: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.