:doctype: book

include::attributes.txt[]

// Attributes

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

// Content start

JavaScript is a fully-supported client language for the {aws} CDK and is considered stable. Working with the {aws} Cloud Development Kit ({aws} CDK) in JavaScript uses familiar tools, including https://nodejs.org/[Node.js] and the Node Package Manager (npm). You may also use 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 good support for JavaScript.

[#javascript-prerequisites] == Get started with JavaScript

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].

JavaScript {aws} CDK applications require no additional prerequisites beyond these.

= [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.

====

[#javascript-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 javascript:

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

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

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

[#javascript-local] == Using local cdk

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

Some teams prefer to specify all dependencies within each project, including tools like the CDK Toolkit. This practice lets you pin such 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 a dependency for the CDK Toolkit in the JavaScript 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 javascript | npx aws-cdk init --language javascript |===

|=== | 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@1.120 --version prints 1.120.0.

= [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 $\* — ====

[[javascript-managemodules,javascript-managemodules.title]] == 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-lib/<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,none,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,none,subs=“verbatim,attributes”]

## npm update

In JavaScript, 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 require(), not ES6-style import directives. Older versions of Node.js do not support ES6 imports, so using the older syntax is more widely compatible. (If you really want to use ES6 imports, use https://www.npmjs.com/package/esm[esm] to ensure your project is compatible with all supported versions of Node.js.)
* Generally, import individual classes from aws-cdk-lib.

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

## const { App, Stack } = require(‘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”]

## const cdk = require(‘aws-cdk-lib’);

* Generally, import {aws} Construct Libraries using short namespace aliases.

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

## const { s3 } = require(‘aws-cdk-lib/aws-s3’);

[#work-with-cdk-javascript-dependencies] == Managing dependencies in JavaScript

In JavaScript 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-javascript-dependencies-apps] === CDK applications

The following is an example package.json file generated by the cdk init --language typescript command. The file generated for JavaScript is similar, only without the TypeScript-related entries.

== [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-javascript-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-javascript-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,none,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.

====

[#javascript-cdk-idioms] == {aws} CDK idioms in JavaScript

[#javascript-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*, 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.

Using an IDE or editor that has good JavaScript autocomplete will help avoid misspelling property names. If a construct is expecting an encryptionKeys property, and you spell it encryptionkeys, when instantiating the construct, you haven’t passed the value you intended. This can cause an error at synthesis time if the property is required, or cause the property to be silently ignored if it is optional. In the latter case, you may get a default behavior you intended to override. Take special care here.

When subclassing an {aws} Construct Library class (or overriding a method that takes a props-like argument), you may want to accept additional properties for your own use. These values will be ignored by the parent class or overridden method, because they are never accessed in that code, so you can generally pass on all the props you received.

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.

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

Missing values in an object (such as props) have the value undefined in JavaScript. The usual techniques apply for dealing with these. For example, a common idiom for accessing a property of a value that may be undefined is as follows:

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

// a may be undefined, but if it is not, it may have an attribute b // c is undefined if a is undefined, OR if a doesn’t have an attribute b let c = a && a.b; —

However, if a could have some other “falsy” value besides undefined, it is better to make the test more explicit. Here, we’ll take advantage of the fact that null and undefined are equal to test for them both at once:

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

## let c = a == null ? a : a.b;

= [TIP]

Node.js 14.0 and later support new operators that can simplify the handling of undefined values. For more information, see the https://github.com/tc39/proposal-optional-chaining/blob/master/README.md[optional chaining] and https://github.com/tc39/proposal-nullish-coalescing/blob/master/README.md[nullish coalescing] proposals.

====

[#javascript-using-typescript-examples] == Using TypeScript examples with JavaScript

https://www.typescriptlang.org/[TypeScript] is the language we use to develop the {aws} CDK, and it was the first language supported for developing applications, so many available {aws} CDK code examples are written in TypeScript. These code examples can be a good resource for JavaScript developers; you just need to remove the TypeScript-specific parts of the code.

TypeScript snippets often use the newer ECMAScript import and export keywords to import objects from other modules and to declare the objects to be made available outside the current module. Node.js has just begun supporting these keywords in its latest releases. Depending on the version of Node.js you’re using (or wish to support), you might rewrite imports and exports to use the older syntax.

Imports can be replaced with calls to the require() function.

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

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

Exports can be assigned to the module.exports object.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — export class Stack1 extends cdk.Stack { // … }

export class Stack2 extends cdk.Stack { // … } —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — class Stack1 extends cdk.Stack { // … }

class Stack2 extends cdk.Stack { // … }

== module.exports = { Stack1, Stack2 }

====

= [NOTE]

An alternative to using the old-style imports and exports is to use the https://www.npmjs.com/package/esm[esm] module.

====

Once you’ve got the imports and exports sorted, you can dig into the actual code. You may run into these commonly-used TypeScript features:

* Type annotations
* Interface definitions
* Type conversions/casts
* Access modifiers

Type annotations may be provided for variables, class members, function parameters, and function return types. For variables, parameters, and members, types are specified by following the identifier with a colon and the type. Function return values follow the function signature and consist of a colon and the type.

To convert type-annotated code to JavaScript, remove the colon and the type. Class members must have some value in JavaScript; set them to undefined if they only have a type annotation in TypeScript.

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — var encrypted: boolean = true;

class myStack extends cdk.Stack { bucket: s3.Bucket; // … }

function makeEnv(account: string, region: string) : object { // … } —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — var encrypted = true;

class myStack extends cdk.Stack { bucket = undefined; // … }

function makeEnv(account, region) { // … } —

In TypeScript, interfaces are used to give bundles of required and optional properties, and their types, a name. You can then use the interface name as a type annotation. TypeScript will make sure that the object you use as, for example, an argument to a function has the required properties of the right types.

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

interface myFuncProps { code: lambda.Code, handler?: string } — ====

JavaScript does not have an interface feature, so once you’ve removed the type annotations, delete the interface declarations entirely.

When a function or method returns a general-purpose type (such as object), but you want to treat that value as a more specific child type to access properties or methods that are not part of the more general type’s interface, TypeScript lets you *cast* the value using as followed by a type or interface name. JavaScript doesn’t support (or need) this, so simply remove as and the following identifier. A less-common cast syntax is to use a type name in brackets, <LikeThis>; these casts, too, must be removed.

Finally, TypeScript supports the access modifiers public, protected, and private for members of classes. All class members in JavaScript are public. Simply remove these modifiers wherever you see them.

Knowing how to identify and remove these TypeScript features goes a long way toward adapting short TypeScript snippets to JavaScript. But it may be impractical to convert longer TypeScript examples in this fashion, since they are more likely to use other TypeScript features. For these situations, we recommend https://github.com/alangpierce/sucrase[Sucrase]. Sucrase won’t complain if code uses an undefined variable, for example, as tsc would. If it is syntactically valid, then with few exceptions, Sucrase can translate it to JavaScript. This makes it particularly valuable for converting snippets that may not be runnable on their own.

[#javascript-to-typescript] == Migrating to TypeScript

Many JavaScript developers move to https://www.typescriptlang.org/[TypeScript] as their projects get larger and more complex. TypeScript is a superset of JavaScript–all JavaScript code is valid TypeScript code, so no changes to your code are required–and it is also a supported {aws} CDK language. Type annotations and other TypeScript features are optional and can be added to your {aws} CDK app as you find value in them. TypeScript also gives you early access to new JavaScript features, such as optional chaining and nullish coalescing, before they’re finalized–and without requiring that you upgrade Node.js.

TypeScript’s “shape-based” interfaces, which define bundles of required and optional properties (and their types) within an object, allow common mistakes to be caught while you’re writing the code, and make it easier for your IDE to provide robust autocomplete and other real-time coding advice.

Coding in TypeScript does involve an additional step: compiling your app with the TypeScript compiler, tsc. For typical {aws} CDK apps, compilation requires a few seconds at most.

The easiest way to migrate an existing JavaScript {aws} CDK app to TypeScript is to create a new TypeScript project using cdk init app --language typescript, then copy your source files (and any other necessary files, such as assets like {aws} Lambda function source code) to the new project. Rename your JavaScript files to end in .ts and begin developing in TypeScript.