: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].