:doctype: book

include::attributes.txt[]

// Attributes

[.topic] :info\_titleabbrev: CDK stacks :info\_abstract: An {aws} CDK stack is a single unit of deployment. It represents a collection of {aws} resources. When you develop + CDK applications, you define {aws} resources using CDK constructs and organize them within CDK + stacks. When you deploy, the resources within a CDK stack are deployed together as an {aws} CloudFormation stack. :keywords: {aws} CDK, {aws} CDK concepts, {aws} CDK stacks, stacks, stack, {aws} CloudFormation, {aws} CloudFormation stacks

[#stacks] = Introduction to {aws} CDK stacks

== [abstract]

## An {aws} CDK stack is a single unit of deployment. It represents a collection of {aws} resources. When you develop CDK applications, you define {aws} resources using CDK constructs and organize them within CDK stacks. When you deploy, the resources within a CDK stack are deployed together as an {aws} CloudFormation stack.

// Content start

An {aws} CDK stack is the smallest single unit of deployment. It represents a collection of {aws} resources that you define using CDK constructs. When you deploy CDK apps, the resources within a CDK stack are deployed together as an {aws} CloudFormation stack. To learn more about {aws} CloudFormation stacks, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/stacks.html[Managing {aws} resources as a single unit with {aws} CloudFormation stacks] in the *{aws} CloudFormation User Guide*.

You define a stack by extending or inheriting from the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack] construct. The following example is a common pattern for defining a CDK stack on a separate file, known as a *stack file*. Here, we extend or inherit the Stack class and define a constructor that accepts scope, id, and props. Then, we invoke the base Stack class constructor using super with the received scope, id, and props:

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

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

// Define your constructs here

} } —

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

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

// Define your constructs here

} }

== module.exports = { MyCdkStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Stack, ) from constructs import Construct

class MyCdkStack(Stack):

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

# Define your constructs here —-

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

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

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

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

// Define your constructs here } } —-

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

namespace MyCdk { public class MyCdkStack : Stack { internal MyCdkStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define your constructs here } } } —

Go:: + [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 CdkDemoAppStackProps struct { awscdk.StackProps }

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

…. // The code that defines your stack goes here

return stack } ….

func main() { defer jsii.Close()

…. app := awscdk.NewApp(nil)

NewCdkDemoAppStack(app, “CdkDemoAppStack”, &CdkDemoAppStackProps{ awscdk.StackProps{ Env: env(), }, })

app.Synth(nil) } ….

== //…

====

The previous example has only defined a stack. To create the stack, it must be instantiated within the context of your CDK app. A common pattern is to define your CDK app and initialize your stack on a separate file, known as an *application file*.

The following is an example that creates a CDK stack named MyCdkStack. Here, the CDK app is created and MyCdkStack is instantiated in the context of the app:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node import ‘source-map-support/register’; import \* as cdk from ‘aws-cdk-lib’; import { MyCdkStack } from ‘../lib/my-cdk-stack’;

const app = new cdk.App(); new MyCdkStack(app, ‘MyCdkStack’, { }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node

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

const app = new cdk.App(); new MyCdkStack(app, ‘MyCdkStack’, { }); —

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

import aws\_cdk as cdk

from my\_cdk.my\_cdk\_stack import MyCdkStack

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

== app.synth()

Java:: + [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 MyCdkApp { public static void main(final String[] args) { App app = new App();

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

app.synth(); } } —- ….

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

namespace MyCdk { sealed class Program { public static void Main(string[] args) { var app = new App(); new MyCdkStack(app, “MyCdkStack”, new StackProps {}); app.Synth(); } } } —

Go:: + [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” )

// …

func main() { defer jsii.Close()

app := awscdk.NewApp(nil)

NewMyCdkStack(app, “MyCdkStack”, &MyCdkStackProps{ awscdk.StackProps{ Env: env(), }, })

app.Synth(nil) }

== // …

====

The following example creates a CDK app that contains two stacks:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new App();

new MyFirstStack(app, ‘stack1’); new MySecondStack(app, ‘stack2’);

== app.synth();

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new App();

new MyFirstStack(app, ‘stack1’); new MySecondStack(app, ‘stack2’);

== app.synth();

Python:: + [source,python,subs=“verbatim,attributes”] — app = App()

MyFirstStack(app, ‘stack1’) MySecondStack(app, ‘stack2’)

== app.synth()

Java:: + [source,java,subs=“verbatim,attributes”] — App app = new App();

new MyFirstStack(app, “stack1”); new MySecondStack(app, “stack2”);

== app.synth();

C#:: + [source,csharp,subs=“verbatim,attributes”] — var app = new App();

new MyFirstStack(app, “stack1”); new MySecondStack(app, “stack2”);

== app.Synth();

Go:: + [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 MyFirstStackProps struct { awscdk.StackProps }

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

…. // The code that defines your stack goes here

return myFirstStack } ….

type MySecondStackProps struct { awscdk.StackProps }

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

…. // The code that defines your stack goes here

return mySecondStack } ….

func main() { defer jsii.Close()

…. app := awscdk.NewApp(nil)

NewMyFirstStack(app, “MyFirstStack”, &MyFirstStackProps{ awscdk.StackProps{ Env: env(), }, })

NewMySecondStack(app, “MySecondStack”, &MySecondStackProps{ awscdk.StackProps{ Env: env(), }, })

app.Synth(nil) } ….

== // …

====

[#stack-api] == About the stack API

The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack] object provides a rich API, including the following:

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

For information about how environments are determined for stacks, see xref:environments[Environments for the {aws} CDK]. +

* stack.addDependency(stack) (Python: stack.add\_dependency(stack)) – Can be used to explicitly define dependency order between two stacks. This order is respected by the cdk deploy command when deploying multiple stacks at once.
* stack.tags – Returns a https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.TagManager.html[TagManager] that you can use to add or remove stack-level tags. This tag manager tags all resources within the stack, and also tags the stack itself when it’s created through {aws} CloudFormation.
* stack.partition, stack.urlSuffix (Python: url\_suffix), stack.stackId (Python: stack\_id), and stack.notificationArn (Python: notification\_arn) – Return tokens that resolve to the respective {aws} CloudFormation pseudo parameters, such as +{ "Ref": "{aws}::Partition" }+. These tokens are associated with the specific stack object so that the {aws} CDK framework can identify cross-stack references.
* stack.availabilityZones (Python: availability\_zones) – Returns the set of Availability Zones available in the environment in which this stack is deployed. For environment-agnostic stacks, this always returns an array with two Availability Zones. For environment-specific stacks, the {aws} CDK queries the environment and returns the exact set of Availability Zones available in the Region that you specified.
* stack.parseArn(arn) and stack.formatArn(comps) (Python: parse\_arn, format\_arn) – Can be used to work with Amazon Resource Names (ARNs).
* stack.toJsonString(obj) (Python: to\_json\_string) – Can be used to format an arbitrary object as a JSON string that can be embedded in an {aws} CloudFormation template. The object can include tokens, attributes, and references, which are only resolved during deployment.
* stack.templateOptions (Python: template\_options) – Use to specify {aws} CloudFormation template options, such as Transform, Description, and Metadata, for your stack.

[#stacks-work] == Working with stacks

Stacks are deployed as an {aws} CloudFormation stack into an {aws} xref:environments[environment]. The environment covers a specific {aws} account and {aws} Region.

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

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

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

$ cdk synth ++++++—-++++++

This approach is conceptually different from how {aws} CloudFormation templates are normally used, where a template can be deployed multiple times and parameterized through https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html[{aws} CloudFormation parameters]. Although {aws} CloudFormation parameters can be defined in the {aws} CDK, they are generally discouraged because {aws} CloudFormation parameters are resolved only during deployment. This means that you cannot determine their value in your code.

For example, to conditionally include a resource in your app based on a parameter value, you must set up an https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/conditions-section-structure.html[{aws} CloudFormation condition] and tag the resource with it. The {aws} CDK takes an approach where concrete templates are resolved at synthesis time. Therefore, you can use an if statement to check the value to determine whether a resource should be defined or some behavior should be applied.

= [NOTE]

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

====

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

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

interface EnvProps { prod: boolean; }

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

class MyService extends Construct {

constructor(scope: Construct, id: string, props?: EnvProps) {

super(scope, id);

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

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

== app.synth();

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

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

class MyService extends Construct {

constructor(scope, id, props) {

super(scope, id);

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

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

== app.synth();

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

= imagine these stacks declare a bunch of related resources

class ControlPlane(Stack): pass class DataPlane(Stack): pass class Monitoring(Stack): pass

class MyService(Construct):

def *init*(self, scope: Construct, id: str, \*, prod=False):

super().\_\_init\_\_(scope, id)

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

app = App(); MyService(app, “beta”) MyService(app, “prod”, prod=True)

== app.synth()

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

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

public class MyApp {

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

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

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

static class MyService extends Construct { MyService(Construct scope, String id) { this(scope, id, false); }

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

}

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

new MyService(app, "beta");  
new MyService(app, "prod", true);  
  
app.synth();

} } —- ….

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

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

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

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

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

class Program { static void Main(string[] args) {

var app = new App();  
 new MyService(app, "beta");  
 new MyService(app, "prod", prod: true);  
 app.Synth();

} } —-

Go:: + [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 ControlPlaneStackProps struct { awscdk.StackProps }

func NewControlPlaneStack(scope constructs.Construct, id string, props \*ControlPlaneStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } ControlPlaneStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

…. // The code that defines your stack goes here

return ControlPlaneStack } ….

type DataPlaneStackProps struct { awscdk.StackProps }

func NewDataPlaneStack(scope constructs.Construct, id string, props \*DataPlaneStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } DataPlaneStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

…. // The code that defines your stack goes here

return DataPlaneStack } ….

type MonitoringStackProps struct { awscdk.StackProps }

func NewMonitoringStack(scope constructs.Construct, id string, props \*MonitoringStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } MonitoringStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

…. // The code that defines your stack goes here

return MonitoringStack } ….

type MyServiceStackProps struct { awscdk.StackProps Prod bool }

func NewMyServiceStack(scope constructs.Construct, id string, props \*MyServiceStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } MyServiceStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

…. NewControlPlaneStack(MyServiceStack, “cp”, &ControlPlaneStackProps{ StackProps: sprops, }) NewDataPlaneStack(MyServiceStack, “data”, &DataPlaneStackProps{ StackProps: sprops, }) NewMonitoringStack(MyServiceStack, “mon”, &MonitoringStackProps{ StackProps: sprops, })

return MyServiceStack } ….

func main() { defer jsii.Close()

…. app := awscdk.NewApp(nil)

betaProps := MyServiceStackProps{ StackProps: awscdk.StackProps{ Env: env(), }, Prod: false, }

NewMyServiceStack(app, “beta”, &betaProps)

prodProps := MyServiceStackProps{ StackProps: awscdk.StackProps{ Env: env(), }, Prod: true, }

NewMyServiceStack(app, “prod”, &prodProps)

app.Synth(nil) } ….

== // …

This {aws} CDK app eventually consists of six stacks, three for each environment:

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

$ cdk ls

betacpDA8372D3 betadataE23DB2BA betamon632BD457 prodcp187264CE proddataF7378CE5 prodmon631A1083 — ====

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyStack(this, ‘not:a:stack:name’, { stackName: ‘this-is-stack-name’ }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new MyStack(this, ‘not:a:stack:name’, { stackName: ‘this-is-stack-name’ }); —

Python:: + [source,python,subs=“verbatim,attributes”] — MyStack(self, “not:a:stack:name”, stack\_name=“this-is-stack-name”) —

Java:: + [source,java,subs=“verbatim,attributes”] — new MyStack(this, “not:a:stack:name”, StackProps.builder() .StackName(“this-is-stack-name”).build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new MyStack(this, “not:a:stack:name”, new StackProps { StackName = “this-is-stack-name” }); — ====

[#stack-nesting] === Working with nested stacks

A *nested stack* is a CDK stack that you create inside another stack, known as the parent stack. You create nested stacks using the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.NestedStack.html[NestedStack] construct.

By using nested stacks, you can organize resources across multiple stacks. Nested stacks also offer a way around the {aws} CloudFormation 500-resource limit for stacks. A nested stack counts as only one resource in the stack that contains it. However, it can contain up to 500 resources, including additional nested stacks.

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

At synthesis time, the nested stack is synthesized to its own {aws} CloudFormation template, which is uploaded to the {aws} CDK staging bucket at deployment. Nested stacks are bound to their parent stack and are not treated as independent deployment artifacts. They aren’t listed by cdk list, and they can’t be deployed by cdk deploy.

References between parent stacks and nested stacks are automatically translated to stack parameters and outputs in the generated {aws} CloudFormation templates, as with any xref:resource-stack[cross-stack reference].

= [WARNING]

Changes in security posture are not displayed before deployment for nested stacks. This information is displayed only for top-level stacks.

====