:doctype: book

include::attributes.txt[]

// Attributes [.topic] [#work-with-cdk-typescript] = Working with the {aws} CDK in TypeScript :info\_titleabbrev: In TypeScript

== [abstract]

## TypeScript is a fully-supported client language for the {aws} CDK and is considered stable.

// Content start

TypeScript is a fully-supported client language for the {aws} Cloud Development Kit ({aws} CDK) and is considered stable. Working with the {aws} CDK in TypeScript uses familiar tools, including Microsoft’s TypeScript compiler (tsc), https://nodejs.org/[Node.js] and the Node Package Manager (npm). You may also use link:https://yarnpkg.com/[Yarn] if you prefer, though the examples in this Guide use NPM. The modules comprising the {aws} Construct Library are distributed via the NPM repository, https://www.npmjs.com/[npmjs.org].

You can use any editor or IDE. Many {aws} CDK developers use https://code.visualstudio.com/[Visual Studio Code] (or its open-source equivalent https://vscodium.com/[VSCodium]), which has excellent support for TypeScript.

[#typescript-prerequisites] == Get started with TypeScript

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].

You also need TypeScript itself (version 3.8 or later). If you don’t already have it, you can install it using npm.

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

## $ npm install -g typescript

= [NOTE]

If you get a permission error, and have administrator access on your system, try sudo npm install -g typescript.

====

Keep TypeScript up to date with a regular npm update -g typescript.

= [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.

====

[#typescript-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 typescript:

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

$ mkdir my-project $ cd my-project $ cdk init app –language typescript —

Creating a project also installs the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib-readme.html[aws-cdk-lib] module and its dependencies.

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 TypeScript identifier; for example, it should not start with a number or contain spaces.

[#typescript-local] == Using local tsc and cdk

For the most part, this guide assumes you install TypeScript and the CDK Toolkit globally (npm install -g typescript aws-cdk), and the provided command examples (such as cdk synth) follow this assumption. This approach makes it easy to keep both components up to date, and since both take a strict approach to backward compatibility, there is generally little risk in always using the latest versions.

Some teams prefer to specify all dependencies within each project, including tools like the TypeScript compiler and the CDK Toolkit. This practice lets you pin these components to specific versions and ensure that all developers on your team (and your CI/CD environment) use exactly those versions. This eliminates a possible source of change, helping to make builds and deployments more consistent and repeatable.

The CDK includes dependencies for both TypeScript and the CDK Toolkit in the TypeScript project template’s package.json, so if you want to use this approach, you don’t need to make any changes to your project. All you need to do is use slightly different commands for building your app and for issuing cdk commands.

[cols=“1h,1,1”] |=== |Operation | Use global tools | Use local tools

|=== | Initialize project | cdk init --language typescript | npx aws-cdk init --language typescript |===

|=== | Build | tsc | npm run build |===

|=== | Run CDK Toolkit command | +cdk ...+ | +npm run cdk ...+ or +npx aws-cdk ...+ |===

npx aws-cdk runs the version of the CDK Toolkit installed locally in the current project, if one exists, falling back to the global installation, if any. If no global installation exists, npx downloads a temporary copy of the CDK Toolkit and runs that. You may specify an arbitrary version of the CDK Toolkit using the @ syntax: npx aws-cdk@2.0 --version prints 2.0.0.

= [TIP]

Set up an alias so you can use the cdk command with a local CDK Toolkit installation.

[role=“tablist”] macOS/Linux:: + [source,bash,subs=“verbatim,attributes”] — $ alias cdk=“npx aws-cdk” —

Windows:: + [source,none,subs=“verbatim,attributes”] — doskey cdk=npx aws-cdk $\* — ====

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

Use the Node Package Manager (npm) to install and update {aws} Construct Library modules for use by your apps, as well as other packages you need. (You may use yarn instead of npm if you prefer.) npm also installs the dependencies for those modules automatically.

Most {aws} CDK constructs are in the main CDK package, named aws-cdk-lib, which is a default dependency in new projects created by cdk init. “Experimental” {aws} Construct Library modules, where higher-level constructs are still under development, are named like @aws-cdk/<SERVICE-NAME>-alpha. The service name has an *aws-* prefix. If you’re unsure of a module’s name, https://www.npmjs.com/search?q=%40aws-cdk[search for it on NPM].

= [NOTE]

The https://docs.aws.amazon.com/cdk/api/v2/docs/aws-construct-library.html[CDK API Reference] also shows the package names.

====

For example, the command below installs the experimental module for {aws} CodeStar.

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

## $ npm install @aws-cdk/aws-codestar-alpha

Some services’ Construct Library support is in more than one namespace. For example, besides aws-route53, there are three additional Amazon Route 53 namespaces, aws-route53-targets, aws-route53-patterns, and aws-route53resolver.

Your project’s dependencies are maintained in package.json. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project’s NPM dependencies to the latest permitted version according to the rules you specified in package.json:

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

## $ npm update

In TypeScript, you import modules into your code under the same name you use to install them using NPM. We recommend the following practices when importing {aws} CDK classes and {aws} Construct Library modules in your applications. Following these guidelines will help make your code consistent with other {aws} CDK applications as well as easier to understand.

* Use ES6-style import directives, not require().
* Generally, import individual classes from aws-cdk-lib.

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

## import { App, Stack } from ‘aws-cdk-lib’;

* If you need many classes from aws-cdk-lib, you may use a namespace alias of cdk instead of importing the individual classes. Avoid doing both.

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

## import \* as cdk from ‘aws-cdk-lib’;

* Generally, import {aws} service constructs using short namespace aliases.

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

## import { aws\_s3 as s3 } from ‘aws-cdk-lib’;

[#work-with-cdk-typescript-dependencies] == Managing dependencies in TypeScript

In TypeScript CDK projects, dependencies are specified in the package.json file in the project’s main directory. The core {aws} CDK modules are in a single NPM package called aws-cdk-lib.

When you install a package using npm install, NPM records the package in package.json for you.

If you prefer, you may use Yarn in place of NPM. However, the CDK does not support Yarn’s plug-and-play mode, which is default mode in Yarn 2. Add the following to your project’s .yarnrc.yml file to turn off this feature.

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

## nodeLinker: node-modules

[#work-with-cdk-typescript-dependencies-apps] === CDK applications

The following is an example package.json file generated by the cdk init --language typescript command:

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

{ “name”: “my-package”, “version”: “0.1.0”, “bin”: { “my-package”: “bin/my-package.js” }, “scripts”: { “build”: “tsc”, “watch”: “tsc -w”, “test”: “jest”, “cdk”: “cdk” }, “devDependencies”: { “@types/jest”: “{caret}26.0.10”, “@types/node”: “10.17.27”, “jest”: “{caret}26.4.2”, “ts-jest”: “{caret}26.2.0”, “aws-cdk”: “2.16.0”, “ts-node”: “{caret}9.0.0”, “typescript”: “~3.9.7” }, “dependencies”: { “aws-cdk-lib”: “2.16.0”, “constructs”: “{caret}10.0.0”, “source-map-support”: “{caret}0.5.16” } } —

For deployable CDK apps, aws-cdk-lib must be specified in the dependencies section of package.json. You can use a caret ({caret}) version number specifier to indicate that you will accept later versions than the one specified as long as they are within the same major version.

For experimental constructs, specify exact versions for the alpha construct library modules, which have APIs that may change. Do not use {caret} or ~ since later versions of these modules may bring API changes that can break your app.

Specify versions of libraries and tools needed to test your app (for example, the jest testing framework) in the devDependencies section of package.json. Optionally, use {caret} to specify that later compatible versions are acceptable.

[#work-with-cdk-typescript-dependencies-libraries] === Third-party construct libraries

If you’re developing a construct library, specify its dependencies using a combination of the peerDependencies and devDependencies sections, as shown in the following example package.json file.

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

{ “name”: “my-package”, “version”: “0.0.1”, “peerDependencies”: { “aws-cdk-lib”: “{caret}2.14.0”, “@aws-cdk/aws-appsync-alpha”: “2.10.0-alpha”, “constructs”: “{caret}10.0.0” }, “devDependencies”: { “aws-cdk-lib”: “2.14.0”, “@aws-cdk/aws-appsync-alpha”: “2.10.0-alpha”, “constructs”: “10.0.0”, “jsii”: “{caret}1.50.0”, “aws-cdk”: “{caret}2.14.0” } } —

In peerDependencies, use a caret ({caret}) to specify the lowest version of aws-cdk-lib that your library works with. This maximizes the compatibility of your library with a range of CDK versions. Specify exact versions for alpha construct library modules, which have APIs that may change. Using peerDependencies makes sure that there is only one copy of all CDK libraries in the node\_modules tree.

In devDependencies, specify the tools and libraries you need for testing, optionally with {caret} to indicate that later compatible versions are acceptable. Specify exactly (without {caret} or ~) the lowest versions of aws-cdk-lib and other CDK packages that you advertise your library be compatible with. This practice makes sure that your tests run against those versions. This way, if you inadvertently use a feature found only in newer versions, your tests can catch it.

= [WARNING]

peerDependencies are installed automatically only by NPM 7 and later. If you are using NPM 6 or earlier, or if you are using Yarn, you must include the dependencies of your dependencies in devDependencies. Otherwise, they won’t be installed, and you will receive a warning about unresolved peer dependencies.

====

[#work-with-cdk-typescript-dependencies-install] === Installing and updating dependencies

Run the following command to install your project’s dependencies.

==== [role=“tablist”] NPM:: + [source,none,subs=“verbatim,attributes”] —

= Install the latest version of everything that matches the ranges in ‘package.json’

npm install

= Install the same exact dependency versions as recorded in ‘package-lock.json’

## npm ci

Yarn:: + [source,bash,subs=“verbatim,attributes”] —

= Install the latest version of everything that matches the ranges in ‘package.json’

$ yarn upgrade

= Install the same exact dependency versions as recorded in ‘yarn.lock’

## $ yarn install –frozen-lockfile

====

To update the installed modules, the preceding npm install and yarn upgrade commands can be used. Either command updates the packages in node\_modules to the latest versions that satisfy the rules in package.json. However, they do not update package.json itself, which you might want to do to set a new minimum version. If you host your package on GitHub, you can configure https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/configuring-dependabot-version-updates[Dependabot version updates] to automatically update package.json. Alternatively, use https://www.npmjs.com/package/npm-check-updates[npm-check-updates].

= [IMPORTANT]

By design, when you install or update dependencies, NPM and Yarn choose the latest version of every package that satisfies the requirements specified in package.json. There is always a risk that these versions may be broken (either accidentally or intentionally). Test thoroughly after updating your project’s dependencies.

====

[#typescript-cdk-idioms] == {aws} CDK idioms in TypeScript

[#typescript-props] === Props

All {aws} Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*. Argument *props* is a bundle of key/value pairs that the construct uses to configure the {aws} resources it creates. Other classes and methods also use the “bundle of attributes” pattern for arguments.

In TypeScript, the shape of props is defined using an interface that tells you the required and optional arguments and their types. Such an interface is defined for each kind of props argument, usually specific to a single construct or method. For example, the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.Bucket.html[Bucket] construct (in the aws-cdk-lib/aws-s3 module) specifies a props argument conforming to the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.BucketProps.html[BucketProps] interface.

If a property is itself an object, for example the https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.BucketProps.html#websiteredirect[websiteRedirect] property of BucketProps, that object will have its own interface to which its shape must conform, in this case https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws\_s3.RedirectTarget.html[RedirectTarget].

If you are subclassing an {aws} Construct Library class (or overriding a method that takes a props-like argument), you can inherit from the existing interface to create a new one that specifies any new props your code requires. When calling the parent class or base method, generally you can pass the entire props argument you received, since any attributes provided in the object but not specified in the interface will be ignored.

A future release of the {aws} CDK could coincidentally add a new property with a name you used for your own property. Passing the value you receive up the inheritance chain can then cause unexpected behavior. It’s safer to pass a shallow copy of the props you received with your property removed or set to undefined. For example:

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

## super(scope, name, {…props, encryptionKeys: undefined});

Alternatively, name your properties so that it is clear that they belong to your construct. This way, it is unlikely they will collide with properties in future {aws} CDK releases. If there are many of them, use a single appropriately-named object to hold them.

[#typescript-missing-values] === Missing values

Missing values in an object (such as props) have the value undefined in TypeScript. Version 3.7 of the language introduced operators that simplify working with these values, making it easier to specify defaults and “short-circuit” chaining when an undefined value is reached. For more information about these features, see the https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html[TypeScript 3.7 Release Notes], specifically the first two features, Optional Chaining and Nullish Coalescing.

[#typescript-running] == Build and run CDK apps

Generally, you should be in the project’s root directory when building and running your application.

Node.js cannot run TypeScript directly; instead, your application is converted to JavaScript using the TypeScript compiler, tsc. The resulting JavaScript code is then executed.

The {aws} CDK automatically does this whenever it needs to run your app. However, it can be useful to compile manually to check for errors and to run tests. To compile your TypeScript app manually, issue npm run build. You may also issue npm run watch to enter watch mode, in which the TypeScript compiler automatically rebuilds your app whenever you save changes to a source file.