: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]