:doctype: book

include::attributes.txt[]

// Attributes

[.topic] [#work-with-cdk-go] = Working with the {aws} CDK in [.noloc]Go :info\_titleabbrev: In Go

// Content start

[.noloc]Go is a fully-supported client language for the {aws} Cloud Development Kit ({aws} CDK) and is considered stable. Working with the {aws} CDK in Go uses familiar tools. The Go version of the {aws} CDK even uses Go-style identifiers.

Unlike the other languages the CDK supports, [.noloc]Go is not a traditional object-oriented programming language. [.noloc]Go uses composition where other languages often leverage inheritance. We have tried to employ idiomatic [.noloc]Go approaches as much as possible, but there are places where the CDK may differ.

This topic provides guidance when working with the {aws} CDK in [.noloc]Go. See the https://aws.amazon.com/blogs/developer/getting-started-with-the-aws-cloud-development-kit-and-go/[announcement blog post] for a walkthrough of a simple Go project for the {aws} CDK.

[#go-prerequisites] == Get started with [.noloc]Go

To work with the {aws} CDK, you must have an {aws} account and credentials and have installed Node.js and the {aws} CDK Toolkit. See xref:getting-started[Getting started with the {aws} CDK].

The [.noloc]Go bindings for the {aws} CDK use the standard https://golang.org/dl/[Go toolchain], v1.18 or later. You can use the editor of your choice.

= [NOTE]

Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

====

[#go-newproject] == Creating a project

You create a new {aws} CDK project by invoking cdk init in an empty directory. Use the --language option and specify go:

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

mkdir my-project cd my-project cdk init app –language go —

cdk init uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a [.noloc]Go identifier; for example, it should not start with a number or contain spaces.

The resulting project includes a reference to the core {aws} CDK [.noloc]Go module, github.com/aws/aws-cdk-go/awscdk/v2, in go.mod. Issue go get to install this and other required modules.

[#go-managemodules] == Managing {aws} Construct Library modules

In most {aws} CDK documentation and examples, the word “module” is often used to refer to {aws} Construct Library modules, one or more per {aws} service, which differs from idiomatic [.noloc]Go usage of the term. The CDK Construct Library is provided in one [.noloc]Go module with the individual Construct Library modules, which support the various {aws} services, provided as [.noloc]Go packages within that module.

Some services’ {aws} Construct Library support is in more than one Construct Library module ([.noloc]Go package). For example, Amazon Route 53 has three Construct Library modules in addition to the main awsroute53 package, named awsroute53patterns, awsroute53resolver, and awsroute53targets.

The {aws} CDK’s core package, which you’ll need in most {aws} CDK apps, is imported in [.noloc]Go code as github.com/aws/aws-cdk-go/awscdk/v2. Packages for the various services in the {aws} Construct Library live under github.com/aws/aws-cdk-go/awscdk/v2. For example, the Amazon S3 module’s namespace is github.com/aws/aws-cdk-go/awscdk/v2/awss3.

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

import ( “github.com/aws/aws-cdk-go/awscdk/v2/awss3” // … ) —

Once you have imported the Construct Library modules ([.noloc]Go packages) for the services you want to use in your app, you access constructs in that module using, for example, awss3.Bucket.

[#work-with-cdk-go-dependencies] == Managing dependencies in [.noloc]Go

In [.noloc]Go, dependencies versions are defined in go.mod. The default go.mod is similar to the one shown here.

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

module my-package

go 1.16

require ( github.com/aws/aws-cdk-go/awscdk/v2 v2.16.0 github.com/aws/constructs-go/constructs/v10 v10.0.5 github.com/aws/jsii-runtime-go v1.29.0 ) —

Package names (modules, in Go parlance) are specified by URL with the required version number appended. [.noloc]Go’s module system does not support version ranges.

Issue the go get command to install all required modules and update go.mod. To see a list of available updates for your dependencies, issue go list -m -u all.

[#go-cdk-idioms] == {aws} CDK idioms in [.noloc]Go

[#go-naming] === Field and method names

Field and method names use camel casing (likeThis) in TypeScript, the CDK’s language of origin. In [.noloc]Go, these follow [.noloc]Go conventions, so are Pascal-cased (LikeThis).

[#go-cdk-jsii-close] === Cleaning up

In your main method, use defer jsii.Close() to make sure your CDK app cleans up after itself.

[#go-missing-values] === Missing values and pointer conversion

In [.noloc]Go, missing values in {aws} CDK objects such as property bundles are represented by nil. [.noloc]Go doesn’t have nullable types; the only type that can contain nil is a pointer. To allow values to be optional, then, all CDK properties, arguments, and return values are pointers, even for primitive types. This applies to required values as well as optional ones, so if a required value later becomes optional, no breaking change in type is needed.

When passing literal values or expressions, use the following helper functions to create pointers to the values.

* jsii.String
* jsii.Number
* jsii.Bool
* jsii.Time

For consistency, we recommend that you use pointers similarly when defining your own constructs, even though it may seem more convenient to, for example, receive your construct’s id as a string rather than a pointer to a string.

When dealing with optional {aws} CDK values, including primitive values as well as complex types, you should explicitly test pointers to make sure they are not nil before doing anything with them. Go does not have “syntactic sugar” to help handle empty or missing values as some other languages do. However, required values in property bundles and similar structures are guaranteed to exist (construction fails otherwise), so these values need not be nil-checked.

[#go-props] === Constructs and Props

Constructs, which represent one or more {aws} resources and their associated attributes, are represented in [.noloc]Go as interfaces. For example, awss3.Bucket is an interface. Every construct has a factory function, such as awss3.NewBucket, to return a struct that implements the corresponding interface.

All factory functions take three arguments: the scope in which the construct is being defined (its parent in the construct tree), an id, and props, a bundle of key/value pairs that the construct uses to configure the resources it creates. The “bundle of attributes” pattern is also used elsewhere in the {aws} CDK.

In [.noloc]Go, props are represented by a specific struct type for each construct. For example, an awss3.Bucket takes a props argument of type awss3.BucketProps. Use a struct literal to write props arguments.

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

var bucket = awss3.NewBucket(stack, jsii.String(“amzn-s3-demo-bucket”), &awss3.BucketProps{ Versioned: jsii.Bool(true), }) —

[#go-generic-structures] === Generic structures

In some places, the {aws} CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, {aws} CodeBuild’s link:https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_codebuild.BuildSpec.html#static-fromwbrobjectvalue[BuildSpec.fromObject()] method.) In Go, these objects are represented as slices and an empty interface, respectively.

The CDK provides variadic helper functions such as jsii.Strings for building slices containing primitive types.

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

## jsii.Strings(“One”, “Two”, “Three”)

[#go-writing-constructs] === Developing custom constructs

In [.noloc]Go, it is usually more straightforward to write a new construct than to extend an existing one. First, define a new struct type, anonymously embedding one or more existing types if extension-like semantics are desired. Write methods for any new functionality you’re adding and the fields necessary to hold the data they need. Define a props interface if your construct needs one. Finally, write a factory function NewMyConstruct() to return an instance of your construct.

If you are simply changing some default values on an existing construct or adding a simple behavior at instantiation, you don’t need all that plumbing. Instead, write a factory function that calls the factory function of the construct you’re “extending.” In other CDK languages, for example, you might create a TypedBucket construct that enforces the type of objects in an Amazon S3 bucket by overriding the s3.Bucket type and, in your new type’s initializer, adding a bucket policy that allows only specified filename extensions to be added to the bucket. In [.noloc]Go, it is easier to simply write a NewTypedBucket that returns an s3.Bucket (instantiated using s3.NewBucket) to which you have added an appropriate bucket policy. No new construct type is necessary because the functionality is already available in the standard bucket construct; the new “construct” just provides a simpler way to configure it.

[#go-running] == Building, synthesizing, and deploying

The {aws} CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and to run tests. You can do this by issuing go build at a command prompt while in your project’s root directory.

Run any tests you’ve written by running go test at a command prompt.