sure that your implementation is correct.

= [NOTE]

Complete versions of the TypeScript, Python, and Java apps used as examples in this topic are https://github.com/cdklabs/aws-cdk-testing-examples/[available on GitHub].

====

[#testing-getting-started] == Getting started

To illustrate how to write these tests, we’ll create a stack that contains an {aws} Step Functions state machine and an {aws} Lambda function. The Lambda function is subscribed to an Amazon SNS topic and simply forwards the message to the state machine.

First, create an empty CDK application project using the CDK Toolkit and installing the libraries we’ll need. The constructs we’ll use are all in the main CDK package, which is a default dependency in projects created with the CDK Toolkit. However, you must install your testing framework.

==== [role=“tablist”] TypeScript:: + [source,none,subs=“verbatim,attributes”] — $ mkdir state-machine && cd state-machine cdk init –language=typescript npm install –save-dev jest @types/jest — + Create a directory for your tests. + [source,none,subs=“verbatim,attributes”] — $ mkdir test — + 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
* {blank}
* == Add a new jest top-level key with a moduleFileExtensions declaration
* These changes are shown in the following outline. Place the new text where indicated in package.json. The “…” placeholders indicate existing parts of the file that should not be changed. +

## [source,json,subs=“verbatim,attributes”]

* { … “scripts”: { … “test”: “jest” }, “devDependencies”: { … “@types/jest”: “{caret}24.0.18”, “jest”: “{caret}24.9.0” }, “jest”: { “moduleFileExtensions”: [“js”] } } —

JavaScript:: + [source,none,subs=“verbatim,attributes”] — $ mkdir state-machine && cd state-machine $ cdk init –language=javascript $ npm install –save-dev jest — + Create a directory for your tests. + [source,none,subs=“verbatim,attributes”] — $ mkdir test — + 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 to the devDependencies section
* {blank}
* == Add a new jest top-level key with a moduleFileExtensions declaration
* These changes are shown in the following outline. Place the new text where indicated in package.json. The “…” placeholders indicate existing parts of the file that shouldn’t be changed. +

## [source,json,subs=“verbatim,attributes”]

* { … “scripts”: { … “test”: “jest” }, “devDependencies”: { … “jest”: “{caret}24.9.0” }, “jest”: { “moduleFileExtensions”: [“js”] } } —

Python:: + [source,none,subs=“verbatim,attributes”] — $ mkdir state-machine && cd state-machine $ cdk init –language=python $ source .venv/bin/activate # On Windows, run ‘.’ instead $ python -m pip install -r requirements.txt $ python -m pip install -r requirements-dev.txt —

Java:: + [source,none,subs=“verbatim,attributes”] — $ mkdir state-machine && cd-state-machine $ cdk init –language=java — + Open the project in your preferred Java IDE. (In Eclipse, use *File* > *Import* > Existing Maven Projects.)

C#:: + [source,none,subs=“verbatim,attributes”] — $ mkdir state-machine && cd-state-machine $ cdk init –language=csharp — + Open src\StateMachine.sln in Visual Studio. + Right-click the solution in Solution Explorer and choose *Add* > *New Project*. Search for MSTest C# and add an *MSTest Test Project* for C#. (The default name TestProject1is fine.) + Right-click TestProject1 and choose *Add* > *Project Reference*, and add the StateMachine project as a reference. ====

[#testing-app] == The example stack

Here’s the stack that will be tested in this topic. As we’ve previously described, it contains a Lambda function and a Step Functions state machine, and accepts one or more Amazon SNS topics. The Lambda function is subscribed to the Amazon SNS topics and forwards them to the state machine.

You don’t have to do anything special to make the app testable. In fact, this CDK stack is not different in any important way from the other example stacks in this Guide.

==== [role=“tablist”] TypeScript:: + Place the following code in lib/state-machine-stack.ts: + + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from “aws-cdk-lib”; import \* as sns from “aws-cdk-lib/aws-sns”; import \* as sns\_subscriptions from “aws-cdk-lib/aws-sns-subscriptions”; import \* as lambda from “aws-cdk-lib/aws-lambda”; import \* as sfn from “aws-cdk-lib/aws-stepfunctions”; import { Construct } from “constructs”;

export interface StateMachineStackProps extends cdk.StackProps { readonly topics: sns.Topic[]; }

export class StateMachineStack extends cdk.Stack { constructor(scope: Construct, id: string, props: StateMachineStackProps) { super(scope, id, props);

…. // In the future this state machine will do some work… const stateMachine = new sfn.StateMachine(this, “StateMachine”, { definition: new sfn.Pass(this, “StartState”), });

// This Lambda function starts the state machine. const func = new lambda.Function(this, “LambdaFunction”, { runtime: lambda.Runtime.NODEJS\_18\_X, handler: “handler”, code: lambda.Code.fromAsset(“./start-state-machine”), environment: { STATE\_MACHINE\_ARN: stateMachine.stateMachineArn, }, }); stateMachine.grantStartExecution(func);

const subscription = new sns\_subscriptions.LambdaSubscription(func); for (const topic of props.topics) { topic.addSubscription(subscription); } } } —- ….

JavaScript:: + Place the following code in lib/state-machine-stack.js: + + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(“aws-cdk-lib”); const sns = require(“aws-cdk-lib/aws-sns”); const sns\_subscriptions = require(“aws-cdk-lib/aws-sns-subscriptions”); const lambda = require(“aws-cdk-lib/aws-lambda”); const sfn = require(“aws-cdk-lib/aws-stepfunctions”);

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

…. // In the future this state machine will do some work… const stateMachine = new sfn.StateMachine(this, “StateMachine”, { definition: new sfn.Pass(this, “StartState”), });

// This Lambda function starts the state machine. const func = new lambda.Function(this, “LambdaFunction”, { runtime: lambda.Runtime.NODEJS\_18\_X, handler: “handler”, code: lambda.Code.fromAsset(“./start-state-machine”), environment: { STATE\_MACHINE\_ARN: stateMachine.stateMachineArn, }, }); stateMachine.grantStartExecution(func);

const subscription = new sns\_subscriptions.LambdaSubscription(func); for (const topic of props.topics) { topic.addSubscription(subscription); } } } ….

== module.exports = { StateMachineStack }

Python:: + Place the following code in state\_machine/state\_machine\_stack.py: + [source,python,subs=“verbatim,attributes”] — from typing import List

import aws\_cdk.aws\_lambda as lambda\_ import aws\_cdk.aws\_sns as sns import aws\_cdk.aws\_sns\_subscriptions as sns\_subscriptions import aws\_cdk.aws\_stepfunctions as sfn import aws\_cdk as cdk

class StateMachineStack(cdk.Stack): def *init*( self, scope: cdk.Construct, construct\_id: str, \*, topics: List[sns.Topic], **kwargs ) -> None: super().*init*(scope, construct\_id,** kwargs)

…. # In the future this state machine will do some work… state\_machine = sfn.StateMachine( self, “StateMachine”, definition=sfn.Pass(self, “StartState”) )

# This Lambda function starts the state machine.  
func = lambda\_.Function(  
 self,  
 "LambdaFunction",  
 runtime=lambda\_.Runtime.NODEJS\_18\_X,  
 handler="handler",  
 code=lambda\_.Code.from\_asset("./start-state-machine"),  
 environment={  
 "STATE\_MACHINE\_ARN": state\_machine.state\_machine\_arn,  
 },  
)  
state\_machine.grant\_start\_execution(func)  
  
subscription = sns\_subscriptions.LambdaSubscription(func)  
for topic in topics:  
 topic.add\_subscription(subscription) ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — package software.amazon.samples.awscdkassertionssamples;

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; 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.sns.ITopicSubscription; import software.amazon.awscdk.services.sns.Topic; import software.amazon.awscdk.services.sns.subscriptions.LambdaSubscription; import software.amazon.awscdk.services.stepfunctions.Pass; import software.amazon.awscdk.services.stepfunctions.StateMachine;

import java.util.Collections; import java.util.List;

public class StateMachineStack extends Stack { public StateMachineStack(final Construct scope, final String id, final List++++++topics) { this(scope, id, null, topics); }++++++

…. public StateMachineStack(final Construct scope, final String id, final StackProps props, final List topics) { super(scope, id, props);

// In the future this state machine will do some work...  
final StateMachine stateMachine = StateMachine.Builder.create(this, "StateMachine")  
 .definition(new Pass(this, "StartState"))  
 .build();  
  
// This Lambda function starts the state machine.  
final Function func = Function.Builder.create(this, "LambdaFunction")  
 .runtime(Runtime.NODEJS\_18\_X)  
 .handler("handler")  
 .code(Code.fromAsset("./start-state-machine"))  
 .environment(Collections.singletonMap("STATE\_MACHINE\_ARN", stateMachine.getStateMachineArn()))  
 .build();  
stateMachine.grantStartExecution(func);  
  
final ITopicSubscription subscription = new LambdaSubscription(func);  
for (final Topic topic : topics) {  
 topic.addSubscription(subscription);  
}

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.Lambda; using Amazon.CDK.{aws}.StepFunctions; using Amazon.CDK.{aws}.SNS; using Amazon.CDK.{aws}.SNS.Subscriptions; using Constructs;

using System.Collections.Generic;

namespace AwsCdkAssertionSamples { public class StateMachineStackProps : StackProps { public Topic[] Topics; }

…. public class StateMachineStack : Stack {

internal StateMachineStack(Construct scope, string id, StateMachineStackProps props = null) : base(scope, id, props)  
{  
 // In the future this state machine will do some work...  
 var stateMachine = new StateMachine(this, "StateMachine", new StateMachineProps  
 {  
 Definition = new Pass(this, "StartState")  
 });  
  
 // This Lambda function starts the state machine.  
 var func = new Function(this, "LambdaFunction", new FunctionProps  
 {  
 Runtime = Runtime.NODEJS\_18\_X,  
 Handler = "handler",  
 Code = Code.FromAsset("./start-state-machine"),  
 Environment = new Dictionary<string, string>  
 {  
 { "STATE\_MACHINE\_ARN", stateMachine.StateMachineArn }  
 }  
 });  
 stateMachine.GrantStartExecution(func);  
  
 foreach (Topic topic in props?.Topics ?? new Topic[0])  
 {  
 var subscription = new LambdaSubscription(func);  
 }  
  
}

} } —- ==== ….

We’ll modify the app’s main entry point so that we don’t actually instantiate our stack. We don’t want to accidentally deploy it. Our tests will create an app and an instance of the stack for testing. This is a useful tactic when combined with test-driven development: make sure that the stack passes all tests before you enable deployment.

==== [role=“tablist”] TypeScript:: + In bin/state-machine.ts: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node import \* as cdk from “aws-cdk-lib”;

const app = new cdk.App();

// Stacks are intentionally not created here – this application isn’t meant to // be deployed. —

JavaScript:: + In bin/state-machine.js: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node const cdk = require(“aws-cdk-lib”);

const app = new cdk.App();

// Stacks are intentionally not created here – this application isn’t meant to // be deployed. —

Python:: + In app.py: + [source,python,subs=“verbatim,attributes”] — #!/usr/bin/env python3 import os

import aws\_cdk as cdk

app = cdk.App()

= Stacks are intentionally not created here – this application isn’t meant to

= be deployed.

== app.synth()

Java:: + [source,java,subs=“verbatim,attributes”] — package software.amazon.samples.awscdkassertionssamples;

import software.amazon.awscdk.App;

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

…. // Stacks are intentionally not created here – this application isn’t meant to be deployed.

app.synth();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

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

…. // Stacks are intentionally not created here – this application isn’t meant to be deployed.

app.Synth();  
}

} } —- ==== ….

[#testing-lambda] == The Lambda function

Our example stack includes a Lambda function that starts our state machine. We must provide the source code for this function so the CDK can bundle and deploy it as part of creating the Lambda function resource.

* Create the folder start-state-machine in the app’s main directory.
* In this folder, create at least one file. For example, you can save the following code in start-state-machines/index.js.

## [source,javascript,subs=“verbatim,attributes”]

* exports.handler = async function (event, context) { return ‘hello world’; }; —
* However, any file will work, since we won’t actually be deploying the stack.

[#testing-running-tests] == Running tests

For reference, here are the commands you use to run tests in your {aws} CDK app. These are the same commands that you’d use to run the tests in any project using the same testing framework. For languages that require a build step, include that to make sure that your tests have compiled.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — $ tsc && npm test —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — $ npm test —

Python:: + [source,none,subs=“verbatim,attributes”] — $ python -m pytest —

Java:: + [source,none,subs=“verbatim,attributes”] — $ mvn compile && mvn test —

C#:: + Build your solution (F6) to discover the tests, then run the tests (*Test* > *Run All Tests*). To choose which tests to run, open Test Explorer (*Test* > *Test Explorer*). + Or: + [source,none,subs=“verbatim,attributes”] — $ dotnet test src — ====

[#testing-fine-grained] == Fine-grained assertions

The first step for testing a stack with fine-grained assertions is to synthesize the stack, because we’re writing assertions against the generated {aws} CloudFormation template.

Our StateMachineStackStack requires that we pass it the Amazon SNS topic to be forwarded to the state machine. So in our test, we’ll create a separate stack to contain the topic.

Ordinarily, when writing a CDK app, you can subclass Stack and instantiate the Amazon SNS topic in the stack’s constructor. In our test, we instantiate Stack directly, then pass this stack as the Topic’s scope, attaching it to the stack. This is functionally equivalent and less verbose. It also helps make stacks that are used only in tests “look different” from the stacks that you intend to deploy.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Capture, Match, Template } from “aws-cdk-lib/assertions”; import \* as cdk from “aws-cdk-lib”; import \* as sns from “aws-cdk-lib/aws-sns”; import { StateMachineStack } from “../lib/state-machine-stack”;

describe(“StateMachineStack”, () => { test(“synthesizes the way we expect”, () => { const app = new cdk.App();

…. // Since the StateMachineStack consumes resources from a separate stack // (cross-stack references), we create a stack for our SNS topics to live // in here. These topics can then be passed to the StateMachineStack later, // creating a cross-stack reference. const topicsStack = new cdk.Stack(app, “TopicsStack”);

// Create the topic the stack we’re testing will reference. const topics = [new sns.Topic(topicsStack, “Topic1”, {})];

// Create the StateMachineStack. const stateMachineStack = new StateMachineStack(app, “StateMachineStack”, { topics: topics, // Cross-stack reference });

// Prepare the stack for assertions. const template = Template.fromStack(stateMachineStack); }) }) —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Capture, Match, Template } = require(“aws-cdk-lib/assertions”); const cdk = require(“aws-cdk-lib”); const sns = require(“aws-cdk-lib/aws-sns”); const { StateMachineStack } = require(“../lib/state-machine-stack”);

describe(“StateMachineStack”, () => { test(“synthesizes the way we expect”, () => { const app = new cdk.App();

…. // Since the StateMachineStack consumes resources from a separate stack // (cross-stack references), we create a stack for our SNS topics to live // in here. These topics can then be passed to the StateMachineStack later, // creating a cross-stack reference. const topicsStack = new cdk.Stack(app, “TopicsStack”);

// Create the topic the stack we’re testing will reference. const topics = [new sns.Topic(topicsStack, “Topic1”, {})];

// Create the StateMachineStack. const StateMachineStack = new StateMachineStack(app, “StateMachineStack”, { topics: topics, // Cross-stack reference });

// Prepare the stack for assertions. const template = Template.fromStack(stateMachineStack); }) }) —- ….

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import aws\_sns as sns import aws\_cdk as cdk from aws\_cdk.assertions import Template

from app.state\_machine\_stack import StateMachineStack

def test\_synthesizes\_properly(): app = cdk.App()

…. # Since the StateMachineStack consumes resources from a separate stack # (cross-stack references), we create a stack for our SNS topics to live # in here. These topics can then be passed to the StateMachineStack later, # creating a cross-stack reference. topics\_stack = cdk.Stack(app, “TopicsStack”)

# Create the topic the stack we’re testing will reference.

topics = [sns.Topic(topics\_stack, “Topic1”)]

# Create the StateMachineStack.

state\_machine\_stack = StateMachineStack( app, “StateMachineStack”, topics=topics # Cross-stack reference )

# Prepare the stack for assertions.

template = Template.from\_stack(state\_machine\_stack) —- ….

Java:: + [source,java,subs=“verbatim,attributes”] — package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test; import software.amazon.awscdk.assertions.Capture; import software.amazon.awscdk.assertions.Match; import software.amazon.awscdk.assertions.Template; import software.amazon.awscdk.App; import software.amazon.awscdk.Stack; import software.amazon.awscdk.services.sns.Topic;

import java.util.\*;

import static org.assertj.core.api.Assertions.assertThat;

public class StateMachineStackTest { @Test public void testSynthesizesProperly() { final App app = new App();

…. // Since the StateMachineStack consumes resources from a separate stack (cross-stack references), we create a stack // for our SNS topics to live in here. These topics can then be passed to the StateMachineStack later, creating a // cross-stack reference. final Stack topicsStack = new Stack(app, “TopicsStack”);

// Create the topic the stack we're testing will reference.  
final List<Topic> topics = Collections.singletonList(Topic.Builder.create(topicsStack, "Topic1").build());  
  
// Create the StateMachineStack.  
final StateMachineStack stateMachineStack = new StateMachineStack(  
 app,  
 "StateMachineStack",  
 topics // Cross-stack reference  
);  
  
// Prepare the stack for assertions.  
final Template template = Template.fromStack(stateMachineStack)

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK; using Amazon.CDK.{aws}.SNS; using Amazon.CDK.Assertions; using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>; using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1 { [TestClass] public class StateMachineStackTest { [TestMethod] public void TestMethod1() { var app = new App();

…. // Since the StateMachineStack consumes resources from a separate stack (cross-stack references), we create a stack // for our SNS topics to live in here. These topics can then be passed to the StateMachineStack later, creating a // cross-stack reference. var topicsStack = new Stack(app, “TopicsStack”);

// Create the topic the stack we're testing will reference.  
 var topics = new Topic[] { new Topic(topicsStack, "Topic1") };  
  
 // Create the StateMachineStack.  
 var StateMachineStack = new StateMachineStack(app, "StateMachineStack", new StateMachineStackProps  
 {  
 Topics = topics  
 });  
  
 // Prepare the stack for assertions.  
 var template = Template.FromStack(stateMachineStack);  
  
 // test will go here  
}

} } —- ==== ….

Now we can assert that the Lambda function and the Amazon SNS subscription were created.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Assert it creates the function with the correct properties… template.hasResourceProperties(“{aws}::Lambda::Function”, { Handler: “handler”, Runtime: “nodejs14.x”, });

// Creates the subscription… template.resourceCountIs(“{aws}::SNS::Subscription”, 1); —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Assert it creates the function with the correct properties… template.hasResourceProperties(“{aws}::Lambda::Function”, { Handler: “handler”, Runtime: “nodejs14.x”, });

// Creates the subscription… template.resourceCountIs(“{aws}::SNS::Subscription”, 1); —-

Python:: + [source,python,subs=“verbatim,attributes”] —

= Assert that we have created the function with the correct properties

…. template.has\_resource\_properties( “{aws}::Lambda::Function”, { “Handler”: “handler”, “Runtime”: “nodejs14.x”, }, )

# Assert that we have created a subscription

template.resource\_count\_is(“{aws}::SNS::Subscription”, 1) —- ….

Java:: + [source,java,subs=“verbatim,attributes”] — // Assert it creates the function with the correct properties… template.hasResourceProperties(“{aws}::Lambda::Function”, Map.of( “Handler”, “handler”, “Runtime”, “nodejs14.x” ));

// Creates the subscription...  
 template.resourceCountIs("{aws}::SNS::Subscription", 1); ----

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Prepare the stack for assertions. var template = Template.FromStack(stateMachineStack);

…. // Assert it creates the function with the correct properties… template.HasResourceProperties(“{aws}::Lambda::Function”, new StringDict { { “Handler”, “handler”}, { “Runtime”, “nodejs14x” } });

// Creates the subscription...  
 template.ResourceCountIs("{aws}::SNS::Subscription", 1); ---- ====

….

Our Lambda function test asserts that two particular properties of the function resource have specific values. By default, the hasResourceProperties method performs a partial match on the resource’s properties as given in the synthesized CloudFormation template. This test requires that the provided properties exist and have the specified values, but the resource can also have other properties, which are not tested.

Our Amazon SNS assertion asserts that the synthesized template contains a subscription, but nothing about the subscription itself. We included this assertion mainly to illustrate how to assert on resource counts. The Template class offers more specific methods to write assertions against the Resources, Outputs, and Mapping sections of the CloudFormation template.

[#testing-fine-grained-matchers] *Matchers*:: + The default partial matching behavior of hasResourceProperties can be changed using *matchers* from the +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.assertions.Match.html#methods[Match]+ class. + Matchers range from lenient (Match.anyValue) to strict (Match.objectEquals). They can be nested to apply different matching methods to different parts of the resource properties. Using Match.objectEquals and Match.anyValue together, for example, we can test the state machine’s IAM role more fully, while not requiring specific values for properties that may change. + ==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Fully assert on the state machine’s IAM role with matchers. template.hasResourceProperties( “{aws}::IAM::Role”, Match.objectEquals({ AssumeRolePolicyDocument: { Version: “2012-10-17”, Statement: [ { Action: “sts:AssumeRole”, Effect: “Allow”, Principal: { Service: { “Fn::Join”: [ ““, [“states.”, Match.anyValue(), “.amazonaws.com”], ], }, }, }, ], }, }) ); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Fully assert on the state machine’s IAM role with matchers. template.hasResourceProperties( “{aws}::IAM::Role”, Match.objectEquals({ AssumeRolePolicyDocument: { Version: “2012-10-17”, Statement: [ { Action: “sts:AssumeRole”, Effect: “Allow”, Principal: { Service: { “Fn::Join”: [ ““, [“states.”, Match.anyValue(), “.amazonaws.com”], ], }, }, }, ], }, }) ); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk.assertions import Match

# Fully assert on the state machine’s IAM role with matchers. template.has\_resource\_properties( “{aws}::IAM::Role”, Match.object\_equals( { “AssumeRolePolicyDocument”: { “Version”: “2012-10-17”, “Statement”: [ { “Action”: “sts:AssumeRole”, “Effect”: “Allow”, “Principal”: { “Service”: { “Fn::Join”: [ ““, [ “states.”, Match.any\_value(), “.amazonaws.com”, ], ], }, }, }, ], }, } ), ) —-

Java:: + [source,java,subs=“verbatim,attributes”] — // Fully assert on the state machine’s IAM role with matchers. template.hasResourceProperties(“{aws}::IAM::Role”, Match.objectEquals( Collections.singletonMap(“AssumeRolePolicyDocument”, Map.of( “Version”, “2012-10-17”, “Statement”, Collections.singletonList(Map.of( “Action”, “sts:AssumeRole”, “Effect”, “Allow”, “Principal”, Collections.singletonMap( “Service”, Collections.singletonMap( “Fn::Join”, Arrays.asList( ““, Arrays.asList(”states.”, Match.anyValue(), “.amazonaws.com”) ) ) ) )) )) )); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Fully assert on the state machine’s IAM role with matchers. template.HasResource(“{aws}::IAM::Role”, Match.ObjectEquals(new ObjectDict { { “AssumeRolePolicyDocument”, new ObjectDict { { “Version”, “2012-10-17” }, { “Action”, “sts:AssumeRole” }, { “Principal”, new ObjectDict { { “Version”, “2012-10-17” }, { “Statement”, new object[] { new ObjectDict { { “Action”, “sts:AssumeRole” }, { “Effect”, “Allow” }, { “Principal”, new ObjectDict { { “Service”, new ObjectDict { { ““, new object[] {”states”, Match.AnyValue(), “.amazonaws.com” } } } } } } } } } } } } } })); — ==== + Many CloudFormation resources include serialized JSON objects represented as strings. The Match.serializedJson() matcher can be used to match properties inside this JSON. + For example, Step Functions state machines are defined using a string in the JSON-based https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html[Amazon States Language]. We’ll use Match.serializedJson() to make sure that our initial state is the only step. Again, we’ll use nested matchers to apply different kinds of matching to different parts of the object. + ==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Assert on the state machine’s definition with the Match.serializedJson() // matcher. template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, { DefinitionString: Match.serializedJson( // Match.objectEquals() is used implicitly, but we use it explicitly // here for extra clarity. Match.objectEquals({ StartAt: “StartState”, States: { StartState: { Type: “Pass”, End: true, // Make sure this state doesn’t provide a next state – we can’t // provide both Next and set End to true. Next: Match.absent(), }, }, }) ), }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Assert on the state machine’s definition with the Match.serializedJson() // matcher. template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, { DefinitionString: Match.serializedJson( // Match.objectEquals() is used implicitly, but we use it explicitly // here for extra clarity. Match.objectEquals({ StartAt: “StartState”, States: { StartState: { Type: “Pass”, End: true, // Make sure this state doesn’t provide a next state – we can’t // provide both Next and set End to true. Next: Match.absent(), }, }, }) ), }); —

Python:: + [source,python,subs=“verbatim,attributes”] — # Assert on the state machine’s definition with the serialized\_json matcher. template.has\_resource\_properties( “{aws}::StepFunctions::StateMachine”, { “DefinitionString”: Match.serialized\_json( # Match.object\_equals() is the default, but specify it here for clarity Match.object\_equals( { “StartAt”: “StartState”, “States”: { “StartState”: { “Type”: “Pass”, “End”: True, # Make sure this state doesn’t provide a next state – # we can’t provide both Next and set End to true. “Next”: Match.absent(), }, }, } ) ), }, ) —

Java:: + [source,java,subs=“verbatim,attributes”] — // Assert on the state machine’s definition with the Match.serializedJson() matcher. template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, Collections.singletonMap( “DefinitionString”, Match.serializedJson( // Match.objectEquals() is used implicitly, but we use it explicitly here for extra clarity. Match.objectEquals(Map.of( “StartAt”, “StartState”, “States”, Collections.singletonMap( “StartState”, Map.of( “Type”, “Pass”, “End”, true, // Make sure this state doesn’t provide a next state – we can’t provide // both Next and set End to true. “Next”, Match.absent() ) ) )) ) )); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Assert on the state machine’s definition with the Match.serializedJson() matcher template.HasResourceProperties(“{aws}::StepFunctions::StateMachine”, new ObjectDict { { “DefinitionString”, Match.SerializedJson( // Match.objectEquals() is used implicitly, but we use it explicitly here for extra clarity. Match.ObjectEquals(new ObjectDict { { “StartAt”, “StartState” }, { “States”, new ObjectDict { { “StartState”, new ObjectDict { { “Type”, “Pass” }, { “End”, “True” }, // Make sure this state doesn’t provide a next state – we can’t provide // both Next and set End to true. { “Next”, Match.Absent() } }} }} }) + )}}); — ====

[#testing-fine-grained-capture] *Capturing*:: + It’s often useful to test properties to make sure they follow specific formats, or have the same value as another property, without needing to know their exact values ahead of time. The assertions module provides this capability in its +link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.assertions.Capture.html[Capture]+ class. + By specifying a Capture instance in place of a value in hasResourceProperties, that value is retained in the Capture object. The actual captured value can be retrieved using the object’s as methods, including asNumber(), asString(), and asObject, and subjected to test. Use Capture with a matcher to specify the exact location of the value to be captured within the resource’s properties, including serialized JSON properties. + The following example tests to make sure that the starting state of our state machine has a name beginning with Start. It also tests that this state is present within the list of states in the machine. + ==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // Capture some data from the state machine’s definition. const startAtCapture = new Capture(); const statesCapture = new Capture(); template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, { DefinitionString: Match.serializedJson( Match.objectLike({ StartAt: startAtCapture, States: statesCapture, }) ), });

…. // Assert that the start state starts with “Start”. expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the // state machine definition. expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString()); —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — // Capture some data from the state machine’s definition. const startAtCapture = new Capture(); const statesCapture = new Capture(); template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, { DefinitionString: Match.serializedJson( Match.objectLike({ StartAt: startAtCapture, States: statesCapture, }) ), });

…. // Assert that the start state starts with “Start”. expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the // state machine definition. expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString()); —- ….

Python:: + [source,python,subs=“verbatim,attributes”] — import re

…. from aws\_cdk.assertions import Capture

# …

# Capture some data from the state machine’s definition.

start\_at\_capture = Capture() states\_capture = Capture() template.has\_resource\_properties( “{aws}::StepFunctions::StateMachine”, { “DefinitionString”: Match.serialized\_json( Match.object\_like( { “StartAt”: start\_at\_capture, “States”: states\_capture, } ) ), }, )

# Assert that the start state starts with “Start”.

assert re.match(“^Start”, start\_at\_capture.as\_string())

# Assert that the start state actually exists in the states object of the

# state machine definition.

assert start\_at\_capture.as\_string() in states\_capture.as\_object() —- ….

Java:: + [source,java,subs=“verbatim,attributes”] — // Capture some data from the state machine’s definition. final Capture startAtCapture = new Capture(); final Capture statesCapture = new Capture(); template.hasResourceProperties(“{aws}::StepFunctions::StateMachine”, Collections.singletonMap( “DefinitionString”, Match.serializedJson( Match.objectLike(Map.of( “StartAt”, startAtCapture, “States”, statesCapture )) ) ));

…. // Assert that the start state starts with “Start”. assertThat(startAtCapture.asString()).matches(“^Start.+”);

// Assert that the start state actually exists in the states object of the state machine definition.  
assertThat(statesCapture.asObject()).containsKey(startAtCapture.asString()); ----

….

C#:: + [source,csharp,subs=“verbatim,attributes”] — // Capture some data from the state machine’s definition. var startAtCapture = new Capture(); var statesCapture = new Capture(); template.HasResourceProperties(“{aws}::StepFunctions::StateMachine”, new ObjectDict { { “DefinitionString”, Match.SerializedJson( new ObjectDict { { “StartAt”, startAtCapture }, { “States”, statesCapture } } )} });

Assert.IsTrue(startAtCapture.ToString().StartsWith("Start"));  
 Assert.IsTrue(statesCapture.AsObject().ContainsKey(startAtCapture.ToString())); ---- ====

[#testing-snapshot] == Snapshot tests

In *snapshot testing*, you compare the entire synthesized CloudFormation template against a previously stored baseline (often called a “master”) template. Unlike fine-grained assertions, snapshot testing isn’t useful in catching regressions. This is because snapshot testing applies to the entire template, and things besides code changes can cause small (or not-so-small) differences in synthesis results. These changes may not even affect your deployment, but they will still cause a snapshot test to fail.

For example, you might update a CDK construct to incorporate a new best practice, which can cause changes to the synthesized resources or how they’re organized. Alternatively, you might update the CDK Toolkit to a version that reports additional metadata. Changes to context values can also affect the synthesized template.

Snapshot tests can be of great help in refactoring, though, as long as you hold constant all other factors that might affect the synthesized template. You will know immediately if a change you made has unintentionally changed the template. If the change is intentional, simply accept the new template as the baseline.

For example, if we have this DeadLetterQueue construct:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — export class DeadLetterQueue extends sqs.Queue { public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

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

// Add the alarm this.messagesInQueueAlarm = new cloudwatch.Alarm(this, ‘Alarm’, { alarmDescription: ‘There are messages in the Dead Letter Queue’, evaluationPeriods: 1, threshold: 1, metric: this.metricApproximateNumberOfMessagesVisible(), }); } } —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class DeadLetterQueue extends sqs.Queue {

constructor(scope, id) { super(scope, id);

// Add the alarm this.messagesInQueueAlarm = new cloudwatch.Alarm(this, ‘Alarm’, { alarmDescription: ‘There are messages in the Dead Letter Queue’, evaluationPeriods: 1, threshold: 1, metric: this.metricApproximateNumberOfMessagesVisible(), }); } }

== module.exports = { DeadLetterQueue }

Python:: + [source,python,subs=“verbatim,attributes”] — class DeadLetterQueue(sqs.Queue): def *init*(self, scope: Construct, id: str): super().*init*(scope, id)

self.messages\_in\_queue\_alarm = cloudwatch.Alarm(  
 self,  
 "Alarm",  
 alarm\_description="There are messages in the Dead Letter Queue.",  
 evaluation\_periods=1,  
 threshold=1,  
 metric=self.metric\_approximate\_number\_of\_messages\_visible(),  
 ) ----

Java:: + [source,java,subs=“verbatim,attributes”] — public class DeadLetterQueue extends Queue { private final IAlarm messagesInQueueAlarm;

…. public DeadLetterQueue(@NotNull Construct scope, @NotNull String id) { super(scope, id);

this.messagesInQueueAlarm = Alarm.Builder.create(this, "Alarm")  
 .alarmDescription("There are messages in the Dead Letter Queue.")  
 .evaluationPeriods(1)  
 .threshold(1)  
 .metric(this.metricApproximateNumberOfMessagesVisible())  
 .build();

}

public IAlarm getMessagesInQueueAlarm() { return messagesInQueueAlarm; } } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — namespace AwsCdkAssertionSamples { public class DeadLetterQueue : Queue { public IAlarm messagesInQueueAlarm;

public DeadLetterQueue(Construct scope, string id) : base(scope, id)  
 {  
 messagesInQueueAlarm = new Alarm(this, "Alarm", new AlarmProps  
 {  
 AlarmDescription = "There are messages in the Dead Letter Queue.",  
 EvaluationPeriods = 1,  
 Threshold = 1,  
 Metric = this.MetricApproximateNumberOfMessagesVisible()  
 });  
 }

} } —- ====

We can test it like this:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Match, Template } from “aws-cdk-lib/assertions”; import \* as cdk from “aws-cdk-lib”; import { DeadLetterQueue } from “../lib/dead-letter-queue”;

describe(“DeadLetterQueue”, () => { test(“matches the snapshot”, () => { const stack = new cdk.Stack(); new DeadLetterQueue(stack, “DeadLetterQueue”);

const template = Template.fromStack(stack); expect(template.toJSON()).toMatchSnapshot(); }); }); —-

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Match, Template } = require(“aws-cdk-lib/assertions”); const cdk = require(“aws-cdk-lib”); const { DeadLetterQueue } = require(“../lib/dead-letter-queue”);

describe(“DeadLetterQueue”, () => { test(“matches the snapshot”, () => { const stack = new cdk.Stack(); new DeadLetterQueue(stack, “DeadLetterQueue”);

const template = Template.fromStack(stack); expect(template.toJSON()).toMatchSnapshot(); }); }); —-

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk\_lib as cdk from aws\_cdk\_lib.assertions import Match, Template

from app.dead\_letter\_queue import DeadLetterQueue

def snapshot\_test(): stack = cdk.Stack() DeadLetterQueue(stack, “DeadLetterQueue”)

template = Template.from\_stack(stack) assert template.to\_json() == snapshot —-

Java:: + [source,java,subs=“verbatim,attributes”] — package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test; import au.com.origin.snapshots.Expect; import software.amazon.awscdk.assertions.Match; import software.amazon.awscdk.assertions.Template; import software.amazon.awscdk.Stack;

import java.util.Collections; import java.util.Map;

public class DeadLetterQueueTest { @Test public void snapshotTest() { final Stack stack = new Stack(); new DeadLetterQueue(stack, “DeadLetterQueue”);

final Template template = Template.fromStack(stack);  
 expect.toMatchSnapshot(template.toJSON());

} } —-

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK; using Amazon.CDK.Assertions; using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>; using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1 { [TestClass] public class StateMachineStackTest

…. [TestClass] public class DeadLetterQueueTest { [TestMethod] public void SnapshotTest() { var stack = new Stack(); new DeadLetterQueue(stack, “DeadLetterQueue”);

var template = Template.FromStack(stack);  
  
 return Verifier.Verify(template.ToJSON());  
}

} } —- ==== ….

[#testing-tips] == Tips for tests

Remember, your tests will live just as long as the code they test, and they will be read and modified just as often. Therefore, it pays to take a moment to consider how best to write them.

Don’t copy and paste setup lines or common assertions. Instead, refactor this logic into fixtures or helper functions. Use good names that reflect what each test actually tests.

Don’t try to do 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 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.

include::testing-locally.adoc[leveloffset=+1]

:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#versioning] = {aws} CDK versioning :info\_titleabbrev: Versioning :keywords: versioning, {aws} CDK reference

== [abstract]

## This topic provides reference information on how the {aws} Cloud Development Kit ({aws} CDK) handles versioning.

// Content start

This topic provides reference information on how the {aws} Cloud Development Kit ({aws} CDK) handles versioning.

Version numbers consist of three numeric version parts: *major*.\_minor\_.\_patch\_, and strictly adhere to the https://semver.org[semantic versioning] model. This means that breaking changes to stable APIs are limited to major releases.

Minor and patch releases are backward compatible. The code written in a previous version with the same major version can be upgraded to a newer version within the same major version. It will also continue to build and run, producing the same output.

[#cdk-toolkit-versioning] == {aws} CDK CLI compatibility

Each version of the main {aws} CDK library (aws-cdk-lib) is compatible with the {aws} CDK CLI (aws-cdk-cli) version that was current at the time of the CDK library’s release. It is also compatible with any newer version of the CDK CLI. Each version of the CDK library maintains this compatibility until the library’s *End of Life* date. Therefore, as long as you’re using a supported CDK library version, it is always safe to upgrade your CDK CLI version.

Each version of the CDK library may also work with CDK CLI versions older than the version that was current at the time of the CDK library’s release. However, this is not guaranteed. Compatibility depends on the CDK library’s cloud assembly schema version. The {aws} CDK generates a cloud assembly during synthesis and the CDK CLI consumes it for deployment. The schema that defines the format of the cloud assembly is strictly specified and versioned. Therefore, an older version of the CDK CLI would need to support the cloud assembly schema version of the CDK library for them to be compatible.

When the cloud assembly version required by the CDK library is not compatible with the version supported by the CDK CLI, you receive an error message like the following:

== [source,none,subs=“verbatim,attributes”]

Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but found 4.0.0. Please upgrade your CLI in order to interact with this app. —

To resolve this error, update the CDK CLI to a version compatible with the required cloud assembly version, or to the latest available version. The alternative (downgrading the construct library modules your app uses) is generally not recommended.

= [NOTE]

For more information on the exact combinations of versions that work together, see the https://github.com/aws/aws-cdk-cli/blob/main/COMPATIBILITY.md[compatibility table] in the *aws-cdk-cli GitHub repository*.

====

[#aws-construct-lib-stability] == {aws} Construct Library versioning

The modules in the {aws} Construct Library move through various stages as they are developed from concept to mature API. Different stages offer varying degrees of API stability in subsequent versions of the {aws} CDK.

APIs in the main {aws} CDK library, aws-cdk-lib, are stable, and the library is fully semantically versioned. This package includes {aws} CloudFormation (L1) constructs for all {aws} services and all stable higher-level (L2 and L3) modules. (It also includes the core CDK classes like App and Stack). APIs will not be removed from this package (though they may be deprecated) until the next major release of the CDK. No individual API will ever have breaking changes. When a breaking change is required, an entirely new API will be added.

New APIs under development for a service already incorporated in aws-cdk-lib are identified using a Beta<N> suffix, where N starts at 1 and is incremented with each breaking change to the new API. Beta<N> APIs are never removed, only deprecated, so your existing app continues to work with newer versions of aws-cdk-lib. When the API is deemed stable, a new API without the Beta<N> suffix is added.

When higher-level (L2 or L3) APIs begin to be developed for an {aws} service that previously had only L1 APIs, those APIs are initially distributed in a separate package. The name of such a package has an “Alpha” suffix, and its version matches the first version of aws-cdk-lib it is compatible with, with an alpha sub-version. When the module supports the intended use cases, its APIs are added to aws-cdk-lib.

[#aws-construct-lib-versioning-binding] == Language binding stability

Over time, we might add support to the {aws} CDK for additional programming languages. Although the API described in all the languages is the same, the way that API is expressed varies by language and might 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.

[cols=“1,1”, options=“header”] |=== | Language | Stability

|=== | TypeScript | Stable |===

|=== | JavaScript | Stable |===

|=== | Python | Stable |===

|=== | Java | Stable |===

|=== | C#/.NET | Stable |===

|=== | Go | Stable |===