# Core Concepts

Master the fundamental concepts that power AWS CDK.

## 🧠 Essential Concepts

### Building Blocks

* [Apps](apps.md) - The root of your CDK application
* [Stacks](stacks.md) - Deployment units in CDK
* [Constructs](constructs.md) - Reusable cloud components

### Advanced Concepts

* [Tokens](tokens.md) - Managing unknown values ([detailed guide](tokens/))
* [Environments](environments.md) - Multi-environment deployments
* [Context](context.md) - Configuration and feature flags

### Working with Multiple Resources

* [Create Multiple Stacks](create-multiple-stacks.md) - Multi-stack applications
* [Identifiers](identifiers.md) - Resource naming and references
* [Parameters](parameters.md) - Parameterizing your stacks

### Configuration

* [Feature Flags](feature-flags.md) - Controlling CDK behavior
* [Core Concepts Overview](core-concepts.md) - High-level overview

## 🎯 Learning Path

**For beginners:** 1. Start with [Core Concepts Overview](core-concepts.md) 2. Learn about [Constructs](constructs.md) 3. Understand [Stacks](stacks.md) and [Apps](apps.md) 4. Explore [Tokens](tokens.md) when you need dynamic values

**For intermediate users:** - [Environments](environments.md) for multi-env deployments - [Create Multiple Stacks](create-multiple-stacks.md) for complex applications

## 🔗 Related Sections

* [Getting Started](../01-getting-started/) - Set up your environment first
* [Development](../03-development/) - Apply these concepts in practice
* [Deployment](../04-deployment/) - Deploy your applications

# AWS CDK Tokens Guide

This section covers AWS CDK tokens in detail, split into logical learning modules for better understanding.

## Learning Path

1. [**Tokens Introduction**](01-tokens-introduction.md) - What are tokens and why are they important
2. [**Tokens Examples**](02-tokens-examples.md) - Basic examples of working with tokens
3. [**Passing Tokens**](03-tokens-passing.md) - How to pass tokens between constructs and stacks
4. [**How Tokens Work**](04-tokens-how-they-work.md) - Understanding token encoding and resolution
5. [**String Tokens**](05-tokens-string.md) - Working with string-based tokens
6. [**List Tokens**](06-tokens-list.md) - Working with list/array tokens
7. [**Number Tokens**](07-tokens-number.md) - Working with numeric tokens
8. [**Advanced Token Topics**](08-tokens-advanced.md) - Lazy tokens, JSON tokens, and advanced patterns

## Quick Start

If you’re new to tokens, start with the [Introduction](01-tokens-introduction.md) and [Examples](02-tokens-examples.md). For specific use cases, jump to the relevant section.

## Cross-References

This content relates to: - [Constructs](../constructs.md) - How tokens work within constructs - [Stacks](../stacks.md) - Passing tokens between stacks - [Apps](../apps.md) - Token resolution at the app level

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Tokens :info\_abstract: In the {aws} Cloud Development Kit ({aws} CDK), tokens are placeholders for values that aren’t known when defining constructs or synthesizing stacks. These values will be fully resolved at deployment, when your actual infrastructure is created. When developing {aws} CDK applications, you will work with tokens to manage these values across your application. :keywords: {aws} CDK, Tokens, {aws} CloudFormation, concepts

[#tokens] = Tokens and the {aws} CDK

== [abstract]

## In the {aws} Cloud Development Kit ({aws} CDK), *tokens* are placeholders for values that aren’t known when defining constructs or synthesizing stacks. These values will be fully resolved at deployment, when your actual infrastructure is created. When developing {aws} CDK applications, you will work with tokens to manage these values across your application.

// Content start

In the {aws} Cloud Development Kit ({aws} CDK), *tokens* are placeholders for values that aren’t known when defining constructs or synthesizing stacks. These values will be fully resolved at deployment, when your actual infrastructure is created. When developing {aws} CDK applications, you will work with tokens to manage these values across your application.

[#tokens-example] == Token example

The following is an example of a CDK stack that defines a construct for an Amazon Simple Storage Service (Amazon S3) bucket. Since the name of our bucket is not yet known, the value for bucketName is stored as a token:

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// Store value of the S3 bucket name const myBucketName = myBucket.bucketName;

// Print the current value for the S3 bucket name at synthesis console.log(“myBucketName:” + bucketName); } } —- ….

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// Store value of the S3 bucket name const myBucketName = myBucket.bucketName;

// Print the current value for the S3 bucket name at synthesis console.log(“myBucketName:” + myBucketName); } } ….

== module.exports = { CdkDemoAppStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Stack ) from constructs import Construct from aws\_cdk import aws\_s3 as s3

class CdkDemoAppStack(Stack):

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

# Define an S3 bucket  
my\_bucket = s3.Bucket(self, "myBucket")  
  
# Store the value of the S3 bucket name  
my\_bucket\_name = my\_bucket.bucket\_name  
  
# Print the current value for the S3 bucket name at synthesis  
print(f"myBucketName: {my\_bucket\_name}") ----

….

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

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.services.s3.Bucket;

import java.util.Map;

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

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

// Define an S3 bucket  
Bucket myBucket = Bucket.Builder.create(this, "myBucket")  
 .build();  
  
// Store the token for the bucket name  
String myBucketName = myBucket.getBucketName();  
  
// Print the token at synthesis  
System.out.println("myBucketName: " + myBucketName);

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.{aws}.S3;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define an S3 bucket var myBucket = new Bucket(this, “myBucket”);

…. // Store the token for the bucket name var myBucketName = myBucket.BucketName;

// Print the token at synthesis  
 System.Console.WriteLine($"myBucketName: {myBucketName}");  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( “fmt”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awss3” “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)

…. // Define an S3 bucket myBucket := awss3.NewBucket(stack, jsii.String(“myBucket”), &awss3.BucketProps{})

// Store the token for the bucket name myBucketName := myBucket.BucketName()

// Print the token at synthesis fmt.Println(“myBucketName:”, \*myBucketName)

return stack } ….

== // …

When we run cdk synth to synthesize our stack, the value for myBucketName will be displayed in the token format of ${Token[TOKEN.<1234>]}. This token format is a result of how the {aws} CDK encodes tokens. In this example, the token is encoded as a string:

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

$ cdk synth –quiet myBucketName: ${Token[TOKEN.21]} —

Since the value for our bucket name is not known at synthesis, the token is rendered as myBucket<unique-hash>. Our {aws} CloudFormation template uses the Ref intrinsic function to reference its value, which will be known at deployment:

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

Resources: myBucket<5AF9C99B>: # … Outputs: bucketNameOutput: Description: The name of the S3 bucket Value: Ref: myBucket<5AF9C99B> — ====

For more information on how the unique hash is generated, see xref:how-synth-default-logical-ids[Generated logical IDs in your {aws} CloudFormation template].

[#tokens-example] == Token example

The following is an example of a CDK stack that defines a construct for an Amazon Simple Storage Service (Amazon S3) bucket. Since the name of our bucket is not yet known, the value for bucketName is stored as a token:

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// Store value of the S3 bucket name const myBucketName = myBucket.bucketName;

// Print the current value for the S3 bucket name at synthesis console.log(“myBucketName:” + bucketName); } } —- ….

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// Store value of the S3 bucket name const myBucketName = myBucket.bucketName;

// Print the current value for the S3 bucket name at synthesis console.log(“myBucketName:” + myBucketName); } } ….

== module.exports = { CdkDemoAppStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Stack ) from constructs import Construct from aws\_cdk import aws\_s3 as s3

class CdkDemoAppStack(Stack):

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

# Define an S3 bucket  
my\_bucket = s3.Bucket(self, "myBucket")  
  
# Store the value of the S3 bucket name  
my\_bucket\_name = my\_bucket.bucket\_name  
  
# Print the current value for the S3 bucket name at synthesis  
print(f"myBucketName: {my\_bucket\_name}") ----

….

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

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.services.s3.Bucket;

import java.util.Map;

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

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

// Define an S3 bucket  
Bucket myBucket = Bucket.Builder.create(this, "myBucket")  
 .build();  
  
// Store the token for the bucket name  
String myBucketName = myBucket.getBucketName();  
  
// Print the token at synthesis  
System.out.println("myBucketName: " + myBucketName);

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.{aws}.S3;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define an S3 bucket var myBucket = new Bucket(this, “myBucket”);

…. // Store the token for the bucket name var myBucketName = myBucket.BucketName;

// Print the token at synthesis  
 System.Console.WriteLine($"myBucketName: {myBucketName}");  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( “fmt”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awss3” “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)

…. // Define an S3 bucket myBucket := awss3.NewBucket(stack, jsii.String(“myBucket”), &awss3.BucketProps{})

// Store the token for the bucket name myBucketName := myBucket.BucketName()

// Print the token at synthesis fmt.Println(“myBucketName:”, \*myBucketName)

return stack } ….

== // …

When we run cdk synth to synthesize our stack, the value for myBucketName will be displayed in the token format of ${Token[TOKEN.<1234>]}. This token format is a result of how the {aws} CDK encodes tokens. In this example, the token is encoded as a string:

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

$ cdk synth –quiet myBucketName: ${Token[TOKEN.21]} —

Since the value for our bucket name is not known at synthesis, the token is rendered as myBucket<unique-hash>. Our {aws} CloudFormation template uses the Ref intrinsic function to reference its value, which will be known at deployment:

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

Resources: myBucket<5AF9C99B>: # … Outputs: bucketNameOutput: Description: The name of the S3 bucket Value: Ref: myBucket<5AF9C99B> — ====

For more information on how the unique hash is generated, see xref:how-synth-default-logical-ids[Generated logical IDs in your {aws} CloudFormation template].

[#tokens-passing] == Passing tokens

Tokens can be passed around as if they were the actual value they represent. The following is an example that passes the token for our bucket name to a construct for an {aws} Lambda function:

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// …

// Define a Lambda function const myFunction = new lambda.Function(this, “myFunction”, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: “index.handler”, code: lambda.Code.fromInline(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };), functionName: myBucketName + “Function”, // Pass token for the S3 bucket name environment: { BUCKET\_NAME: myBucketName, // Pass token for the S3 bucket name } }); } } —- ….

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// …

// Define a Lambda function const myFunction = new lambda.Function(this, ‘myFunction’, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: ‘index.handler’, code: lambda.Code.fromInline(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };), functionName: myBucketName + ‘Function’, // Pass token for the S3 bucket name environment: { BUCKET\_NAME: myBucketName, // Pass token for the S3 bucket name } }); } } ….

== module.exports = { CdkDemoAppStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Stack ) from constructs import Construct from aws\_cdk import aws\_s3 as s3 from aws\_cdk import aws\_lambda as \_lambda

class CdkDemoAppStack(Stack):

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

# Define an S3 bucket  
my\_bucket = s3.Bucket(self, "myBucket")  
  
# ...  
  
# Define a Lambda function  
my\_function = \_lambda.Function(self, "myFunction",  
 runtime=\_lambda.Runtime.NODEJS\_20\_X,  
 handler="index.handler",  
 code=\_lambda.Code.from\_inline("""  
 exports.handler = async function(event) {  
 return {  
 statusCode: 200,  
 body: JSON.stringify('Hello World!'),  
 };  
 };  
 """),  
 function\_name=f"{my\_bucket\_name}Function", # Pass token for the S3 bucket name  
 environment={  
 "BUCKET\_NAME": my\_bucket\_name # Pass token for the S3 bucket name  
 }  
) ----

….

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

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.services.s3.Bucket; import software.amazon.awscdk.services.lambda.Code; import software.amazon.awscdk.services.lambda.Function; import software.amazon.awscdk.services.lambda.Runtime;

import java.util.Map;

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

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

// Define an S3 bucket  
Bucket myBucket = Bucket.Builder.create(this, "myBucket")  
 .build();  
  
// ...  
  
// Define a Lambda function  
Function myFunction = Function.Builder.create(this, "myFunction")  
 .runtime(Runtime.NODEJS\_20\_X)  
 .handler("index.handler")  
 .code(Code.fromInline(  
 "exports.handler = async function(event) {" +  
 "return {" +  
 "statusCode: 200," +  
 "body: JSON.stringify('Hello World!')," +  
 "};" +  
 "};"  
 ))  
 .functionName(myBucketName + "Function") // Pass the token for the s3 bucket to the function construct  
 .environment(Map.of("BUCKET\_NAME", myBucketName)) // Pass the bucket name as environment variable  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.{aws}.S3; using Amazon.CDK.{aws}.Lambda; using System; using System.Collections.Generic;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define an S3 bucket var myBucket = new Bucket(this, “myBucket”);

…. // …

// Define a Lambda function  
 var myFunction = new Function(this, "myFunction", new FunctionProps  
 {  
 Runtime = Runtime.NODEJS\_20\_X,  
 Handler = "index.handler",  
 Code = Code.FromInline(@"  
 exports.handler = async function(event) {  
 return {  
 statusCode: 200,  
 body: JSON.stringify('Hello World!'),  
 };  
 };  
 "),  
 // Pass the token for the S3 bucket name  
 Environment = new Dictionary<string, string>  
 {  
 { "BUCKET\_NAME", myBucketName }  
 },  
 FunctionName = $"{myBucketName}Function" // Pass the token for the s3 bucket to the function construct  
 });  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( “fmt”

“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/awss3” “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)

…. // Define an S3 bucket myBucket := awss3.NewBucket(stack, jsii.String(“myBucket”), &awss3.BucketProps{})

// …

// Define a Lambda function myFunction := awslambda.NewFunction(stack, jsii.String(“myFunction”), &awslambda.FunctionProps{ Runtime: awslambda.Runtime\_NODEJS\_20\_X(), Handler: jsii.String(“index.handler”), Code: awslambda.Code\_FromInline(jsii.String(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };)), FunctionName: jsii.String(fmt.Sprintf(“%sFunction”, *myBucketName)), // Pass the token for the S3 bucket to the function name Environment: &map[string]*string{ “BUCKET\_NAME”: myBucketName, }, })

return stack } // … —- ….

When we synthesize our template, the Ref and Fn::Join intrinsic functions are used to specify the values, which will be known at deployment:

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

Resources: myBucket<5AF9C99B>: Type: {aws}::S3::Bucket # … myFunction<884E1557>: Type: {aws}::Lambda::Function Properties: # … Environment: Variables: BUCKET\_NAME: Ref: myBucket<5AF9C99B> FunctionName: Fn::Join: - “” - - Ref: myBucket<5AF9C99B> - Function # … — ====

[#tokens-work] == How token encodings work

Tokens are objects that implement the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.IResolvable.html[IResolvable] interface, which contains a single resolve method. During synthesis, the {aws} CDK calls this method to produce the final value for tokens in your CloudFormation template.

= [NOTE]

You’ll rarely work directly with the IResolvable interface. You will most likely only see string-encoded versions of tokens.

====

[#tokens-work-types] === Token encoding types

Tokens participate in the synthesis process to produce arbitrary values of any type. Other functions typically only accept arguments of basic types, such as string or number. To use tokens in these cases, you can encode them into one of three types by using static methods on the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Token.html[cdk.Token] class.

* https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Token.html#static-aswbrstringvalue-options[Token.asString] to generate a string encoding (or call .toString() on the token object).
* https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Token.html#static-aswbrlistvalue-options[Token.asList] to generate a list encoding.
* https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Token.html#static-aswbrnumbervalue[Token.asNumber] to generate a numeric encoding.

These take an arbitrary value, which can be an IResolvable, and encode them into a primitive value of the indicated type.

= [IMPORTANT]

Because any one of the previous types can potentially be an encoded token, be careful when you parse or try to read their contents. For example, if you attempt to parse a string to extract a value from it, and the string is an encoded token, your parsing fails. Similarly, if you try to query the length of an array or perform math operations with a number, you must first verify that they aren’t encoded tokens.

====

[#tokens-check] == How to check for tokens in your app

To check whether a value has an unresolved token in it, call the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Token.html#static-iswbrunresolvedobj[Token.isUnresolved] (Python: is\_unresolved) method. The following is an example that checks if the value for our Amazon S3 bucket name is a token. If its not a token, we then validate the length of the bucket name:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — // …

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’); // …

// Check if bucket name is a token. If not, check if length is less than 10 characters if (cdk.Token.isUnresolved(myBucketName)) { console.log(“Token identified.”); } else if (!cdk.Token.isUnresolved(myBucketName) && myBucketName.length > 10) { throw new Error(‘Maximum length for name is 10 characters.’); };

// … } } —- ….

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

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

…. // Define an S3 bucket const myBucket = new s3.Bucket(this, ‘myBucket’);

// …

// Check if bucket name is a token. If not, check if length is less than 10 characters if (Token.isUnresolved(myBucketName)) { console.log(“Token identified.”); } else if (!Token.isUnresolved(myBucketName) && myBucketName.length > 10) { throw new Error(‘Maximum length for name is 10 characters.’); };

// … } } —- ….

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

= …

class CdkDemoAppStack(Stack):

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

# Define an S3 bucket  
my\_bucket = s3.Bucket(self, "myBucket")  
  
# ...  
  
# Check if bucket name is a token. If not, check if length is less than 10 characters  
if Token.is\_unresolved(my\_bucket\_name):  
 print("Token identified.")  
elif not Token.is\_unresolved(my\_bucket\_name) and len(my\_bucket\_name) < 10:  
 raise ValueError("Maximum length for name is 10 characters.")  
  
# ... ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — // … import software.amazon.awscdk.Token; import software.amazon.awscdk.services.s3.Bucket; // …

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

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

// Define an S3 bucket  
Bucket myBucket = Bucket.Builder.create(this, "myBucket")  
 .build();  
  
// ...  
  
// Get the bucket name  
String myBucketName = myBucket.getBucketName();  
  
// Check if the bucket name is a token. If not, check if length is less than 10 characters  
if (Token.isUnresolved(myBucketName)) {  
 System.out.println("Token identified.");  
} else if (!Token.isUnresolved(myBucketName) && myBucketName.length() > 10) {  
 throw new IllegalArgumentException("Maximum length for name is 10 characters.");  
}  
  
// ...

} } } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.AWS.S3; using Amazon.CDK.AWS.Lambda; using System; using System.Collections.Generic;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define an S3 bucket var myBucket = new Bucket(this, “myBucket”);

…. // …

// Get the bucket name  
 var myBucketName = myBucket.BucketName;  
  
 // Check if bucket name is a token. If not, check if length is less than 10 characters  
 if (Token.IsUnresolved(myBucketName))  
 {  
 System.Console.WriteLine("Token identified.");  
 }  
 else if (!Token.IsUnresolved(myBucketName) && myBucketName.Length > 10)  
 {  
 throw new System.Exception("Maximum length for name is 10 characters.");  
 }  
  
 // ...  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — // …

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)

…. // Define an S3 bucket myBucket := awss3.NewBucket(stack, jsii.String(“myBucket”), &awss3.BucketProps{})

// …

// Check if the bucket name is unresolved (a token) if tokenUnresolved := awscdk.Token\_IsUnresolved(myBucketName); tokenUnresolved != nil && *tokenUnresolved { fmt.Println(“Token identified.”) } else if tokenUnresolved != nil && !*tokenUnresolved && len(\*myBucketName) > 10 { panic(“Maximum length for name is 10 characters.”) }

// … } —- ==== ….

When we run cdk synth, myBucketName is identified as a token:

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

$ cdk synth –quiet Token identified. —

= [NOTE]

You can use token encodings to escape the type system. For example, you could string-encode a token that produces a number value at synthesis time. If you use these functions, it’s your responsibility to make sure that your template resolves to a usable state after synthesis.

====

[#tokens-string] == Working with string-encoded tokens

String-encoded tokens look like the following.

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

## ${TOKEN[Bucket.Name.1234]}

They can be passed around like regular strings, and can be concatenated, as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const functionName = bucket.bucketName + ‘Function’; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const functionName = bucket.bucketName + ‘Function’; —

Python:: + [source,python,subs=“verbatim,attributes”] — function\_name = bucket.bucket\_name + “Function” —

Java:: + [source,java,subs=“verbatim,attributes”] — String functionName = bucket.getBucketName().concat(“Function”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — string functionName = bucket.BucketName + “Function”; —

Go:: + [source,go,subs=“verbatim,attributes”] — functionName := \*bucket.BucketName() + “Function” — ====

You can also use string interpolation, if your language supports it, as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const functionName = ${bucket.bucketName}Function; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const functionName = ${bucket.bucketName}Function; —

Python:: + [source,python,subs=“verbatim,attributes”] — function\_name = f”{bucket.bucket\_name}Function” —

Java:: + [source,java,subs=“verbatim,attributes”] — String functionName = String.format(“%sFunction”. bucket.getBucketName()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — string functionName = $"${bucket.bucketName}Function”; —

Go:: Use fmt.Sprintf for similar functionality: + [source,go,subs=“verbatim,attributes”] — functionName := fmt.Sprintf(“%sFunction”, \*bucket.BucketName()) — ====

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

[#tokens-list] == Working with list-encoded tokens

List-encoded tokens look like the following:

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

## [“#{TOKEN[Stack.NotificationArns.1234]}”]

The only safe thing to do with these lists is pass them directly to other constructs. Tokens in string list form cannot be concatenated, nor can an element be taken from the token. The only safe way to manipulate them is by using {aws} CloudFormation intrinsic functions like https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-select.html[Fn.select].

[#tokens-number] == Working with number-encoded tokens

Number-encoded tokens are a set of tiny negative floating-point numbers that look like the following.

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

## -1.8881545897087626e+289

As with list tokens, you cannot modify the number value, as doing so is likely to break the number token.

The following is an example of a construct that contains a token encoded as a number:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Stack, Duration, StackProps } from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import \* as rds from ‘aws-cdk-lib/aws-rds’; import \* as ec2 from ‘aws-cdk-lib/aws-ec2’;

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

…. // Define a new VPC const vpc = new ec2.Vpc(this, ‘MyVpc’, { maxAzs: 3, // Maximum number of availability zones to use });

// Define an RDS database cluster const dbCluster = new rds.DatabaseCluster(this, ‘MyRDSCluster’, { engine: rds.DatabaseClusterEngine.AURORA, instanceProps: { vpc, }, });

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// Print the value for our token at synthesis console.log(“portToken:” + portToken); } } —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Stack, Duration } = require(‘aws-cdk-lib’); const lambda = require(‘aws-cdk-lib/aws-lambda’); const rds = require(‘aws-cdk-lib/aws-rds’); const ec2 = require(‘aws-cdk-lib/aws-ec2’);

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

…. // Define a new VPC const vpc = new ec2.Vpc(this, ‘MyVpc’, { maxAzs: 3, // Maximum number of availability zones to use });

// Define an RDS database cluster const dbCluster = new rds.DatabaseCluster(this, ‘MyRDSCluster’, { engine: rds.DatabaseClusterEngine.AURORA, instanceProps: { vpc, }, });

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// Print the value for our token at synthesis console.log(“portToken:” + portToken); } } ….

== module.exports = { CdkDemoAppStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Duration, Stack, ) from aws\_cdk import aws\_rds as rds from aws\_cdk import aws\_ec2 as ec2 from constructs import Construct

class CdkDemoAppStack(Stack):

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

# Define a new VPC  
vpc = ec2.Vpc(self, 'MyVpc',  
 max\_azs=3 # Maximum number of availability zones to use  
)  
  
# Define an RDS database cluster  
db\_cluster = rds.DatabaseCluster(self, 'MyRDSCluster',  
 engine=rds.DatabaseClusterEngine.AURORA,  
 instance\_props=rds.InstanceProps(  
 vpc=vpc  
 )  
)  
  
# Get the port token (this is a token encoded as a number)  
port\_token = db\_cluster.cluster\_endpoint.port  
  
# Print the value for our token at synthesis  
print(f"portToken: {port\_token}") ----

….

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

import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.services.ec2.Vpc; import software.amazon.awscdk.services.rds.DatabaseCluster; import software.amazon.awscdk.services.rds.DatabaseClusterEngine; import software.amazon.awscdk.services.rds.InstanceProps;

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

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

// Define a new VPC  
Vpc vpc = Vpc.Builder.create(this, "MyVpc")  
 .maxAzs(3) // Maximum number of availability zones to use  
 .build();  
  
// Define an RDS database cluster  
DatabaseCluster dbCluster = DatabaseCluster.Builder.create(this, "MyRDSCluster")  
 .engine(DatabaseClusterEngine.AURORA)  
 .instanceProps(InstanceProps.builder()  
 .vpc(vpc)  
 .build())  
 .build();  
  
// Get the port token (this is a token encoded as a number)  
Number portToken = dbCluster.getClusterEndpoint().getPort();  
  
// Print the value for our token at synthesis  
System.out.println("portToken: " + portToken);

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Constructs; using Amazon.CDK.{aws}.EC2; using Amazon.CDK.{aws}.RDS; using System; using System.Collections.Generic;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define a new VPC var vpc = new Vpc(this, “MyVpc”, new VpcProps { MaxAzs = 3 // Maximum number of availability zones to use });

…. // Define an RDS database cluster var dbCluster = new DatabaseCluster(this, “MyRDSCluster”, new DatabaseClusterProps { Engine = DatabaseClusterEngine.AURORA, // Remove parentheses InstanceProps = new Amazon.CDK.{aws}.RDS.InstanceProps // Specify RDS InstanceProps { Vpc = vpc } });

// Get the port token (this is a token encoded as a number)  
 var portToken = dbCluster.ClusterEndpoint.Port;  
  
 // Print the value for our token at synthesis  
 System.Console.WriteLine($"portToken: {portToken}");  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( “fmt”

“github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awsec2” “github.com/aws/aws-cdk-go/awscdk/v2/awsrds” “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)

…. // Define a new VPC vpc := awsec2.NewVpc(stack, jsii.String(“MyVpc”), &awsec2.VpcProps{ MaxAzs: jsii.Number(3), // Maximum number of availability zones to use })

// Define an RDS database cluster dbCluster := awsrds.NewDatabaseCluster(stack, jsii.String(“MyRDSCluster”), &awsrds.DatabaseClusterProps{ Engine: awsrds.DatabaseClusterEngine\_AURORA(), InstanceProps: &awsrds.InstanceProps{ Vpc: vpc, }, })

// Get the port token (this is a token encoded as a number) portToken := dbCluster.ClusterEndpoint().Port()

// Print the value for our token at synthesis fmt.Println(“portToken:”, portToken)

return stack } ….

== // …

====

When we run cdk synth, the value for portToken is displayed as a number-encoded token:

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

$ cdk synth –quiet portToken: -1.8881545897087968e+289 —

[#tokens-number-pass] === Pass number-encoded tokens

When you pass number-encoded tokens to other constructs, it may make sense to convert them to strings first. For example, if you want to use the value of a number-encoded string as part of a concatenated string, converting it helps with readability.

In the following example,portToken is a number-encoded token that we want to pass to our Lambda function as part of connectionString:

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

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

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// …

// Example connection string with the port token as a number const connectionString = jdbc:mysql://mydb.cluster.amazonaws.com:${portToken}/mydatabase;

// Use the connection string as an environment variable in a Lambda function const myFunction = new lambda.Function(this, ‘MyLambdaFunction’, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: ‘index.handler’, code: lambda.Code.fromInline(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };), environment: { DATABASE\_CONNECTION\_STRING: connectionString, // Using the port token as part of the string }, });

// Output the value of our connection string at synthesis console.log(“connectionString:” + connectionString);

// Output the connection string new CfnOutput(this, ‘ConnectionString’, { value: connectionString, }); } } —- ….

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

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

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// …

// Example connection string with the port token as a number const connectionString = jdbc:mysql://mydb.cluster.amazonaws.com:${portToken}/mydatabase;

// Use the connection string as an environment variable in a Lambda function const myFunction = new lambda.Function(this, ‘MyLambdaFunction’, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: ‘index.handler’, code: lambda.Code.fromInline(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };), environment: { DATABASE\_CONNECTION\_STRING: connectionString, // Using the port token as part of the string }, });

// Output the value of our connection string at synthesis console.log(“connectionString:” + connectionString);

// Output the connection string new CfnOutput(this, ‘ConnectionString’, { value: connectionString, }); } } ….

== module.exports = { CdkDemoAppStack }

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Duration, Stack, CfnOutput, ) from aws\_cdk import aws\_lambda as \_lambda

= …

class CdkDemoAppStack(Stack):

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

# Define a new VPC  
# ...  
  
# Define an RDS database cluster  
# ...  
  
# Get the port token (this is a token encoded as a number)  
port\_token = db\_cluster.cluster\_endpoint.port  
  
# ...  
  
# Example connection string with the port token as a number  
connection\_string = f"jdbc:mysql://mydb.cluster.amazonaws.com:{port\_token}/mydatabase"  
  
# Use the connection string as an environment variable in a Lambda function  
my\_function = \_lambda.Function(self, 'MyLambdaFunction',  
 runtime=\_lambda.Runtime.NODEJS\_20\_X,  
 handler='index.handler',  
 code=\_lambda.Code.from\_inline("""  
 exports.handler = async function(event) {  
 return {  
 statusCode: 200,  
 body: JSON.stringify('Hello World!'),  
 };  
 };  
 """),  
 environment={  
 'DATABASE\_CONNECTION\_STRING': connection\_string # Using the port token as part of the string  
 }  
)  
  
# Output the value of our connection string at synthesis  
print(f"connectionString: {connection\_string}")  
  
# Output the connection string  
CfnOutput(self, 'ConnectionString',  
 value=connection\_string  
) ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — // … import software.amazon.awscdk.CfnOutput; import software.amazon.awscdk.services.lambda.Function; import software.amazon.awscdk.services.lambda.Runtime; import software.amazon.awscdk.services.lambda.Code;

import java.util.Map;

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

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

// Define a new VPC  
// ...  
  
// Define an RDS database cluster  
// ...  
  
// Get the port token (this is a token encoded as a number)  
Number portToken = dbCluster.getClusterEndpoint().getPort();  
  
// ...  
  
// Example connection string with the port token as a number  
String connectionString = "jdbc:mysql://mydb.cluster.amazonaws.com:" + portToken + "/mydatabase";  
  
// Use the connection string as an environment variable in a Lambda function  
Function myFunction = Function.Builder.create(this, "MyLambdaFunction")  
 .runtime(Runtime.NODEJS\_20\_X)  
 .handler("index.handler")  
 .code(Code.fromInline(  
 "exports.handler = async function(event) {\n" +  
 " return {\n" +  
 " statusCode: 200,\n" +  
 " body: JSON.stringify('Hello World!'),\n" +  
 " };\n" +  
 "};"))  
 .environment(Map.of(  
 "DATABASE\_CONNECTION\_STRING", connectionString // Using the port token as part of the string  
 ))  
 .build();  
  
// Output the value of our connection string at synthesis  
System.out.println("connectionString: " + connectionString);  
  
// Output the connection string  
CfnOutput.Builder.create(this, "ConnectionString")  
 .value(connectionString)  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — // … using Amazon.CDK.AWS.Lambda; using Amazon.CDK.AWS.RDS; using Amazon.CDK; using Constructs; using System; using System.Collections.Generic;

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define a new VPC // …

…. // Define an RDS database cluster var dbCluster = new DatabaseCluster(this, “MyRDSCluster”, new DatabaseClusterProps { // … properties would go here });

// Get the port token (this is a token encoded as a number)  
 var portToken = dbCluster.ClusterEndpoint.Port;  
  
 // ...  
  
 // Example connection string with the port token as a number  
 var connectionString = $"jdbc:mysql://mydb.cluster.amazonaws.com:{portToken}/mydatabase";  
  
 // Use the connection string as an environment variable in a Lambda function  
 var myFunction = new Function(this, "MyLambdaFunction", new FunctionProps  
 {  
 Runtime = Runtime.NODEJS\_20\_X,  
 Handler = "index.handler",  
 Code = Code.FromInline(@"  
 exports.handler = async function(event) {  
 return {  
 statusCode: 200,  
 body: JSON.stringify('Hello World!'),  
 };  
 };  
 "),  
 Environment = new Dictionary<string, string>  
 {  
 { "DATABASE\_CONNECTION\_STRING", connectionString } // Using the port token as part of the string  
 }  
 });  
  
 // Output the value of our connection string at synthesis  
 Console.WriteLine($"connectionString: {connectionString}");  
  
 // Output the connection string  
 new CfnOutput(this, "ConnectionString", new CfnOutputProps  
 {  
 Value = connectionString  
 });  
}

} } ….

’’’

Go:: + [source,go,subs=“verbatim,attributes”] — // … “github.com/aws/aws-cdk-go/awscdk/v2/awslambda” )

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)

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) portToken := dbCluster.ClusterEndpoint().Port()

// …

// Example connection string with the port token as a number connectionString := fmt.Sprintf(“jdbc:mysql://mydb.cluster.amazonaws.com:%s/mydatabase”, portToken)

// Use the connection string as an environment variable in a Lambda function myFunction := awslambda.NewFunction(stack, jsii.String(“MyLambdaFunction”), &awslambda.FunctionProps{ Runtime: awslambda.Runtime\_NODEJS\_20\_X(), Handler: jsii.String(“index.handler”), Code: awslambda.Code\_FromInline(jsii.String(exports.handler = async function(event) { return { statusCode: 200, body: JSON.stringify('Hello World!'), }; };)), Environment: &map[string]\*string{ “DATABASE\_CONNECTION\_STRING”: jsii.String(connectionString), // Using the port token as part of the string }, })

// Output the value of our connection string at synthesis fmt.Println(“connectionString:”, connectionString)

// Output the connection string awscdk.NewCfnOutput(stack, jsii.String(“ConnectionString”), &awscdk.CfnOutputProps{ Value: jsii.String(connectionString), })

return stack } ….

== // …

====

If we pass this value to connectionString, the output value when we run cdk synth may be confusing due to the number-encoded string:

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

$ cdk synth –quiet connectionString: jdbc:mysql://mydb.cluster.amazonaws.com:-1.888154589708796e+289/mydatabase —

To convert a number-encoded token to a string, use https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tokenization.html#static-stringifywbrnumberx[cdk.Tokenization.stringifyNumber(<token>)]. In the following example, we convert the number-encoded token to a string before defining our connection string:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { Stack, Duration, Tokenization, CfnOutput, StackProps } from ‘aws-cdk-lib’; // …

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

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// …

// Convert the encoded number to an encoded string for use in the connection string const portAsString = Tokenization.stringifyNumber(portToken);

// Example connection string with the port token as a string const connectionString = jdbc:mysql://mydb.cluster.amazonaws.com:${portAsString}/mydatabase;

// Use the connection string as an environment variable in a Lambda function const myFunction = new lambda.Function(this, ‘MyLambdaFunction’, { // … environment: { DATABASE\_CONNECTION\_STRING: connectionString, // Using the port token as part of the string }, });

// Output the value of our connection string at synthesis console.log(“connectionString:” + connectionString);

// Output the connection string new CfnOutput(this, ‘ConnectionString’, { value: connectionString, }); } } —- ….

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

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

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) const portToken = dbCluster.clusterEndpoint.port;

// …

// Convert the encoded number to an encoded string for use in the connection string const portAsString = Tokenization.stringifyNumber(portToken);

// Example connection string with the port token as a string const connectionString = jdbc:mysql://mydb.cluster.amazonaws.com:${portAsString}/mydatabase;

// Use the connection string as an environment variable in a Lambda function const myFunction = new lambda.Function(this, ‘MyLambdaFunction’, { // … environment: { DATABASE\_CONNECTION\_STRING: connectionString, // Using the port token as part of the string }, });

// Output the value of our connection string at synthesis console.log(“connectionString:” + connectionString);

// Output the connection string new CfnOutput(this, ‘ConnectionString’, { value: connectionString, }); } } ….

== module.exports = { CdkDemoAppStack }

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

= …

class CdkDemoAppStack(Stack):

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

# Define a new VPC  
# ...  
  
# Define an RDS database cluster  
# ...  
  
# Get the port token (this is a token encoded as a number)  
port\_token = db\_cluster.cluster\_endpoint.port  
  
# Convert the encoded number to an encoded string for use in the connection string  
port\_as\_string = Tokenization.stringify\_number(port\_token)  
  
# Example connection string with the port token as a string  
connection\_string = f"jdbc:mysql://mydb.cluster.amazonaws.com:{port\_as\_string}/mydatabase"  
  
# Use the connection string as an environment variable in a Lambda function  
my\_function = \_lambda.Function(self, 'MyLambdaFunction',  
 # ...  
 environment={  
 'DATABASE\_CONNECTION\_STRING': connection\_string # Using the port token as part of the string  
 }  
)  
  
# Output the value of our connection string at synthesis  
print(f"connectionString: {connection\_string}")  
  
# Output the connection string  
CfnOutput(self, 'ConnectionString',  
 value=connection\_string  
) ----

….

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

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

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

// Define a new VPC  
// ...  
  
// Define an RDS database cluster  
// ...  
  
// Get the port token (this is a token encoded as a number)  
Number portToken = dbCluster.getClusterEndpoint().getPort();  
  
// ...  
  
// Convert the encoded number to an encoded string for use in the connection string  
String portAsString = Tokenization.stringifyNumber(portToken);  
  
// Example connection string with the port token as a string  
String connectionString = "jdbc:mysql://mydb.cluster.amazonaws.com:" + portAsString + "/mydatabase";  
  
// Use the connection string as an environment variable in a Lambda function  
Function myFunction = Function.Builder.create(this, "MyLambdaFunction")  
 // ...  
 .environment(Map.of(  
 "DATABASE\_CONNECTION\_STRING", connectionString // Using the port token as part of the string  
 ))  
 .build();  
  
// Output the value of our connection string at synthesis  
System.out.println("connectionString: " + connectionString);  
  
// Output the connection string  
CfnOutput.Builder.create(this, "ConnectionString")  
 .value(connectionString)  
 .build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — // …

namespace CdkDemoApp { public class CdkDemoAppStack : Stack { internal CdkDemoAppStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { // Define a new VPC // …

…. // Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number)  
 var portToken = dbCluster.ClusterEndpoint.Port;  
  
 // ...  
  
 // Convert the encoded number to an encoded string for use in the connection string  
 var portAsString = Tokenization.StringifyNumber(portToken);  
  
 // Example connection string with the port token as a string  
 var connectionString = $"jdbc:mysql://mydb.cluster.amazonaws.com:{portAsString}/mydatabase";  
  
 // Use the connection string as an environment variable in a Lambda function  
 var myFunction = new Function(this, "MyLambdaFunction", new FunctionProps  
 {  
 // ...  
 Environment = new Dictionary<string, string>  
 {  
 { "DATABASE\_CONNECTION\_STRING", connectionString } // Using the port token as part of the string  
 }  
 });  
  
 // Output the value of our connection string at synthesis  
 Console.WriteLine($"connectionString: {connectionString}");  
  
 // Output the connection string  
 new CfnOutput(this, "ConnectionString", new CfnOutputProps  
 {  
 Value = connectionString  
 });  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — // …

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)

…. // Define a new VPC // …

// Define an RDS database cluster // …

// Get the port token (this is a token encoded as a number) portToken := dbCluster.ClusterEndpoint().Port()

// …

// Convert the encoded number to an encoded string for use in the connection string portAsString := awscdk.Tokenization\_StringifyNumber(portToken)

// Example connection string with the port token as a string connectionString := fmt.Sprintf(“jdbc:mysql://mydb.cluster.amazonaws.com:%s/mydatabase”, portAsString)

// Use the connection string as an environment variable in a Lambda function myFunction := awslambda.NewFunction(stack, jsii.String(“MyLambdaFunction”), &awslambda.FunctionProps{ // … Environment: &map[string]\*string{ “DATABASE\_CONNECTION\_STRING”: jsii.String(connectionString), // Using the port token as part of the string }, })

// Output the value of our connection string at synthesis fmt.Println(“connectionString:”, connectionString)

// Output the connection string awscdk.NewCfnOutput(stack, jsii.String(“ConnectionString”), &awscdk.CfnOutputProps{ Value: jsii.String(connectionString), })

fmt.Println(myFunction)

return stack } ….

== // …

====

When we run cdk synth, the value for our connection string is represented in a cleaner and clearer format:

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

$ cdk synth –quiet connectionString: jdbc:mysql://mydb.cluster.amazonaws.com:${Token[TOKEN.242]}/mydatabase —

[#tokens-lazy] == Lazy values

In addition to representing deploy-time values, such as {aws} CloudFormation xref:parameters[parameters], tokens are also commonly used to represent synthesis-time lazy values. These are values for which the final value will be determined before synthesis has completed, but not at the point where the value is constructed. Use tokens to pass a literal string or number value to another construct, while the actual value at synthesis time might depend on some calculation that has yet to occur.

You can construct tokens representing synth-time lazy values using static methods on the Lazy class, such as https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Lazy.html#static-stringproducer-options[Lazy.string] and https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Lazy.html#static-numberproducer[Lazy.number]. These methods accept an object whose produce property is a function that accepts a context argument and returns the final value when called.

The following example creates an Auto Scaling group whose capacity is determined after its creation.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — let actualValue: number;

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

// At some later point actualValue = 10; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — let actualValue;

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

// At some later point actualValue = 10; —

Python:: + [source,python,subs=“verbatim,attributes”] — class Producer: def *init*(self, func): self.produce = func

actual\_value = None

AutoScalingGroup(self, “Group”, desired\_capacity=Lazy.number\_value(Producer(lambda context: actual\_value)) )

= At some later point

## actual\_value = 10

Java:: + [source,java,subs=“verbatim,attributes”] — double actualValue = 0;

class ProduceActualValue implements INumberProducer {

@Override public Number produce(IResolveContext context) { return actualValue; } }

AutoScalingGroup.Builder.create(this, “Group”) .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point actualValue = 10; —

C#:: + [source,csharp,subs=“verbatim,attributes”] — public class NumberProducer : INumberProducer { Func++++++function;++++++

…. public NumberProducer(Func function) { this.function = function; }

public Double Produce(IResolveContext context) { return function(); } } ….

double actualValue = 0;

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

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

[#tokens-json] == Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html#towbrjsonwbrstringobj-space[stack.toJsonString], as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const stack = Stack.of(this); const str = stack.toJsonString({ value: bucket.bucketName }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const stack = Stack.of(this); const str = stack.toJsonString({ value: bucket.bucketName }); —

Python:: + [source,python,subs=“verbatim,attributes”] — stack = Stack.of(self) string = stack.to\_json\_string(dict(value=bucket.bucket\_name)) —

Java:: + [source,java,subs=“verbatim,attributes”] — Stack stack = Stack.of(this); String stringVal = stack.toJsonString(java.util.Map.of( // Map.of requires Java 9+ put(“value”, bucket.getBucketName()))); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var stack = Stack.Of(this); var stringVal = stack.ToJsonString(new Dictionary<string, string> { [“value”] = bucket.BucketName }); — ====

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Apps :keywords: {aws} CDK, {aws} CDK app

[#apps] = {aws} CDK apps

== [abstract]

## The {aws} Cloud Development Kit ({aws} CDK) application or *app* is a collection of one or more CDK xref:stacks[stacks]. Stacks are a collection of one or more xref:constructs[constructs], which define {aws} resources and properties. Therefore, the overall grouping of your stacks and constructs are known as your CDK app.

// Content start

The {aws} Cloud Development Kit ({aws} CDK) application or *app* is a collection of one or more CDK xref:stacks[stacks]. Stacks are a collection of one or more xref:constructs[constructs], which define {aws} resources and properties. Therefore, the overall grouping of your stacks and constructs are known as your CDK app.

[#apps-define] == How to create a CDK app

You create an app by defining an app instance in the application file of your xref:projects[project]. To do this, you import and use the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.App.html[App] construct from the {aws} Construct Library. The App construct doesn’t require any initialization arguments. It is the only construct that can be used as the root.

The +https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.App.html[App]+ and +https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack]+ classes from the {aws} Construct Library are unique constructs. Compared to other constructs, they don’t configure {aws} resources on their own. Instead, they are used to provide context for your other constructs. All constructs that represent {aws} resources must be defined, directly or indirectly, within the scope of a Stack construct. Stack constructs are defined within the scope of an App construct.

Apps are then synthesized to create {aws} CloudFormation templates for your stacks. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new App(); new MyFirstStack(app, ‘hello-cdk’); app.synth(); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const app = new App(); new MyFirstStack(app, ‘hello-cdk’); app.synth(); —

Python:: + [source,python,subs=“verbatim,attributes”] — app = App() MyFirstStack(app, “hello-cdk”) app.synth() —

Java:: + [source,java,subs=“verbatim,attributes”] — App app = new App(); new MyFirstStack(app, “hello-cdk”); app.synth(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var app = new App(); new MyFirstStack(app, “hello-cdk”); app.Synth(); —

Go:: + [source,go,subs=“verbatim,attributes”] — app := awscdk.NewApp(nil)

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

== app.Synth(nil)

====

Stacks within a single app can easily refer to each other’s resources and properties. The {aws} CDK infers dependencies between stacks so that they can be deployed in the correct order. You can deploy any or all of the stacks within an app with a single cdk deploy command.

[#apps-tree] == The construct tree

Constructs are defined inside of other constructs using the scope argument that is passed to every construct, with the App class as the root. In this way, an {aws} CDK app defines a hierarchy of constructs known as the *construct tree*.

The root of this tree is your app, which is an instance of the App class. Within the app, you instantiate one or more stacks. Within stacks, you instantiate constructs, which may themselves instantiate resources or other constructs, and so on down the tree.

Constructs are *always* explicitly defined within the scope of another construct, which creates relationships between constructs. Almost always, you should pass this (in Python, self) as the scope, indicating that the new construct is a child of the current construct. The intended pattern is that you derive your construct from https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct], then instantiate the constructs it uses in its constructor.

Passing the scope explicitly allows each construct to add itself to the tree, with this behavior entirely contained within the https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct base class]. It works the same way in every language supported by the {aws} CDK and does not require additional customization.

= [IMPORTANT]

Technically, it’s possible to pass some scope other than this when instantiating a construct. You can add constructs anywhere in the tree, or even in another stack in the same app. For example, you could write a mixin-style function that adds constructs to a scope passed in as an argument. The practical difficulty here is that you can’t easily ensure that the IDs you choose for your constructs are unique within someone else’s scope. This practice also makes your code more difficult to understand, maintain, and reuse. Therefore, we recommend that you use the general structure of the construct tree.

====

The {aws} CDK uses the IDs of all constructs in the path from the tree’s root to each child construct to generate the unique IDs required by {aws} CloudFormation. This approach means that construct IDs only need to be unique within their scope, rather than within the entire stack as in native {aws} CloudFormation. However, if you move a construct to a different scope, its generated stack-unique ID changes, and {aws} CloudFormation won’t consider it the same resource.

The construct tree is separate from the constructs that you define in your {aws} CDK code. However, it’s accessible through any construct’s node attribute, which is a reference to the node that represents that construct in the tree. Each node is a https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Node.html[Node] instance, the attributes of which provide access to the tree’s root and to the node’s parent scopes and children.

. node.children – The direct children of the construct. . node.id – The identifier of the construct within its scope. . node.path – The full path of the construct including the IDs of all of its parents. . node.root – The root of the construct tree (the app). . node.scope – The scope (parent) of the construct, or undefined if the node is the root. . node.scopes – All parents of the construct, up to the root. . node.uniqueId – The unique alphanumeric identifier for this construct within the tree (by default, generated from node.path and a hash).

The construct tree defines an implicit order in which constructs are synthesized to resources in the final {aws} CloudFormation template. Where one resource must be created before another, {aws} CloudFormation or the {aws} Construct Library generally infers the dependency. They then make sure that the resources are created in the right order.

You can also add an explicit dependency between two nodes by using node.addDependency(). For more information, see https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib-readme.html#dependencies[Dependencies] in the *{aws} CDK API Reference*.

The {aws} CDK provides a simple way to visit every node in the construct tree and perform an operation on each one. For more information, see xref:aspects[Aspects and the {aws} CDK].

:doctype: book

include::attributes.txt[]

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

[#constructs] = {aws} CDK Constructs

== [abstract]

## Constructs are the basic building blocks of {aws} Cloud Development Kit ({aws} CDK) applications. A construct is a component within your application that represents one or more {aws} CloudFormation resources and their configuration. You build your application, piece by piece, by importing and configuring constructs.

// Content start

Constructs are the basic building blocks of {aws} Cloud Development Kit ({aws} CDK) applications. A construct is a component within your application that represents one or more {aws} CloudFormation resources and their configuration. You build your application, piece by piece, by importing and configuring constructs.

[#constructs-import] == Import and use constructs

Constructs are classes that you import into your CDK applications from the xref:libraries-construct[{aws} Construct Library]. You can also create and distribute your own constructs, or use constructs created by third-party developers.

Constructs are part of the Construct Programming Model (CPM). They are available to use with other tools such as CDK for [.noloc]Terraform (CDKtf), CDK for [.noloc]Kubernetes (CDK8s), and [.noloc]Projen.

Numerous third parties have also published constructs compatible with the {aws} CDK. Visit https://constructs.dev/search?q=&cdk=aws-cdk&cdkver=2&offset=0[Construct Hub] to explore the {aws} CDK construct partner ecosystem.

[#constructs-lib-levels] == Construct levels

Constructs from the {aws} Construct Library are categorized into three levels. Each level offers an increasing level of abstraction. The higher the abstraction, the easier to configure, requiring less expertise. The lower the abstraction, the more customization available, requiring more expertise.

[#constructs-lib-levels-one] *Level 1 (L1) constructs*:: L1 constructs, also known as *CFN resources*, are the lowest-level construct and offer no abstraction. Each L1 construct maps directly to a single {aws} CloudFormation resource. With L1 constructs, you import a construct that represents a specific {aws} CloudFormation resource. You then define the resource+'s properties within your construct instance. + L1 constructs are great to use when you are familiar with {aws} CloudFormation and need complete control over defining your {aws} resource properties. + In the {aws} Construct Library, L1 constructs are named starting with +Cfn+, followed by an identifier for the {aws} CloudFormation resource that it represents. For example, the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.CfnBucket.html[+CfnBucket+] construct is an L1 construct that represents an https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[+{aws}::S3::Bucket`] {aws} CloudFormation resource. + L1 constructs are generated from the https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-specification.html[{aws} CloudFormation resource specification]. If a resource exists in {aws} CloudFormation, it’ll be available in the {aws} CDK as an L1 construct. New resources or properties may take up to a week to become available in the {aws} Construct Library. For more information, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html[{aws} resource and property types reference] in the *{aws} CloudFormation User Guide*.

[#constructs-lib-levels-two] *Level 2 (L2) constructs*:: L2 constructs, also known as *curated* constructs, are thoughtfully developed by the CDK team and are usually the most widely used construct type. L2 constructs map directly to single {aws} CloudFormation resources, similar to L1 constructs. Compared to L1 constructs, L2 constructs provide a higher-level abstraction through an intuitive intent-based API. L2 constructs include sensible default property configurations, best practice security policies, and generate a lot of the boilerplate code and glue logic for you. + L2 constructs also provide helper methods for most resources that make it simpler and quicker to define properties, permissions, event-based interactions between resources, and more. + The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[s3.Bucket] class is an example of an L2 construct for an Amazon Simple Storage Service (Amazon S3) bucket resource. + The {aws} Construct Library contains L2 constructs that are designated stable and ready for production use. For L2 constructs under development, they are designated as experimental and offered in a separate module.

[#constructs-lib-levels-three] *Level 3 (L3) constructs*:: L3 constructs, also known as *patterns*, are the highest-level of abstraction. Each L3 construct can contain a collection of resources that are configured to work together to accomplish a specific task or service within your application. L3 constructs are used to create entire {aws} architectures for particular use cases in your application. + To provide complete system designs, or substantial parts of a larger system, L3 constructs offer opinionated default property configurations. They are built around a particular approach toward solving a problem and providing a solution. With L3 constructs, you can create and configure multiple resources quickly, with the fewest amount of input and code. + The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ecs\_patterns.ApplicationLoadBalancedFargateService.html[ecsPatterns.ApplicationLoadBalancedFargateService] class is an example of an L3 construct that represents an {aws} Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster and fronted by an application load balancer. + Similar to L2 constructs, L3 constructs that are ready for production use are included in the {aws} Construct Library. Those under development are offered in separate modules.

[#constructs-define,constructs-define.title] == Defining constructs

[#constructs-composition] === Composition

*Composition* is the key pattern for defining higher-level abstractions through constructs. A high-level construct can be composed from any number of lower-level constructs. From a bottom-up perspective, you use constructs to organize the individual {aws} resources that you want to deploy. You use whatever abstractions are convenient for your purpose, with as many levels as you need.

With composition, you define reusable components and share them like any other code. For example, a team can define a construct that implements the company`’s best practice for an Amazon DynamoDB table, including backup, global replication, automatic scaling, and monitoring. The team can share the construct internally with other teams, or publicly.

Teams can use constructs like any other library package. When the library is updated, developers get access to the new version’s improvements and bug fixes, similar to any other code library.

[#constructs-init] === Initialization

Constructs are implemented in classes that extend the https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct] base class. You define a construct by instantiating the class. All constructs take three parameters when they are initialized:

* *scope* – The construct’s parent or owner. This can either be a stack or another construct. Scope determines the construct’s place in the xref:apps-tree[construct tree]. You should usually pass this (self in Python), which represents the current object, for the scope.
* *id* – An xref:identifiers[identifier] that must be unique within the scope. The identifier serves as a namespace for everything that’s defined within the construct. It’s used to generate unique identifiers, such as xref:resources-physical-names[resource names] and {aws} CloudFormation logical IDs.
* Identifiers need only be unique within a scope. This lets you instantiate and reuse constructs without concern for the constructs and identifiers they might contain, and enables composing constructs into higher-level abstractions. In addition, scopes make it possible to refer to groups of constructs all at once. Examples include for https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Tag.html[tagging], or specifying where the constructs will be deployed.
* *props* – A set of properties or keyword arguments, depending on the language, that define the construct`’s initial configuration. Higher-level constructs provide more defaults, and if all prop elements are optional, you can omit the props parameter completely.

[#constructs-config] === Configuration

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new s3.Bucket(this, ‘MyEncryptedBucket’, { encryption: s3.BucketEncryption.KMS, websiteIndexDocument: ‘index.html’ }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new s3.Bucket(this, ‘MyEncryptedBucket’, { encryption: s3.BucketEncryption.KMS, websiteIndexDocument: ‘index.html’ }); —

Python:: + [source,python,subs=“verbatim,attributes”] — s3.Bucket(self, “MyEncryptedBucket”, encryption=s3.BucketEncryption.KMS, website\_index\_document=“index.html”) —

Java:: + [source,java,subs=“verbatim,attributes”] — Bucket.Builder.create(this, “MyEncryptedBucket”) .encryption(BucketEncryption.KMS\_MANAGED) .websiteIndexDocument(“index.html”).build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new Bucket(this, “MyEncryptedBucket”, new BucketProps { Encryption = BucketEncryption.KMS\_MANAGED, WebsiteIndexDocument = “index.html” }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awss3.NewBucket(stack, jsii.String(“MyEncryptedBucket”), &awss3.BucketProps{ Encryption: awss3.BucketEncryption\_KMS, WebsiteIndexDocument: jsii.String(“index.html”), }) — ====

[#constructs-interact] === Interacting with constructs

Constructs are classes that extend the base https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct] class. After you instantiate a construct, the construct object exposes a set of methods and properties that let you interact with the construct and pass it around as a reference to other parts of the system.

The {aws} CDK framework doesn’t put any restrictions on the APIs of constructs. Authors can define any API they want. However, the {aws} constructs that are included with the {aws} Construct Library, such as s3.Bucket, follow guidelines and common patterns. This provides a consistent experience across all {aws} resources.

Most {aws} constructs have a set of xref:permissions-grants[grant] methods that you can use to grant {aws} Identity and Access Management (IAM) permissions on that construct to a principal. The following example grants the IAM group data-science permission to read from the Amazon S3 bucket raw-data.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const rawData = new s3.Bucket(this, ‘raw-data’); const dataScience = new iam.Group(this, ‘data-science’); rawData.grantRead(dataScience); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const rawData = new s3.Bucket(this, ‘raw-data’); const dataScience = new iam.Group(this, ‘data-science’); rawData.grantRead(dataScience); —

Python:: + [source,python,subs=“verbatim,attributes”] — raw\_data = s3.Bucket(self, ‘raw-data’) data\_science = iam.Group(self, ‘data-science’) raw\_data.grant\_read(data\_science) —

Java:: + [source,java,subs=“verbatim,attributes”] — Bucket rawData = new Bucket(this, “raw-data”); Group dataScience = new Group(this, “data-science”); rawData.grantRead(dataScience); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var rawData = new Bucket(this, “raw-data”); var dataScience = new Group(this, “data-science”); rawData.GrantRead(dataScience); —

Go:: + [source,go,subs=“verbatim,attributes”] — rawData := awss3.NewBucket(stack, jsii.String(“raw-data”), nil) dataScience := awsiam.NewGroup(stack, jsii.String(“data-science”), nil) rawData.GrantRead(dataScience, nil) — ====

Another common pattern is for {aws} constructs to set one of the resource’s attributes from data supplied elsewhere. Attributes can include Amazon Resource Names (ARNs), names, or URLs.

The following code defines an {aws} Lambda function and associates it with an Amazon Simple Queue Service (Amazon SQS) queue through the queue’s URL in an environment variable.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const jobsQueue = new sqs.Queue(this, ‘jobs’); const createJobLambda = new lambda.Function(this, ‘create-job’, { runtime: lambda.Runtime.NODEJS\_18\_X, handler: ‘index.handler’, code: lambda.Code.fromAsset(‘./create-job-lambda-code’), environment: { QUEUE\_URL: jobsQueue.queueUrl } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const jobsQueue = new sqs.Queue(this, ‘jobs’); const createJobLambda = new lambda.Function(this, ‘create-job’, { runtime: lambda.Runtime.NODEJS\_18\_X, handler: ‘index.handler’, code: lambda.Code.fromAsset(‘./create-job-lambda-code’), environment: { QUEUE\_URL: jobsQueue.queueUrl } }); —

Python:: + [source,python,subs=“verbatim,attributes”] — jobs\_queue = sqs.Queue(self, “jobs”) create\_job\_lambda = lambda\_.Function(self, “create-job”, runtime=lambda\_.Runtime.NODEJS\_18\_\_X, handler=“index.handler”, code=lambda\_\_.Code.from\_asset(“./create-job-lambda-code”), environment=dict( QUEUE\_URL=jobs\_queue.queue\_url ) ) —

Java:: + [source,java,subs=“verbatim,attributes”] — final Queue jobsQueue = new Queue(this, “jobs”); Function createJobLambda = Function.Builder.create(this, “create-job”) .handler(“index.handler”) .code(Code.fromAsset(“./create-job-lambda-code”)) .environment(java.util.Map.of( // Map.of is Java 9 or later “QUEUE\_URL”, jobsQueue.getQueueUrl())) .build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var jobsQueue = new Queue(this, “jobs”); var createJobLambda = new Function(this, “create-job”, new FunctionProps { Runtime = Runtime.NODEJS\_18\_X, Handler = “index.handler”, Code = Code.FromAsset(@“.-job-lambda-code”), Environment = new Dictionary<string, string> { [“QUEUE\_URL”] = jobsQueue.QueueUrl } }); —

Go:: + [source,go,subs=“verbatim,attributes”] — createJobLambda := awslambda.NewFunction(stack, jsii.String(“create-job”), &awslambda.FunctionProps{ Runtime: awslambda.Runtime\_NODEJS\_18\_X(), Handler: jsii.String(“index.handler”), Code: awslambda.Code\_FromAsset(jsii.String(“.-job-lambda-code”), nil), Environment: &map[string]\_\_string{ “QUEUE\_URL”: jsii.String(\_\_jobsQueue.QueueUrl()), }, }) — ====

For information about the most common API patterns in the {aws} Construct Library, see xref:resources[Resources and the {aws} CDK].

[#constructs-apps-stacks] === The app and stack construct

The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.App.html[App] and https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html[Stack] classes from the {aws} Construct Library are unique constructs. Compared to other constructs, they don’t configure {aws} resources on their own. Instead, they are used to provide context for your other constructs. All constructs that represent {aws} resources must be defined, directly or indirectly, within the scope of a Stack construct. Stack constructs are defined within the scope of an App construct.

To learn more about CDK apps, see xref:apps[{aws} CDK apps]. To learn more about CDK stacks, see xref:stacks[Introduction to {aws} CDK stacks].

The following example defines an app with a single stack. Within the stack, an L2 construct is used to configure an Amazon S3 bucket resource.

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

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

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

const app = new App(); new HelloCdkStack(app, “HelloCdkStack”); —

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

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

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

const app = new App(); new HelloCdkStack(app, “HelloCdkStack”); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import App, Stack import aws\_cdk.aws\_s3 as s3 from constructs import Construct

class HelloCdkStack(Stack):

…. def **init**(self, scope: Construct, id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, id, \*\*kwargs)

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

….

app = App() HelloCdkStack(app, “HelloCdkStack”) —

Java:: Stack defined in HelloCdkStack.java file: + [source,java,subs=“verbatim,attributes”] — import software.constructs.Construct; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.amazon.awscdk.services.s3.\*;

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

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

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

} } —- + ….

App defined in HelloCdkApp.java file: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.App; import software.amazon.awscdk.StackProps;

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

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

app.synth();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.S3;

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

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

Go:: + [source,go,subs=“verbatim,attributes”] — func NewHelloCdkStack(scope constructs.Construct, id string, props \*HelloCdkStackProps) awscdk.Stack { var sprops awscdk.StackProps if props != nil { sprops = props.StackProps } stack := awscdk.NewStack(scope, &id, &sprops)

…. awss3.NewBucket(stack, jsii.String(“MyFirstBucket”), &awss3.BucketProps{ Versioned: jsii.Bool(true), })

return stack } —- ==== ….

[#constructs-work] == Working with constructs

[#constructs-l1-using] === Working with L1 constructs

L1 constructs map directly to individual {aws} CloudFormation resources. You must provide the resource’s required configuration.

In this example, we create a bucket object using the CfnBucket L1 construct:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new s3.CfnBucket(this, “amzn-s3-demo-bucket”, { bucketName: “amzn-s3-demo-bucket” }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new s3.CfnBucket(this, “amzn-s3-demo-bucket”, { bucketName: “amzn-s3-demo-bucket” }); —

Python:: + [source,python,subs=“verbatim,attributes”] — bucket = s3.CfnBucket(self, “amzn-s3-demo-bucket”, bucket\_name=“amzn-s3-demo-bucket”) —

Java:: + [source,java,subs=“verbatim,attributes”] — CfnBucket bucket = new CfnBucket.Builder().bucketName(“amzn-s3-demo-bucket”).build(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var bucket = new CfnBucket(this, “amzn-s3-demo-bucket”, new CfnBucketProps { BucketName= “amzn-s3-demo-bucket” }); —

Go:: + [source,go,subs=“verbatim,attributes”] — awss3.NewCfnBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.CfnBucketProps{ BucketName: jsii.String(“amzn-s3-demo-bucket”), }) — ====

Construct properties that aren’t simple Booleans, strings, numbers, or containers are handled differently in the supported languages.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new s3.CfnBucket(this, “amzn-s3-demo-bucket”, { bucketName: “amzn-s3-demo-bucket”, corsConfiguration: { corsRules: [{ allowedOrigins: [“\*”], allowedMethods: [“GET”] }] } }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const bucket = new s3.CfnBucket(this, “amzn-s3-demo-bucket”, { bucketName: “amzn-s3-demo-bucket”, corsConfiguration: { corsRules: [{ allowedOrigins: [“\*”], allowedMethods: [“GET”] }] } }); —

Python:: In Python, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property cors\_configuration of a CfnBucket requires a wrapper of type CfnBucket.CorsConfigurationProperty. Here we are defining cors\_configuration on a CfnBucket instance. + [source,python,subs=“verbatim,attributes”] — bucket = CfnBucket(self, “amzn-s3-demo-bucket”, bucket\_name=“amzn-s3-demo-bucket”, cors\_configuration=CfnBucket.CorsConfigurationProperty( cors\_rules=[CfnBucket.CorsRuleProperty( allowed\_origins=[“\*”], allowed\_methods=[“GET”] )] ) ) —

Java:: In Java, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property corsConfiguration of a CfnBucket requires a wrapper of type CfnBucket.CorsConfigurationProperty. Here we are defining corsConfiguration on a CfnBucket instance. + [source,java,subs=“verbatim,attributes”] — CfnBucket bucket = CfnBucket.Builder.create(this, “amzn-s3-demo-bucket”) .bucketName(“amzn-s3-demo-bucket”) .corsConfiguration(new CfnBucket.CorsConfigurationProperty.Builder() .corsRules(Arrays.asList(new CfnBucket.CorsRuleProperty.Builder() .allowedOrigins(Arrays.asList(“\*“)) .allowedMethods(Arrays.asList(”GET”)) .build())) .build()) .build(); —

C#:: In C#, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property CorsConfiguration of a CfnBucket requires a wrapper of type CfnBucket.CorsConfigurationProperty. Here we are defining CorsConfiguration on a CfnBucket instance. + [source,csharp,subs=“verbatim,attributes”] — var bucket = new CfnBucket(this, “amzn-s3-demo-bucket”, new CfnBucketProps { BucketName = “amzn-s3-demo-bucket”, CorsConfiguration = new CfnBucket.CorsConfigurationProperty { CorsRules = new object[] { new CfnBucket.CorsRuleProperty { AllowedOrigins = new string[] { “\*” }, AllowedMethods = new string[] { “GET” }, } } } }); —

Go:: In Go, these types are named using the name of the L1 construct, an underscore, and the property name. For example, the optional property CorsConfiguration of a CfnBucket requires a wrapper of type CfnBucket\_CorsConfigurationProperty. Here we are defining CorsConfiguration on a CfnBucket instance. + [source,go,subs=“verbatim,attributes”] — awss3.NewCfnBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.CfnBucketProps{ BucketName: jsii.String(“amzn-s3-demo-bucket”), CorsConfiguration: &awss3.CfnBucket\_CorsConfigurationProperty{ CorsRules: []awss3.CorsRule{ awss3.CorsRule{ AllowedOrigins: jsii.Strings(“\*“), AllowedMethods: &[]awss3.HttpMethods{”GET”}, }, }, }, }) — ====

= [IMPORTANT]

You can’t use L2 property types with L1 constructs, or vice versa. When working with L1 constructs, always use the types defined for the L1 construct you’re using. Do not use types from other L1 constructs (some may have the same name, but they are not the same type).

Some of our language-specific API references currently have errors in the paths to L1 property types, or don’t document these classes at all. We hope to fix this soon. In the meantime, remember that such types are always inner classes of the L1 construct they are used with.

====

[#constructs-using] === Working with L2 constructs

In the following example, we define an Amazon S3 bucket by creating an object from the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[Bucket] L2 construct:

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

// “this” is HelloCdkStack new s3.Bucket(this, ‘MyFirstBucket’, { versioned: true }); —

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

// “this” is HelloCdkStack new s3.Bucket(this, ‘MyFirstBucket’, { versioned: true }); —

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk.aws\_s3 as s3

= “self” is HelloCdkStack

## s3.Bucket(self, “MyFirstBucket”, versioned=True)

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.s3.\*;

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

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

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

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.S3;

// “this” is HelloCdkStack new Bucket(this, “MyFirstBucket”, new BucketProps { Versioned = true }); —

Go:: + [source,go,subs=“verbatim,attributes”] — import ( “github.com/aws/aws-cdk-go/awscdk/v2/awss3” “github.com/aws/jsii-runtime-go” )

// stack is HelloCdkStack awss3.NewBucket(stack, jsii.String(“MyFirstBucket”), &awss3.BucketProps{ Versioned: jsii.Bool(true), }) — ====

MyFirstBucket is not the name of the bucket that {aws} CloudFormation creates. It is a logical identifier given to the new construct within the context of your CDK app. The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Resource.html#physicalname[physicalName] value will be used to name the {aws} CloudFormation resource.

[#constructs-work-third] == Working with third-party constructs

https://constructs.dev/search?q=&cdk=aws-cdk&cdkver=2&sort=downloadsDesc&offset=0[Construct Hub] is a resource to help you discover additional constructs from {aws}, third parties, and the open-source CDK community.

[#constructs-author] === Writing your own constructs

In addition to using existing constructs, you can also write your own constructs and let anyone use them in their apps. All constructs are equal in the {aws} CDK. Constructs from the {aws} Construct Library are treated the same as a construct from a third-party library published via [.noloc]NPM, [.noloc]Maven, or [.noloc]PyPI. Constructs published to your company’s internal package repository are also treated in the same way.

To declare a new construct, create a class that extends the https://docs.aws.amazon.com/cdk/api/v2/docs/constructs.Construct.html[Construct] base class, in the constructs package, then follow the pattern for initializer arguments.

The following example shows how to declare a construct that represents an Amazon S3 bucket. The S3 bucket sends an Amazon Simple Notification Service (Amazon SNS) notification every time someone uploads a file into it.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — export interface NotifyingBucketProps { prefix?: string; }

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

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class NotifyingBucket extends Construct { constructor(scope, id, props = {}) { super(scope, id); const bucket = new s3.Bucket(this, ‘bucket’); const topic = new sns.Topic(this, ‘topic’); bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic), { prefix: props.prefix }); } }

== module.exports = { NotifyingBucket }

Python:: + [source,python,subs=“verbatim,attributes”] — class NotifyingBucket(Construct):

def **init**(self, scope: Construct, id: str, \*, prefix=None): super().\_\_init\_\_(scope, id) bucket = s3.Bucket(self, “bucket”) topic = sns.Topic(self, “topic”) bucket.add\_object\_created\_notification(s3notify.SnsDestination(topic), s3.NotificationKeyFilter(prefix=prefix)) —-

Java:: + [source,java,subs=“verbatim,attributes”] — public class NotifyingBucket extends Construct {

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

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

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

public NotifyingBucket(final Construct scope, final String id, final BucketProps props, final String prefix) { super(scope, id);

Bucket bucket = new Bucket(this, "bucket");  
Topic topic = new Topic(this, "topic");  
if (prefix != null)  
 bucket.addObjectCreatedNotification(new SnsDestination(topic),  
 NotificationKeyFilter.builder().prefix(prefix).build());

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — public class NotifyingBucketProps : BucketProps { public string Prefix { get; set; } }

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

Go:: + [source,go,subs=“verbatim,attributes”] — type NotifyingBucketProps struct { awss3.BucketProps Prefix \*string }

func NewNotifyingBucket(scope constructs.Construct, id \_\_string, props \*NotifyingBucketProps) awss3.Bucket { var bucket awss3.Bucket if props == nil { bucket = awss3.NewBucket(scope, jsii.String(\_\_id+“Bucket”), nil) } else { bucket = awss3.NewBucket(scope, jsii.String(\_\_id+“Bucket”), &props.BucketProps) } topic := awssns.NewTopic(scope, jsii.String(\_\_id+“Topic”), nil) if props == nil { bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic)) } else { bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic), &awss3.NotificationKeyFilter{ Prefix: props.Prefix, }) } return bucket } — ====

= [NOTE]

Our NotifyingBucket construct inherits not from Bucket but rather from Construct. We are using composition, not inheritance, to bundle an Amazon S3 bucket and an Amazon SNS topic together. In general, composition is preferred over inheritance when developing {aws} CDK constructs.

====

The NotifyingBucket constructor has a typical construct signature: scope, id, and props. The last argument, props, is optional (gets the default value {}) because all props are optional. (The base Construct class does not take a props argument.) You could define an instance of this construct in your app without props, for example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new NotifyingBucket(this, ‘MyNotifyingBucket’); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new NotifyingBucket(this, ‘MyNotifyingBucket’); —

Python:: + [source,python,subs=“verbatim,attributes”] — NotifyingBucket(self, “MyNotifyingBucket”) —

Java:: + [source,java,subs=“verbatim,attributes”] — new NotifyingBucket(this, “MyNotifyingBucket”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new NotifyingBucket(this, “MyNotifyingBucket”); —

Go:: + [source,go,subs=“verbatim,attributes”] — NewNotifyingBucket(stack, jsii.String(“MyNotifyingBucket”), nil) — ====

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — new NotifyingBucket(this, ‘MyNotifyingBucket’, { prefix: ‘images/’ }); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — new NotifyingBucket(this, ‘MyNotifyingBucket’, { prefix: ‘images/’ }); —

Python:: + [source,python,subs=“verbatim,attributes”] — NotifyingBucket(self, “MyNotifyingBucket”, prefix=“images/”) —

Java:: + [source,java,subs=“verbatim,attributes”] — new NotifyingBucket(this, “MyNotifyingBucket”, “/images”); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — new NotifyingBucket(this, “MyNotifyingBucket”, new NotifyingBucketProps { Prefix = “/images” }); —

Go:: + [source,go,subs=“verbatim,attributes”] — NewNotifyingBucket(stack, jsii.String(“MyNotifyingBucket”), &NotifyingBucketProps{ Prefix: jsii.String(“images/”), }) — ====

Typically, you would also want to expose some properties or methods on your constructs. It’s not very useful to have a topic hidden behind your construct, because users of your construct aren’t able to subscribe to it. Adding a topic property lets consumers access the inner topic, as shown in the following example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — export class NotifyingBucket extends Construct { public readonly topic: sns.Topic;

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

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class NotifyingBucket extends Construct {

constructor(scope, id, props) { super(scope, id); const bucket = new s3.Bucket(this, ‘bucket’); this.topic = new sns.Topic(this, ‘topic’); bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic), { prefix: props.prefix }); } }

== module.exports = { NotifyingBucket };

Python:: + [source,python,subs=“verbatim,attributes”] — class NotifyingBucket(Construct):

def **init**(self, scope: Construct, id: str, \*, prefix=None, \*\*kwargs): super().\_\_init\_\_(scope, id) bucket = s3.Bucket(self, “bucket”) self.topic = sns.Topic(self, “topic”) bucket.add\_object\_created\_notification(s3notify.SnsDestination(self.topic), s3.NotificationKeyFilter(prefix=prefix)) —-

Java:: + [source,java,subs=“verbatim,attributes”] — public class NotifyingBucket extends Construct {

…. public Topic topic = null;

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

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

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

public NotifyingBucket(final Construct scope, final String id, final BucketProps props, final String prefix) { super(scope, id);

Bucket bucket = new Bucket(this, "bucket");  
topic = new Topic(this, "topic");  
if (prefix != null)  
 bucket.addObjectCreatedNotification(new SnsDestination(topic),  
 NotificationKeyFilter.builder().prefix(prefix).build());

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — public class NotifyingBucket : Construct { public readonly Topic topic;

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

Go:: To do this in Go, we’ll need a little extra plumbing. Our original NewNotifyingBucket function returned an awss3.Bucket. We’ll need to extend Bucket to include a topic member by creating a NotifyingBucket struct. Our function will then return this type. + [source,go,subs=“verbatim,attributes”] — type NotifyingBucket struct { awss3.Bucket topic awssns.Topic }

func NewNotifyingBucket(scope constructs.Construct, id \_\_string, props \*NotifyingBucketProps) NotifyingBucket { var bucket awss3.Bucket if props == nil { bucket = awss3.NewBucket(scope, jsii.String(\_\_id+“Bucket”), nil) } else { bucket = awss3.NewBucket(scope, jsii.String(\_\_id+“Bucket”), &props.BucketProps) } topic := awssns.NewTopic(scope, jsii.String(\_\_id+“Topic”), nil) if props == nil { bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic)) } else { bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic), &awss3.NotificationKeyFilter{ Prefix: props.Prefix, }) } var nbucket NotifyingBucket nbucket.Bucket = bucket nbucket.topic = topic return nbucket } — ====

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

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const queue = new sqs.Queue(this, ‘NewImagesQueue’); const images = new NotifyingBucket(this, ‘/images’); images.topic.addSubscription(new sns\_sub.SqsSubscription(queue)); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const queue = new sqs.Queue(this, ‘NewImagesQueue’); const images = new NotifyingBucket(this, ‘/images’); images.topic.addSubscription(new sns\_sub.SqsSubscription(queue)); —

Python:: + [source,python,subs=“verbatim,attributes”] — queue = sqs.Queue(self, “NewImagesQueue”) images = NotifyingBucket(self, prefix=“Images”) images.topic.add\_subscription(sns\_sub.SqsSubscription(queue)) —

Java:: + [source,java,subs=“verbatim,attributes”] — NotifyingBucket images = new NotifyingBucket(this, “MyNotifyingBucket”, “/images”); images.topic.addSubscription(new SqsSubscription(queue)); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — var queue = new Queue(this, “NewImagesQueue”); var images = new NotifyingBucket(this, “MyNotifyingBucket”, new NotifyingBucketProps { Prefix = “/images” }); images.topic.AddSubscription(new SqsSubscription(queue)); —

Go:: + [source,go,subs=“verbatim,attributes”] — queue := awssqs.NewQueue(stack, jsii.String(“NewImagesQueue”), nil) images := NewNotifyingBucket(stack, jsii.String(“MyNotifyingBucket”), &NotifyingBucketProps{ Prefix: jsii.String(“/images”), }) images.topic.AddSubscription(awssnssubscriptions.NewSqsSubscription(queue, nil)) — ====

[#constructs-learn] == Learn more

The following video provides a comprehensive overview of CDK constructs, and explains how you can use them in your CDK apps.

video::PzU-i0rJPGw[youtube,align = center,height = 390,fileref = https://www.youtube.com/embed/PzU-i0rJPGw,width = 480]

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Context values :keywords: {aws} CDK, Context values

[#context] = Context values and the {aws} CDK

== [abstract]

## Context values are key-value pairs that can be associated with an app, stack, or construct. They may be supplied to your app from a file (usually either cdk.json or cdk.context.json in your project directory) or on the command line.

// Context start

Context values are key-value pairs that can be associated with an app, stack, or construct. They may be supplied to your app from a file (usually either cdk.json or cdk.context.json in your project directory) or on the command line.

The CDK Toolkit uses context to cache values retrieved from your {aws} account during synthesis. Values include the Availability Zones in your account or the Amazon Machine Image (AMI) IDs currently available for Amazon EC2 instances. Because these values are provided by your {aws} account, they can change between runs of your CDK application. This makes them a potential source of unintended change. The CDK Toolkit’s caching behavior “freezes” these values for your CDK app until you decide to accept the new values.

Imagine the following scenario without context caching. Let’s say you specified “latest Amazon Linux” as the AMI for your Amazon EC2 instances, and a new version of this AMI was released. Then, the next time you deployed your CDK stack, your already-deployed instances would be using the outdated (“wrong”) AMI and would need to be upgraded. Upgrading would result in replacing all your existing instances with new ones, which would probably be unexpected and undesired.

Instead, the CDK records your account’s available AMIs in your project’s cdk.context.json file, and uses the stored value for future synthesis operations. This way, the list of AMIs is no longer a potential source of change. You can also be sure that your stacks will always synthesize to the same {aws} CloudFormation templates.

Not all context values are cached values from your {aws} environment. xref:featureflags[{aws} CDK feature flags] are also context values. You can also create your own context values for use by your apps or constructs.

Context keys are strings. Values may be any type supported by JSON: numbers, strings, arrays, or objects.

= [TIP]

If your constructs create their own context values, incorporate your library’s package name in its keys so they won’t conflict with other packages’ context values.

====

Many context values are associated with a particular {aws} environment, and a given CDK app can be deployed in more than one environment. The key for such values includes the {aws} account and Region so that values from different environments do not conflict.

The following context key illustrates the format used by the {aws} CDK, including the account and Region.

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

## availability-zones:account=123456789012:region=eu-central-1

= [IMPORTANT]

Cached context values are managed by the {aws} CDK and its constructs, including constructs you may write. Do not add or change cached context values by manually editing files. It can be useful, however, to review cdk.context.json occasionally to see what values are being cached. Context values that don’t represent cached values should be stored under the context key of cdk.json. This way, they won’t be cleared when cached values are cleared.

====

[#context-construct] == Sources of context values

Context values can be provided to your {aws} CDK app in six different ways:

* Automatically from the current {aws} account.
* Through the --context option to the cdk command. (These values are always strings.)
* In the project’s cdk.context.json file.
* In the context key of the project’s cdk.json file.
* In the context key of your ~/.cdk.json file.
* In your {aws} CDK app using the construct.node.setContext() method.

The project file cdk.context.json is where the {aws} CDK caches context values retrieved from your {aws} account. This practice avoids unexpected changes to your deployments when, for example, a new Availability Zone is introduced. The {aws} CDK does not write context data to any of the other files listed.

= [IMPORTANT]

Because they’re part of your application’s state, cdk.json and cdk.context.json must be committed to source control along with the rest of your app’s source code. Otherwise, deployments in other environments (for example, a CI pipeline) might produce inconsistent results.

====

Context values are scoped to the construct that created them; they are visible to child constructs, but not to parents or siblings. Context values that are set by the {aws} CDK Toolkit (the cdk command) can be set automatically, from a file, or from the --context option. Context values from these sources are implicitly set on the App construct. Therefore, they’re visible to every construct in every stack in the app.

Your app can read a context value using the construct.node.tryGetContext method. If the requested entry isn’t found on the current construct or any of its parents, the result is undefined. (Alternatively, the result could be your language’s equivalent, such as None in Python.)

[#context-methods] == Context methods

The {aws} CDK supports several context methods that enable {aws} CDK apps to obtain contextual information from the {aws} environment. For example, you can get a list of Availability Zones that are available in a given {aws} account and Region, using the link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html#availabilityzones[stack.availabilityZones] method.

The following are the context methods:

link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_route53.HostedZone.html#static-fromwbrlookupscope-id-query[HostedZone.fromLookup]:: + Gets the hosted zones in your account.

link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.Stack.html#availabilityzones[stack.availabilityZones]:: + Gets the supported Availability Zones.

link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ssm.StringParameter.html#static-valuewbrfromwbrlookupscope-parametername[StringParameter.valueFromLookup]:: + Gets a value from the current Region’s Amazon EC2 Systems Manager Parameter Store.

link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ec2.Vpc.html#static-fromwbrlookupscope-id-options[Vpc.fromLookup]:: + Gets the existing Amazon Virtual Private Clouds in your accounts.

link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_ec2.LookupMachineImage.html[LookupMachineImage]:: + Looks up a machine image for use with a NAT instance in an Amazon Virtual Private Cloud.

If a required context value isn’t available, the {aws} CDK app notifies the CDK Toolkit that the context information is missing. Next, the CLI queries the current {aws} account for the information and stores the resulting context information in the cdk.context.json file. Then, it executes the {aws} CDK app again with the context values.

[#context-viewing] == Viewing and managing context

Use the cdk context command to view and manage the information in your cdk.context.json file. To see this information, use the cdk context command without any options. The output should be something like the following.

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

Context found in cdk.json:

┌───┬─────────────────────────────────────────────────────────────┬─────────────────────────────────────────────────────────┐ │ # │ Key │ Value │ ├───┼─────────────────────────────────────────────────────────────┼─────────────────────────────────────────────────────────┤ │ 1 │ availability-zones:account=123456789012:region=eu-central-1 │ [ “eu-central-1a”, “eu-central-1b”, “eu-central-1c” ] │ ├───┼─────────────────────────────────────────────────────────────┼─────────────────────────────────────────────────────────┤ │ 2 │ availability-zones:account=123456789012:region=eu-west-1 │ [ “eu-west-1a”, “eu-west-1b”, “eu-west-1c” ] │ └───┴─────────────────────────────────────────────────────────────┴─────────────────────────────────────────────────────────┘

Run

cdk context –reset KEY\_OR\_NUMBER

to remove a context key. If it is a cached value, it will be refreshed on the next

cdk synth

== .

To remove a context value, run cdk context --reset, specifying the value’s corresponding key or number. The following example removes the value that corresponds to the second key in the preceding example. This value represents the list of Availability Zones in the Europe (Ireland) Region.

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

## cdk context –reset 2

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

Context value availability-zones:account=123456789012:region=eu-west-1 reset. It will be refreshed on the next SDK synthesis run. —

Therefore, if you want to update to the latest version of the Amazon Linux AMI, use the preceding example to do a controlled update of the context value and reset it. Then, synthesize and deploy your app again.

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

## $ cdk synth

To clear all of the stored context values for your app, run cdk context --clear, as follows.

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

## $ cdk context –clear

Only context values stored in cdk.context.json can be reset or cleared. The {aws} CDK does not touch other context values. Therefore, to protect a context value from being reset using these commands, you might copy the value to cdk.json.

[#context-cli] == {aws} CDK Toolkit --context flag

Use the --context (-c for short) option to pass runtime context values to your CDK app during synthesis or deployment.

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

## $ cdk synth –context key=value MyStack

To specify multiple context values, repeat the --context option any number of times, providing one key-value pair each time.

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

## $ cdk synth –context key1=value1 –context key2=value2 MyStack

When synthesizing multiple stacks, the specified context values are passed to all stacks. To provide different context values to individual stacks, either use different keys for the values, or use multiple cdk synth or cdk deploy commands.

Context values passed from the command line are always strings. If a value is usually of some other type, your code must be prepared to convert or parse the value. You might have non-string context values provided in other ways (for example, in cdk.context.json). To make sure this kind of value works as expected, confirm that the value is a string before converting it.

[#context-example] == Example

Following is an example of using an existing Amazon VPC using {aws} CDK context.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import \* as ec2 from ‘aws-cdk-lib/aws-ec2’; import { Construct } from ‘constructs’;

export class ExistsVpcStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

…. super(scope, id, props);

const vpcid = this.node.tryGetContext(‘vpcid’); const vpc = ec2.Vpc.fromLookup(this, ‘VPC’, { vpcId: vpcid, });

const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

new cdk.CfnOutput(this, ‘publicsubnets’, { value: pubsubnets.subnetIds.toString(), }); } } —- ….

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const ec2 = require(‘aws-cdk-lib/aws-ec2’);

class ExistsVpcStack extends cdk.Stack {

constructor(scope, id, props) {

…. super(scope, id, props);

const vpcid = this.node.tryGetContext(‘vpcid’); const vpc = ec2.Vpc.fromLookup(this, ‘VPC’, { vpcId: vpcid });

const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

new cdk.CfnOutput(this, ‘publicsubnets’, { value: pubsubnets.subnetIds.toString() }); } } ….

== module.exports = { ExistsVpcStack }

Python:: + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk import aws\_cdk.aws\_ec2 as ec2 from constructs import Construct

class ExistsVpcStack(cdk.Stack):

…. def **init**(scope: Construct, id: str, \*\*kwargs):

super().\_\_init\_\_(scope, id, \*\*kwargs)  
  
vpcid = self.node.try\_get\_context("vpcid")  
vpc = ec2.Vpc.from\_lookup(self, "VPC", vpc\_id=vpcid)  
  
pubsubnets = vpc.select\_subnets(subnetType=ec2.SubnetType.PUBLIC)  
  
cdk.CfnOutput(self, "publicsubnets",  
 value=pubsubnets.subnet\_ids.to\_string()) ----

….

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

import software.amazon.awscdk.services.ec2.Vpc; import software.amazon.awscdk.services.ec2.VpcLookupOptions; import software.amazon.awscdk.services.ec2.SelectedSubnets; import software.amazon.awscdk.services.ec2.SubnetSelection; import software.amazon.awscdk.services.ec2.SubnetType; import software.constructs.Construct;

public class ExistsVpcStack extends Stack { public ExistsVpcStack(Construct context, String id) { this(context, id, null); }

…. public ExistsVpcStack(Construct context, String id, StackProps props) { super(context, id, props);

String vpcId = (String)this.getNode().tryGetContext("vpcid");  
Vpc vpc = (Vpc)Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()  
 .vpcId(vpcId).build());  
  
SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()  
 .subnetType(SubnetType.PUBLIC).build());  
  
CfnOutput.Builder.create(this, "publicsubnets")  
 .value(pubSubNets.getSubnetIds().toString()).build();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.EC2; using Constructs;

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

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

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

} } —- ==== ….

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

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

## $ cdk diff -c vpcid=vpc-0cb9c31031d0d3e22

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

Stack ExistsvpcStack Outputs [+] Output publicsubnets publicsubnets: {“Value”:“subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f”} —

The resulting context values can be viewed as shown here.

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

## $ cdk context -j

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

{ “vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-east-1”: { “vpcId”: “vpc-0cb9c31031d0d3e22”, “availabilityZones”: [ “us-east-1a”, “us-east-1b” ], “privateSubnetIds”: [ “subnet-03ecfc033225be285”, “subnet-0cded5da53180ebfa” ], “privateSubnetNames”: [ “Private” ], “privateSubnetRouteTableIds”: [ “rtb-0e955393ced0ada04”, “rtb-05602e7b9f310e5b0” ], “publicSubnetIds”: [ “subnet-06e0ea7dd302d3e8f”, “subnet-01fc0acfb58f3128f” ], “publicSubnetNames”: [ “Public” ], “publicSubnetRouteTableIds”: [ “rtb-00d1fdfd823c82289”, “rtb-04bb1969b42969bcb” ] } } —

include::attributes.txt[]

// Attributes [.topic] [#core-concepts] = Learn {aws} CDK core concepts :info\_titleabbrev: CDK core concepts :keywords: {aws} Cloud Development Kit ({aws} CDK), {aws} CDK, IaC, Infrastructure as code, {aws} CloudFormation, Serverless applications, Modern applications, {aws}

== [abstract]

## Learn core concepts behind the {aws} Cloud Development Kit ({aws} CDK).

// Content start

Learn core concepts behind the {aws} Cloud Development Kit ({aws} CDK).

[#concepts-iac] == {aws} CDK and [.noloc]IaC

The {aws} CDK is an open-source framework that you can use to manage your {aws} infrastructure using code. This approach is known as *infrastructure as code ([.noloc]IaC)*. By managing and provisioning your infrastructure as code, you treat your infrastructure in the same way that developers treat code. This provides many benefits, such as version control and scalability. To learn more about IaC, see https://aws.amazon.com/what-is/iac/[What is Infrastructure as Code?]

[#concepts-cfn] == {aws} CDK and {aws} CloudFormation

The {aws} CDK is tightly integrated with {aws} CloudFormation. {aws} CloudFormation is a fully managed service that you can use to manage and provision your infrastructure on {aws}. With {aws} CloudFormation, you define your infrastructure in templates and deploy them to {aws} CloudFormation. The {aws} CloudFormation service then provisions your infrastructure according to the configuration defined on your templates.

{aws} CloudFormation templates are *declarative*, meaning they declare the desired state or outcome of your infrastructure. Using JSON or YAML, you declare your {aws} infrastructure by defining {aws} *resources* and *properties*. Resources represent the many services on {aws} and properties represent your desired configuration of those services. When you deploy your template to {aws} CloudFormation, your resources and their configured properties are provisioned as described on your template.

With the {aws} CDK, you can manage your infrastructure *imperatively*, using general-purpose programming languages. Instead of just defining a desired state declaratively, you can define the logic or sequence necessary to reach the desired state. For example, you can use if statements or conditional loops that determine how to reach a desired end state for your infrastructure.

Infrastructure created with the {aws} CDK is eventually translated, or *synthesized* into {aws} CloudFormation templates and deployed using the {aws} CloudFormation service. So while the {aws} CDK offers a different approach to creating your infrastructure, you still receive the benefits of {aws} CloudFormation, such as extensive {aws} resource configuration support and robust deployment processes.

To learn more about {aws} CloudFormation, see https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html[What is {aws} CloudFormation?] in the *{aws} CloudFormation User Guide*.

[#concepts-abstractions] == {aws} CDK and abstractions

With {aws} CloudFormation, you must define every detail of how your resources are configured. This provides the benefit of having complete control over your infrastructure. However, this requires you to learn, understand, and create robust templates that contain resource configuration details and relationships between resources, such as permissions and event-driven interactions.

With the {aws} CDK, you can have the same control over your resource configurations. However, the {aws} CDK also offers powerful abstractions, which can speed up and simplify the infrastructure development process. For example, the {aws} CDK includes constructs that provide sensible default configurations and helper methods that generate boilerplate code for you. The {aws} CDK also offers tools, such as the {aws} CDK Command Line Interface ({aws} CDK CLI), that perform infrastructure management actions for you.

[#concepts-learn] == Learn more about core {aws} CDK concepts

[#concepts-learn-interact] *Interacting with the {aws} CDK*:: + When using with the {aws} CDK, you will primarily interact with the {aws} Construct Library and the {aws} CDK CLI.

[#concepts-learn-develop] *Developing with the {aws} CDK*:: + The {aws} CDK can be written in any xref:languages[supported programming language]. You start with a xref:projects[CDK project], which contains a structure of folders and files, including xref:assets[assets]. Within the project, you create a xref:apps[CDK application]. Within the app, you define a xref:stacks[stack], which directly represents a CloudFormation stack. Within the stack, you define your {aws} resources and properties using xref:constructs[constructs].

[#concepts-learn-deploy] *Deploying with the {aws} CDK*:: + You deploy CDK apps into an {aws} xref:environments[environment]. Before deploying, you must perform a one-time xref:bootstrapping[bootstrapping] to prepare your environment.

[#concepts-learn-more] *Learn more*:: + To learn more about {aws} CDK core concepts, see the topics in this section.

include::languages.adoc[leveloffset=+1]

include::libraries.adoc[leveloffset=+1]

include::projects.adoc[leveloffset=+1]

include::apps.adoc[leveloffset=+1]

include::stacks.adoc[leveloffset=+1]

include::stages.adoc[leveloffset=+1]

include::constructs.adoc[leveloffset=+1]

include::environments.adoc[leveloffset=+1]

include::bootstrapping.adoc[leveloffset=+1]

include::resources.adoc[leveloffset=+1]

include::identifiers.adoc[leveloffset=+1]

include::tokens.adoc[leveloffset=+1]

include::parameters.adoc[leveloffset=+1]

include::tagging.adoc[leveloffset=+1]

include::assets.adoc[leveloffset=+1]

include::permissions.adoc[leveloffset=+1]

include::context.adoc[leveloffset=+1]

include::feature-flags.adoc[leveloffset=+1]

include::aspects.adoc[leveloffset=+1]

include::attributes.txt[]

// Attributes

[.topic] [#stack-how-to-create-multiple-stacks] = Example: Create a CDK app with multiple stacks :info\_titleabbrev: Example: CDK app with multiple stacks

// Content start

You can create an {aws} Cloud Development Kit ({aws} CDK) application containing multiple xref:stacks[stacks]. When you deploy the {aws} CDK app, each stack becomes its own {aws} CloudFormation template. You can also synthesize and deploy each stack individually using the {aws} CDK CLI cdk deploy command.

In this example, we cover the following:

* How to extend the Stack class to accept new properties or arguments.
* How to use properties to determine which resources the stack contains and their configuration.
* How to instantiate multiple stacks from this class.

The example in this topic uses a Boolean property, named encryptBucket (Python: encrypt\_bucket). It indicates whether an Amazon S3 bucket should be encrypted. If so, the stack enables encryption using a key managed by {aws} Key Management Service ({aws} KMS). The app creates two instances of this stack, one with encryption and one without.

[#cdk-how-to-create-multiple-stacks-prereqs] == Prerequisites

This example assumes that all xref:getting-started[getting started] steps have been completed.

[#cdk-how-to-create-multiple-stacks-create] == Create a CDK project

First, we create a CDK project using the CDK CLI:

==== [role=“tablist”] TypeScript:: + [source,none,subs=“verbatim,attributes”] — mkdir multistack cd multistack cdk init –language=typescript —

JavaScript:: + [source,none,subs=“verbatim,attributes”] — mkdir multistack cd multistack cdk init –language=javascript —

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

Java:: + [source,none,subs=“verbatim,attributes”] — mkdir multistack cd multistack cdk init –language=java — + You can import the resulting Maven project into your Java IDE.

C#:: + [source,none,subs=“verbatim,attributes”] — mkdir multistack cd multistack cdk init –language=csharp — + You can open the file src/Pipeline.sln in Visual Studio. ====

[#cdk-how-to-create-multiple-stacks-extend-stackprops] == Add an optional parameter

The props argument of the Stack constructor fulfills the interface StackProps. In this example, we want the stack to accept an additional property to tell us whether to encrypt the Amazon S3 bucket. To do this, we create an interface or class that includes the property. This allows the compiler to make sure that the property has a Boolean value and enables auto-completion for it in your IDE.

We open our *stack file* in our IDE or editor and add the new interface, class, or argument. New lines are highlighted in bold:

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

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

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

// The code that defines our stack goes here } } —-

JavaScript:: File: lib/multistack-stack.js + JavaScript doesn’t have an interface feature; we don’t need to add any code. + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-stack’);

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

// The code that defines our stack goes here } }

== module.exports = { MultistackStack }

Python:: File: multistack/multistack\_stack.py + Python does not have an interface feature, so we’ll extend our stack to accept the new property by adding a keyword argument. + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from constructs import Construct

class MultistackStack(cdk.Stack):

…. # The Stack class doesn’t know about our encrypt\_bucket parameter, # so accept it separately and pass along any other keyword arguments. def **init**(self, scope: Construct, id: str, \*, encrypt\_bucket=False, \*\*kwargs) -> None: super().\_\_init\_\_(scope, id, \*\*kwargs)

# The code that defines our stack goes here ----

….

Java:: File: src/main/java/com/myorg/MultistackStack.java + It’s more complicated than we really want to get into to extend a props type in Java. Instead, write the stack’s constructor to accept an optional Boolean parameter. Because props is an optional argument, we’ll write an additional constructor that lets you skip it. It will default to false. + [source,java,subs=“verbatim,attributes”] — package com.myorg;

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

import software.amazon.awscdk.services.s3.Bucket;

public class MultistackStack extends Stack { // additional constructors to allow props and/or encryptBucket to be omitted public MultistackStack(final Construct scope, final String id, boolean encryptBucket) { this(scope, id, null, encryptBucket); }

…. public MultistackStack(final Construct scope, final String id) { this(scope, id, null, false); }

public MultistackStack(final Construct scope, final String id, final StackProps props, final boolean encryptBucket) { super(scope, id, props);

// The code that defines our stack goes here

} } —- ….

C#:: File: src/Multistack/MultistackStack.cs + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using constructs;

namespace Multistack {

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

public class MultistackStack : Stack { public MultistackStack(Construct scope, string id, MultiStackProps props) : base(scope, id, props) { // The code that defines our stack goes here } } } —- ==== ….

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

[#cdk-how-to-create-multiple-stacks-define-stack] == Define the stack class

Next, we define our stack class, using our new property. New code is highlighted in bold:

==== [role=“tablist”] TypeScript:: File: lib/multistack-stack.ts + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from constructs; import \* as s3 from ‘aws-cdk-lib/aws-s3’;

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

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

// Add a Boolean property “encryptBucket” to the stack constructor. // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted. // Encrypted bucket uses KMS-managed keys (SSE-KMS). if (props && props.encryptBucket) { new s3.Bucket(this, “MyGroovyBucket”, { encryption: s3.BucketEncryption.KMS\_MANAGED, removalPolicy: cdk.RemovalPolicy.DESTROY }); } else { new s3.Bucket(this, “MyGroovyBucket”, { removalPolicy: cdk.RemovalPolicy.DESTROY}); } } } —-

JavaScript:: File: lib/multistack-stack.js + [source,javascript,subs=“verbatim,attributes”] — const cdk = require(‘aws-cdk-lib’); const s3 = require(‘aws-cdk-lib/aws-s3’);

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

// Add a Boolean property “encryptBucket” to the stack constructor. // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted. // Encrypted bucket uses KMS-managed keys (SSE-KMS). if ( props && props.encryptBucket) { new s3.Bucket(this, “MyGroovyBucket”, { encryption: s3.BucketEncryption.KMS\_MANAGED, removalPolicy: cdk.RemovalPolicy.DESTROY }); } else { new s3.Bucket(this, “MyGroovyBucket”, { removalPolicy: cdk.RemovalPolicy.DESTROY}); } } }

== module.exports = { MultistackStack }

Python:: File: multistack/multistack\_stack.py + [source,python,subs=“verbatim,attributes”] — import aws\_cdk as cdk from constructs import Construct from aws\_cdk import aws\_s3 as s3

class MultistackStack(cdk.Stack):

…. # The Stack class doesn’t know about our encrypt\_bucket parameter, # so accept it separately and pass along any other keyword arguments. def **init**(self, scope: Construct, id: str, \*, encrypt\_bucket=False, \*\*kwargs) -> None: super().\_\_init\_\_(scope, id, \*\*kwargs)

# Add a Boolean property "encryptBucket" to the stack constructor.  
# If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.  
# Encrypted bucket uses KMS-managed keys (SSE-KMS).  
if encrypt\_bucket:  
 s3.Bucket(self, "MyGroovyBucket",  
 encryption=s3.BucketEncryption.KMS\_MANAGED,  
 removal\_policy=cdk.RemovalPolicy.DESTROY)  
else:  
 s3.Bucket(self, "MyGroovyBucket",  
 removal\_policy=cdk.RemovalPolicy.DESTROY) ----

….

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

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

import software.amazon.awscdk.services.s3.Bucket; import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack { // additional constructors to allow props and/or encryptBucket to be omitted public MultistackStack(final Construct scope, final String id, boolean encryptBucket) { this(scope, id, null, encryptBucket); }

…. public MultistackStack(final Construct scope, final String id) { this(scope, id, null, false); }

// main constructor public MultistackStack(final Construct scope, final String id, final StackProps props, final boolean encryptBucket) { super(scope, id, props);

// Add a Boolean property "encryptBucket" to the stack constructor.  
// If true, creates an encrypted bucket. Otherwise, the bucket is  
// unencrypted. Encrypted bucket uses KMS-managed keys (SSE-KMS).  
if (encryptBucket) {  
 Bucket.Builder.create(this, "MyGroovyBucket")  
 .encryption(BucketEncryption.KMS\_MANAGED)  
 .removalPolicy(RemovalPolicy.DESTROY).build();  
} else {  
 Bucket.Builder.create(this, "MyGroovyBucket")  
 .removalPolicy(RemovalPolicy.DESTROY).build();  
}

} } —- ….

C#:: File: src/Multistack/MultistackStack.cs + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.S3;

namespace Multistack {

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

public class MultistackStack : Stack { public MultistackStack(Construct scope, string id, IMultiStackProps props = null) : base(scope, id, props) { // Add a Boolean property “EncryptBucket” to the stack constructor. // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted. // Encrypted bucket uses KMS-managed keys (SSE-KMS). if (props?.EncryptBucket ?? false) { new Bucket(this, “MyGroovyBucket”, new BucketProps { Encryption = BucketEncryption.KMS\_MANAGED, RemovalPolicy = RemovalPolicy.DESTROY }); } else { new Bucket(this, “MyGroovyBucket”, new BucketProps { RemovalPolicy = RemovalPolicy.DESTROY }); } } } } —- ==== ….

[#stack-how-to-create-multiple-stacks-create-stacks] == Create two stack instances

In our *application file*, we add the code to instantiate two separate stacks. We delete the existing MultistackStack definition and define our two stacks. New code is highlight in bold:

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

const app = new cdk.App();

new MultistackStack(app, “MyWestCdkStack”, { env: {region: “us-west-1”}, encryptBucket: false });

new MultistackStack(app, “MyEastCdkStack”, { env: {region: “us-east-1”}, encryptBucket: true });

== app.synth();

JavaScript:: File: bin/multistack.js + [source,javascript,subs=“verbatim,attributes”] — #!/usr/bin/env node const cdk = require(‘aws-cdk-lib’); const { MultistackStack } = require(‘../lib/multistack-stack’);

const app = new cdk.App();

new MultistackStack(app, “MyWestCdkStack”, { env: {region: “us-west-1”}, encryptBucket: false });

new MultistackStack(app, “MyEastCdkStack”, { env: {region: “us-east-1”}, encryptBucket: true });

== app.synth();

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

import aws\_cdk as cdk

from multistack.multistack\_stack import MultistackStack

app = cdk.App() MultistackStack(app, “MyWestCdkStack”, env=cdk.Environment(region=“us-west-1”), encrypt\_bucket=False)

MultistackStack(app, “MyEastCdkStack”, env=cdk.Environment(region=“us-east-1”), encrypt\_bucket=True)

== app.synth()

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

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

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

…. new MultistackStack(app, “MyWestCdkStack”, StackProps.builder() .env(Environment.builder() .region(“us-west-1”) .build()) .build(), false);

new MultistackStack(app, "MyEastCdkStack", StackProps.builder()  
 .env(Environment.builder()  
 .region("us-east-1")  
 .build())  
 .build(), true);  
  
app.synth();

} } —- ….

C#:: File: src/Multistack/Program.cs + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK;

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

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

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

} } —- ==== ….

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

* One stack with an encrypted Amazon S3 bucket in the us-east-1 {aws} Region.
* One stack with an unencrypted Amazon S3 bucket in the us-west-1 {aws} Region.

[#cdk-how-to-create-multiple-stacks-synth-deploy] == Synthesize and deploy the stack

Next, we can deploy stacks from the app. First, we synthesize an {aws} CloudFormation template for MyEastCdkStack. This is the stack in us-east-1 with the encrypted Amazon S3 bucket.

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

## $ cdk synth MyEastCdkStack

To deploy this stack to our {aws} environment, we can issue one of the following commands. The first command uses our default {aws} profile to obtain the credentials to deploy the stack. The second uses a profile that we specify. For PROFILE\_NAME, we can substitute the name of an {aws} CLI profile that contains appropriate credentials for deploying to the us-east-1 {aws} Region.

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

## $ cdk deploy MyEastCdkStack

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

$ cdk deploy MyEastCdkStack –profile=++++++—-++++++

[#cdk-how-to-create-multiple-stacks-destroy-stack] == Clean up

To avoid charges for resources that we deployed, we destroy the stack using the following command:

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

## cdk destroy MyEastCdkStack

The destroy operation fails if there is anything stored in the stack’s bucket. There shouldn’t be, since we only created the bucket. If we did put something in the bucket, we must delete the bucket contents before destroying the stack. We can use the {aws} Management Console or the {aws} CLI to delete the bucket contents.

include::attributes.txt[]

// Attributes :https–docs-aws-amazon-com-general-latest-gr-rande-html-regional-endpoints: https://docs.aws.amazon.com/general/latest/gr/rande.html#regional-endpoints :https–aws-amazon-com-about-aws-global-infrastructure-regions-az-: https://aws.amazon.com/about-aws/global-infrastructure/regions\_az/

[.topic] :info\_titleabbrev: Environments :keywords: {aws} CDK, {aws} Cloud Development Kit ({aws} CDK), environment, env, {aws} account, {aws} Region

[#environments] = Environments for the {aws} CDK

== [abstract]

## An environment consists of the {aws} account and {aws} Region that you deploy an {aws} Cloud Development Kit ({aws} CDK) stack to.

// Content start

An environment consists of the {aws} account and {aws} Region that you deploy an {aws} Cloud Development Kit ({aws} CDK) stack to.

*{aws} account*:: When you create an {aws} account, you receive an account ID. This ID is a 12-digit number, such as *012345678901*, that uniquely identifies your account. To learn more, see https://docs.aws.amazon.com/accounts/latest/reference/manage-acct-identifiers.html[View {aws} account identifiers] in the *{aws} Account Management Reference Guide*.

*{aws} Region*:: {aws} Regions are named by using a combination of geographical location and a number that represents an Availability Zone in the Region. For example, *[.noloc]us-east-1* represents an Availability Zone in the US East (N. Virginia) Region. To learn more about {aws} Regions, see https://aws.amazon.com/about-aws/global-infrastructure/regions\_az/[Regions and Availability Zones]. For a list of Region codes, see https://docs.aws.amazon.com/general/latest/gr/rande.html#regional-endpoints[Regional endpoints] in the *{aws} General Reference* Reference Guide.

The {aws} CDK can determine environments from your credentials and configuration files. These files can be created and managed with the {aws} Command Line Interface ({aws} CLI). The following is a basic example of these files:

*Credentials file*:: + [source,toml,subs=“verbatim,attributes”] — [default] aws\_access\_key\_id=ASIAIOSFODNN7EXAMPLE aws\_secret\_access\_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY aws\_session\_token = IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE

[user1] aws\_access\_key\_id=ASIAI44QH8DHBEXAMPLE aws\_secret\_access\_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY aws\_session\_token = fcZib3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE —

*Configuration file*:: + [source,toml,subs=“verbatim,attributes”] — [default] region=us-west-2 output=json

[profile user1] region=us-east-1 output=text —

You can pass environment information from these files in your CDK code through environment variables that are provided by the CDK. When you run a CDK CLI command, such as cdk deploy, you then provide the profile from your credentials and configuration files to gather environment information from.

The following is an example of specifying these environment variables in your CDK code:

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

new MyDevStack(app, ‘dev’, { env: { account: process.env.CDK\_DEFAULT\_ACCOUNT, region: process.env.CDK\_DEFAULT\_REGION }}); —

The following is an example of passing values associated with the user1 profile from your credentials and configuration files to the CDK CLI using the --profile option. Values from these files will be passed to your environment variables:

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

$ cdk deploy ++++++–profile ++++++—-++++++++++++

Instead of using values from the credentials and configuration files, you can also hard-code environment values in your CDK code. The following is an example:

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

const envEU = { account: ‘238383838383’, region: ‘eu-west-1’ }; const envUSA = { account: ‘837873873873’, region: ‘us-west-2’ };

new MyFirstStack(app, ‘first-stack-us’, { env: envUSA }); new MyFirstStack(app, ‘first-stack-eu’, { env: envEU }); —

[#environments-learn] == Learn more

To get started with using environments with the {aws} CDK, see xref:configure-env[Configure environments to use with the {aws} CDK].

include::attributes.txt[]

// Attributes :https–github-com-aws-aws-cdk-blob-main-packages-aws-cdk-lib-cx-api-FEATURE-FLAGS-md: https://github.com/aws/aws-cdk/blob/main/packages/aws-cdk-lib/cx-api/FEATURE\_FLAGS.md

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

[#featureflags] = {aws} CDK feature flags

== [abstract]

## The {aws} CDK uses *feature flags* to enable potentially breaking behaviors in a release. Flags are stored as xref:context[Context values and the {aws} CDK] values in cdk.json (or ~/.cdk.json). They are not removed by the cdk context --reset or cdk context --clear commands.

// Content start

The {aws} CDK uses *feature flags* to enable potentially breaking behaviors in a release. Flags are stored as xref:context[Context values and the {aws} CDK] values in cdk.json (or ~/.cdk.json). They are not removed by the cdk context --reset or cdk context --clear commands.

Feature flags are disabled by default. Existing projects that do not specify the flag will continue to work as before with later {aws} CDK releases. New projects created using cdk init include flags enabling all features available in the release that created the project. Edit cdk.json to disable any flags for which you prefer the earlier behavior. You can also add flags to enable new behaviors after upgrading the {aws} CDK.

A list of all current feature flags can be found on the {aws} CDK GitHub repository in {https–github-com-aws-aws-cdk-blob-main-packages-aws-cdk-lib-cx-api-FEATURE-FLAGS-md}[FEATURE\_FLAGS.md]. See the CHANGELOG in a given release for a description of any new feature flags added in that release.

[[featureflags-disabling,featureflags-disabling.title]] == Reverting to v1 behavior

In CDK v2, the defaults for some feature flags have been changed with respect to v1. You can set these back to false to revert to specific {aws} CDK v1 behavior. Use the cdk diff command to inspect the changes to your synthesized template to see if any of these flags are needed.

@aws-cdk/core:newStyleStackSynthesis:: Use the new stack synthesis method, which assumes bootstrap resources with well-known names. Requires xref:bootstrapping[modern bootstrapping], but in turn allows CI/CD via xref:cdk-pipeline[CDK Pipelines] and cross-account deployments out of the box.

@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId:: If your application uses multiple Amazon API Gateway API keys and associates them to usage plans.

@aws-cdk/aws-rds:lowercaseDbIdentifier:: If your application uses Amazon RDS database instance or database clusters, and explicitly specifies the identifier for these.

@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2\_2021:: If your application uses the TLS\_V1\_2\_2019 security policy with Amazon CloudFront distributions. CDK v2 uses security policy TLSv1.2\_2021 by default.

@aws-cdk/core:stackRelativeExports:: If your application uses multiple stacks and you refer to resources from one stack in another, this determines whether absolute or relative path is used to construct {aws} CloudFormation exports.

@aws-cdk/aws-lambda:recognizeVersionProps:: If set to false, the CDK includes metadata when detecting whether a Lambda function has changed. This can cause deployment failures when only the metadata has changed, since duplicate versions are not allowed. There is no need to revert this flag if you’ve made at least one change to all Lambda Functions in your application.

The syntax for reverting these flags in cdk.json is shown here.

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

{ “context”: { “@aws-cdk/core:newStyleStackSynthesis”: false, “@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId”: false, “@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2\_2021”: false, “@aws-cdk/aws-rds:lowercaseDbIdentifier”: false, “@aws-cdk/core:stackRelativeExports”: false, “@aws-cdk/aws-lambda:recognizeVersionProps”: false } } —

:doctype: book

include::attributes.txt[]

// Attributes [.topic] :info\_titleabbrev: Identifiers :keywords: {aws} CDK, Identifiers

[#identifiers] = Identifiers and the {aws} CDK

== [abstract]

## When building {aws} Cloud Development Kit ({aws} CDK) apps, you will use many types of identifiers and names. To use the {aws} CDK effectively and avoid errors, it is important to understand the types of identifiers.

// Content start

When building {aws} Cloud Development Kit ({aws} CDK) apps, you will use many types of identifiers and names. To use the {aws} CDK effectively and avoid errors, it is important to understand the types of identifiers.

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

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

[#identifiers-construct-ids] == Construct IDs

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

= [NOTE]

The id of a stack is also the identifier that you use to refer to it in the xref:cli[{aws} CDK CLI reference].

====

Let’s look at an example where we have two constructs with the identifier MyBucket in our app. The first is defined in the scope of the stack with the identifier Stack1. The second is defined in the scope of a stack with the identifier Stack2. Because they’re defined in different scopes, this doesn’t cause any conflict, and they can coexist in the same app without issues.

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

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

new s3.Bucket(this, ‘MyBucket’); } }

const app = new App(); new MyStack(app, ‘Stack1’); new MyStack(app, ‘Stack2’); —

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

class MyStack extends Stack { constructor(scope, id, props = {}) { super(scope, id, props);

new s3.Bucket(this, ‘MyBucket’); } }

const app = new App(); new MyStack(app, ‘Stack1’); new MyStack(app, ‘Stack2’); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import App, Construct, Stack, StackProps from constructs import Construct from aws\_cdk import aws\_s3 as s3

class MyStack(Stack):

…. def **init**(self, scope: Construct, id: str, \*\*kwargs):

super().\_\_init\_\_(scope, id, \*\*kwargs)  
s3.Bucket(self, "MyBucket")

….

app = App() MyStack(app, ‘Stack1’) MyStack(app, ‘Stack2’) —

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

import software.amazon.awscdk.App; import software.amazon.awscdk.Stack; import software.amazon.awscdk.StackProps; import software.constructs.Construct; import software.amazon.awscdk.services.s3.Bucket;

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

public MyStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props); new Bucket(this, “MyBucket”); } }

// Main.java package com.myorg;

import software.amazon.awscdk.App;

public class Main { public static void main(String[] args) { App app = new App(); new MyStack(app, “Stack1”); new MyStack(app, “Stack2”); } } —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using constructs; using Amazon.CDK.{aws}.S3;

public class MyStack : Stack { public MyStack(Construct scope, string id, IStackProps props) : base(scope, id, props) { new Bucket(this, “MyBucket”); } }

class Program { static void Main(string[] args) { var app = new App(); new MyStack(app, “Stack1”); new MyStack(app, “Stack2”); } } — ====

[#identifiers-paths] == Paths

The constructs in an {aws} CDK application form a hierarchy rooted in the App class. We refer to the collection of IDs from a given construct, its parent construct, its grandparent, and so on to the root of the construct tree, as a *path*.

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

You can access the path of any construct programmatically, as shown in the following example. This gets the path of myConstruct (or my\_construct, as Python developers would write it). Since IDs must be unique within the scope they are created, their paths are always unique within an {aws} CDK application.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const path: string = myConstruct.node.path; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const path = myConstruct.node.path; —

Python:: + [source,python,subs=“verbatim,attributes”] — path = my\_construct.node.path —

Java:: + [source,java,subs=“verbatim,attributes”] — String path = myConstruct.getNode().getPath(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — string path = myConstruct.Node.Path; — ====

[#identifiers-unique-ids] == Unique IDs

{aws} CloudFormation requires that all logical IDs in a template be unique. Because of this, the {aws} CDK must be able to generate a unique identifier for each construct in an application. Resources have paths that are globally unique (the names of all scopes from the stack to a specific resource). Therefore, the {aws} CDK generates the necessary unique identifiers by concatenating the elements of the path and adding an 8-digit hash. (The hash is necessary to distinguish distinct paths, such as A/B/C and A/BC, that would result in the same {aws} CloudFormation identifier. {aws} CloudFormation identifiers are alphanumeric and cannot contain slashes or other separator characters.) The {aws} CDK calls this string the *unique ID* of the construct.

In general, your {aws} CDK app should not need to know about unique IDs. You can, however, access the unique ID of any construct programmatically, as shown in the following example.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const uid: string = Names.uniqueId(myConstruct); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const uid = Names.uniqueId(myConstruct); —

Python:: + [source,python,subs=“verbatim,attributes”] — uid = Names.unique\_id(my\_construct) —

Java:: + [source,java,subs=“verbatim,attributes”] — String uid = Names.uniqueId(myConstruct); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — string uid = Names.Uniqueid(myConstruct); — ====

The *address* is another kind of unique identifier that uniquely distinguishes CDK resources. Derived from the SHA-1 hash of the path, it is not human-readable. However, its constant, relatively short length (always 42 hexadecimal characters) makes it useful in situations where the “traditional” unique ID might be too long. Some constructs may use the address in the synthesized {aws} CloudFormation template instead of the unique ID. Again, your app generally should not need to know about its constructs’ addresses, but you can retrieve a construct’s address as follows.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — const addr: string = myConstruct.node.addr; —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const addr = myConstruct.node.addr; —

Python:: + [source,python,subs=“verbatim,attributes”] — addr = my\_construct.node.addr —

Java:: + [source,java,subs=“verbatim,attributes”] — String addr = myConstruct.getNode().getAddr(); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — string addr = myConstruct.Node.Addr; — ====

[#identifiers-logical-ids] == Logical IDs

Unique IDs serve as the *logical identifiers* (or *logical names*) of resources in the generated {aws} CloudFormation templates for constructs that represent {aws} resources.

For example, the Amazon S3 bucket in the previous example that is created within Stack2 results in an +{aws}::S3::Bucket+ resource. The resource’s logical ID is Stack2MyBucket4DD88B4F in the resulting {aws} CloudFormation template. (For details on how this identifier is generated, see xref:identifiers-unique-ids[Unique IDs].)

[#identifiers-logical-id-stability] === Logical ID stability

Avoid changing the logical ID of a resource after it has been created. {aws} CloudFormation identifies resources by their logical ID. Therefore, if you change the logical ID of a resource, {aws} CloudFormation creates a new resource with the new logical ID, then deletes the existing one. Depending on the type of resource, this might cause service interruption, data loss, or both.

:doctype: book

include::attributes.txt[]

// Attributes

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

[#parameters] = Parameters and the {aws} CDK

== [abstract]

## *Parameters* are custom values that are supplied at deployment time.

// Content start *Parameters* are custom values that are supplied at deployment time. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html[Parameters] are a feature of {aws} CloudFormation. Since the {aws} Cloud Development Kit ({aws} CDK) synthesizes {aws} CloudFormation templates, it also offers support for deployment-time parameters.

[#parameters-about] == About parameters

Using the {aws} CDK, you can define parameters, which can then be used in the properties of constructs you create. You can also deploy stacks that contain parameters.

When deploying the {aws} CloudFormation template using the {aws} CDK CLI, you provide the parameter values on the command line. If you deploy the template through the {aws} CloudFormation console, you are prompted for the parameter values.

In general, we recommend against using {aws} CloudFormation parameters with the {aws} CDK. The usual ways to pass values into {aws} CDK apps are xref:context[context values] and environment variables. Because they are not available at synthesis time, parameter values cannot be easily used for flow control and other purposes in your CDK app.

= [NOTE]

To do control flow with parameters, you can use https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnCondition.html[CfnCondition] constructs, although this is awkward compared to native if statements.

====

Using parameters requires you to be mindful of how the code you’re writing behaves at deployment time, and also at synthesis time. This makes it harder to understand and reason about your {aws} CDK application, in many cases for little benefit.

Generally, it’s better to have your CDK app accept necessary information in a well-defined way and use it directly to declare constructs in your CDK app. An ideal {aws} CDK–generated {aws} CloudFormation template is concrete, with no values remaining to be specified at deployment time.

There are, however, use cases to which {aws} CloudFormation parameters are uniquely suited. If you have separate teams defining and deploying infrastructure, for example, you can use parameters to make the generated templates more widely useful. Also, because the {aws} CDK supports {aws} CloudFormation parameters, you can use the {aws} CDK with {aws} services that use {aws} CloudFormation templates (such as Service Catalog). These {aws} services use parameters to configure the template that’s being deployed.

[#parameters-learn] == Learn more

For instructions on developing CDK apps with parameters, see xref:get-cfn-param[Use CloudFormation parameters to get a CloudFormation value].

: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.

====

# AWS CDK Tokens

Tokens are placeholders for values that aren’t known when defining constructs or synthesizing stacks. This content has been organized into focused sections for better learning.

## 📚 Complete Tokens Guide

The comprehensive tokens documentation has been split into manageable sections:

**👉** [**Access the Complete Tokens Guide**](tokens/README.md)

## Quick Reference

* **What are tokens?** - Placeholders for unknown values at synthesis time
* **When to use tokens?** - For values determined at deployment (ARNs, IDs, etc.)
* **How they work?** - Resolved during CloudFormation deployment

## Learning Path

1. Start with [Tokens Introduction](tokens/01-tokens-introduction.md)
2. Review [Basic Examples](tokens/02-tokens-examples.md)
3. Learn about [Passing Tokens](tokens/03-tokens-passing.md)
4. Understand [How Tokens Work](tokens/04-tokens-how-they-work.md)

For specific token types, jump directly to the relevant section in the [complete guide](tokens/README.md).

## Related Topics

* [Constructs](constructs.md) - How tokens work within constructs
* [Stacks](stacks.md) - Passing tokens between stacks
* [Apps](apps.md) - Token resolution at the app level