# Contributing to the AWS Cloud Development Kit

Thanks for your interest in contributing to the AWS CDK! ??

This document describes how to set up a development environment and submit your contributions. Please read it carefully

and let us know if it's not up-to-date (even better, submit a PR with your corrections ;-)).

- [Getting Started](#getting-started)

- [Pull Requests](#pull-requests)

- [Pull Request Checklist](#pull-request-checklist)

- [Step 1: Open Issue](#step-1-open-issue)

- [Step 2: Design (optional)](#step-2-design-optional)

- [Step 3: Work your Magic](#step-3-work-your-magic)

- [Step 4: Commit](#step-4-commit)

- [Step 5: Pull Request](#step-5-pull-request)

- [Step 6: Merge](#step-6-merge)

- [Tools](#tools)

- [Main build scripts](#main-build-scripts)

- [Partial build tools](#partial-build-tools)

- [Useful aliases](#useful-aliases)

- [Linters](#linters)

- [cfn2ts](#cfn2ts)

- [scripts/foreach.sh](#scriptsforeachsh)

- [Jetbrains support (WebStorm/IntelliJ)](#jetbrains-support-webstormintellij)

- [Workflows](#workflows)

- [Full clean build](#full-clean-build)

- [Full Docker build](#full-docker-build)

- [Partial build](#partial-build)

- [Partial pack](#partial-pack)

- [Quick Iteration](#quick-iteration)

- [Linking against this repository](#linking-against-this-repository)

- [Running integration tests in parallel](#running-integration-tests-in-parallel)

- [Visualizing dependencies in a CloudFormation Template](#visualizing-dependencies-in-a-cloudformation-template)

- [Adding Dependencies](#adding-dependencies)

- [Finding dependency cycles between packages](#finding-dependency-cycles-between-packages)

- [Updating all Dependencies](#updating-all-dependencies)

- [Running CLI integration tests](#running-cli-integration-tests)

- [Changing the Cloud Assembly Schema](#changing-cloud-assembly-schema)

- [API Compatibility Checks](#api-compatibility-checks)

- [Examples](#examples)

- [Feature Flags](#feature-flags)

- [Versioning](#versioning)

- [Troubleshooting](#troubleshooting)

- [Debugging](#debugging)

- [Connecting the VS Code Debugger](#connecting-the-vs-code-debugger)

- [Run a CDK unit test in the debugger](#run-a-cdk-unit-test-in-the-debugger)

- [Related Repositories](#related-repositories)

## Getting Started

### Gitpod

For setting up a local development environment,

we recommend using [Gitpod](http://gitpod.io) -

a service that allows you to spin up an in-browser

Visual Studio Code-compatible editor,

with everything set up and ready to go for CDK development.

Just click the button below to create your private workspace:

[![Open in Gitpod](https://gitpod.io/button/open-in-gitpod.svg)](https://gitpod.io/#https://github.com/aws/aws-cdk)

This will start a new Gitpod workspace,

and immediately kick off a build of the CDK code.

Once it's done (it takes around an hour, unfortunately),

you can work on any package that you want to modify,

as described in ['Quick Iteration'](#quick-iteration) below.

Gitpod is free for 50 hours per month -

make sure to stop your workspace when you're done

(you can always resume it later, and it won't need to run the build again).

### Local dependencies

If you don't want to use Gitpod,

you need to have the following SDKs and tools locally:

- [Node.js >= 10.13.0](https://nodejs.org/download/release/latest-v10.x/)

- We recommend using a version in [Active LTS](https://nodejs.org/en/about/releases/)

- ?? versions `13.0.0` to `13.6.0` are not supported due to compatibility issues with our dependencies.

- [Yarn >= 1.19.1, < 1.3](https://yarnpkg.com/lang/en/docs/install)

- [Java >= OpenJDK 8, 11, 14](https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/downloads-list.html)

- [Apache Maven >= 3.6.0, < 4.0](http://maven.apache.org/install.html)

- [.NET Core SDK 3.1.x](https://www.microsoft.com/net/download)

- [Python >= 3.6.5, < 4.0](https://www.python.org/downloads/release/python-365/)

- [Ruby >= 2.5.1, < 3.0](https://www.ruby-lang.org/en/news/2018/03/28/ruby-2-5-1-released/)

- [Docker 19.03](https://docs.docker.com/get-docker/)

The basic commands to get the repository cloned and built locally follow:

```console

$ git clone https://github.com/aws/aws-cdk.git

$ cd aws-cdk

$ yarn build

```

If you get compiler errors when building, a common cause is a globally installed typescript. Try uninstalling it.

```

npm uninstall -g typescript

```

Alternatively, the [Full Docker build](#full-docker-build) workflow can be used so

that you don't have to worry about installing all those tools on your local machine

and instead only depend on having a working Docker install.

## Pull Requests

### Pull Request Checklist

\* [ ] Testing

- Unit test added (prefer not to modify an existing test, otherwise, it's probably a breaking change)

- \_\_CLI change?:\_\_ coordinate update of integration tests with team

- \_\_cdk-init template change?:\_\_ coordinated update of integration tests with team

\* [ ] Docs

- \_\_jsdocs\_\_: All public APIs documented

- \_\_README\_\_: README and/or documentation topic updated

- \_\_Design\_\_: For significant features, design document added to `design` folder

\* [ ] Title and Description

- \_\_Change type\_\_: title prefixed with \*\*fix\*\*, \*\*feat\*\* and module name in parens, which will appear in changelog

- \_\_Title\_\_: use lower-case and doesn't end with a period

- \_\_Breaking?\_\_: last paragraph: "BREAKING CHANGE: <describe what changed + link for details>"

- \_\_Issues\_\_: Indicate issues fixed via: "\*\*Fixes #xxx\*\*" or "\*\*Closes #xxx\*\*"

\* [ ] Sensitive Modules (requires 2 PR approvers)

- IAM Policy Document (in @aws-cdk/aws-iam)

- EC2 Security Groups and ACLs (in @aws-cdk/aws-ec2)

- Grant APIs (only if not based on official documentation with a reference)

---

### Step 1: Open Issue

If there isn't one already, open an issue describing what you intend to contribute. It's useful to communicate in

advance, because sometimes, someone is already working in this space, so maybe it's worth collaborating with them

instead of duplicating the efforts.

### Step 2: Design (optional)

In some cases, it is useful to seek for feedback by iterating on a design document. This is useful

when you plan a big change or feature, or you want advice on what would be the best path forward.

Sometimes, the GitHub issue is sufficient for such discussions, and can be sufficient to get

clarity on what you plan to do. Sometimes, a design document would work better, so people can provide

iterative feedback.

Before starting on a design, read through the [design guidelines](DESIGN\_GUIDELINES.md) for general

patterns and tips.

In such cases, use the GitHub issue description to collect \*\*requirements\*\* and

\*\*use cases\*\* for your feature.

Then, create a design document in markdown format under the `design/` directory

and request feedback through a pull request. Prefix the PR title with "\*\*RFC:\*\*"

(request for comments).

Once the design is finalized, you can re-purpose this PR for the implementation,

or open a new PR to that end.

### Step 3: Work your Magic

Work your magic. Here are some guidelines:

\* Coding style (abbreviated):

\* In general, follow the style of the code around you

\* 2 space indentation

\* 120 characters wide

\* ATX style headings in markdown (e.g. `## H2 heading`)

\* Every change requires a unit test

\* If you change APIs, make sure to update the module's README file

\* Try to maintain a single feature/bugfix per pull request. It's okay to introduce a little bit of housekeeping

changes along the way, but try to avoid conflating multiple features. Eventually all these are going to go into a

single commit, so you can use that to frame your scope.

\* If your change introduces a new construct, take a look at the our

[example Construct Library](packages/@aws-cdk/example-construct-library) for an explanation of the common patterns we use.

Feel free to start your contribution by copy&pasting files from that project,

and then edit and rename them as appropriate -

it might be easier to get started that way.

#### Integration Tests

Integration tests perform a few functions in the CDK code base -

1. Acts as a regression detector. It does this by running `cdk synth` on the integration test and comparing it against

the `\*.expected.json` file. This highlights how a change affects the synthesized stacks.

2. Allows for a way to verify if the stacks are still valid CloudFormation templates, as part of an intrusive change.

This is done by running `yarn integ` which will run `cdk deploy` across all of the integration tests in that package.

Remember to set up AWS credentials before doing this.

3. (Optionally) Acts as a way to validate that constructs set up the CloudFormation resources as expected. A successful

CloudFormation deployment does not mean that the resources are set up correctly.

For Gitpod users only! The best way to supply CDK with your AWS credentials is to add them as

[persisting environment variables](https://www.gitpod.io/docs/environment-variables).

Adding them works as follows via terminal:

```shell

eval $(gp env -e AWS\_ACCESS\_KEY\_ID=XXXXXXXXX)

eval $(gp env -e AWS\_SECRET\_ACCESS\_KEY=YYYYYYY)

eval $(gp env -e AWS\_DEFAULT\_REGION=ZZZZZZZZ)

eval $(gp env -e)

```

If you are working on a new feature that is using previously unused CloudFormation resource types, or involves

configuring resource types across services, you need to write integration tests that use these resource types or

features.

To the extent possible, include a section (like below) in the integration test file that specifies how the successfully

deployed stack can be verified for correctness. Correctness here implies that the resources have been set up correctly.

The steps here are usually AWS CLI commands but they need not be.

```ts

/\*

\* Stack verification steps:

\* \* <step-1>

\* \* <step-2>

\*/

```

Examples:

\* [integ.destinations.ts](https://github.com/aws/aws-cdk/blob/master/packages/%40aws-cdk/aws-lambda-destinations/test/integ.destinations.ts#L7)

\* [integ.token-authorizer.ts](https://github.com/aws/aws-cdk/blob/master/packages/%40aws-cdk/aws-apigateway/test/authorizers/integ.token-authorizer.ts#L6)

### Step 4: Commit

Create a commit with the proposed changes:

\* Commit title and message (and PR title and description) must adhere to [conventionalcommits](https://www.conventionalcommits.org).

\* The title must begin with `feat(module): title`, `fix(module): title`, `refactor(module): title` or

`chore(module): title`.

\* Title should be lowercase.

\* No period at the end of the title.

\* Commit message should describe \_motivation\_. Think about your code reviewers and what information they need in

order to understand what you did. If it's a big commit (hopefully not), try to provide some good entry points so

it will be easier to follow.

\* Commit message should indicate which issues are fixed: `fixes #<issue>` or `closes #<issue>`.

\* Shout out to collaborators.

\* If not obvious (i.e. from unit tests), describe how you verified that your change works.

\* If this commit includes breaking changes, they must be listed at the end in the following format (notice how multiple breaking changes should be formatted):

```

BREAKING CHANGE: Description of what broke and how to achieve this behavior now

\* \*\*module-name:\*\* Another breaking change

\* \*\*module-name:\*\* Yet another breaking change

```

### Step 5: Pull Request

\* Push to a GitHub fork or to a branch (naming convention: `<user>/<feature-bug-name>`)

\* Submit a Pull Request on GitHub. A reviewer will later be assigned by the maintainers.

\* Please follow the PR checklist written below. We trust our contributors to self-check, and this helps that process!

\* Discuss review comments and iterate until you get at least one Approve. When iterating, push new commits to the

same branch. Usually all these are going to be squashed when you merge to master. The commit messages should be hints

for you when you finalize your merge commit message.

\* Make sure to update the PR title/description if things change. The PR title/description are going to be used as the

commit title/message and will appear in the CHANGELOG, so maintain them all the way throughout the process.

### Step 6: Merge

\* Make sure your PR builds successfully (we have CodeBuild setup to automatically build all PRs)

\* Once approved and tested, a maintainer will squash-merge to master and will use your PR title/description as the

commit message.

## Tools

The CDK is a big project, and, at the moment, all of the CDK modules are mastered in a single monolithic repository

(uses [lerna](https://github.com/lerna/lerna)). There are pros and cons to this approach, and it's especially valuable

to maintain integrity in the early stage of the project where things constantly change across the stack. In the future

we believe many of these modules will be extracted to their own repositories.

Another complexity is that the CDK is packaged using [jsii](https://github.com/aws/jsii) to multiple programming

languages. This means that when a full build is complete, there will be a version of each module for each supported

language.

However, in many cases, you can probably get away with just building a portion of the project, based on areas that you

want to work on.

We recommend that you use [Visual Studio Code](https://code.visualstudio.com/) to work on the CDK. Be sure to install

the [eslint extension](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint) for it as well, since we have

strict linting rules that will prevent your code from compiling, but with VSCode and this extension can be automatically

fixed for you by hitting `Ctrl-.` when your cursor is on a red underline.

### Main build scripts

The build process is divided into stages, so you can invoke them as needed from the root of the repo:

- \_\_`yarn build`\_\_: runs the `build` and `test` commands in all modules (in topological order).

- \_\_`yarn pack`\_\_: packages all modules to all supported languages and produces a `dist/` directory with all the outputs

(running this script requires that you installed the [toolchains](#getting-started) for all target languages on your

system).

### Partial build tools

There are also two useful scripts in the `scripts` directory that can help you build part of the repo:

- \_\_`scripts/buildup`\_\_: builds the current module and all of its dependencies (in topological order).

- \_\_`scripts/builddown`\_\_: builds the current module and all of its consumers (in topological order).

### Useful aliases

You can also add a few useful aliases to your shell profile:

```bash

# runs an npm script via lerna for a the current module

alias lr='lerna run --stream --scope $(node -p "require(\"./package.json\").name")'

# runs "yarn build" (build + test) for the current module

alias lb='lr build'

alias lt='lr test'

# runs "yarn watch" for the current module (recommended to run in a separate terminal session):

alias lw='lr watch'

```

### Linters

All linters are executed automatically as part of the build script, `yarn build`.

They can also be executed independently of the build script. From the root of a specific package (e.g.

`packages/@aws-cdk/aws-ec2`), run the following command to execute all the linters on that package -

```bash

yarn lint

```

The following linters are used -

- [eslint](#eslint)

- [pkglint](#pkglint)

- [awslint](#awslint)

#### eslint

All packages in the repo use a standard base configuration found at [eslintrc.js](tools/cdk-build-tools/config/eslintrc.js).

This can be customized for any package by modifying the `.eslintrc` file found at its root.

If you're using the VS Code and would like to see eslint violations on it, install the [eslint

extension](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint).

#### pkglint

The `pkglint` tool "lints" package.json files across the repo according to [rules.ts](tools/pkglint/lib/rules.ts).

To evaluate (and attempt to fix) all package linting issues in the repo, run the following command from the root of the

repository (after bootstrapping):

```console

$ lerna run pkglint

```

You can also do that per package:

```console

$ lr pkglint

```

#### awslint

\*\*awslint\*\* is a linter for the AWS Construct Library APIs. It is executed as a

part of the build of all AWS modules in the project and enforces the [AWS

Construct Library Design Guidelines](./DESIGN\_GUIDELINES.md).

For more information about this tool, see the [awslint

README](./packages/awslint/README.md).

Generally speaking, if you make any changes which violate an awslint rule, build

will fail with appropriate messages. All rules are documented and explained in

the [guidelines](./DESIGN\_GUIDELINES.md).

Here are a few useful commands:

\* `yarn awslint` in every module will run \_\_awslint\_\_ for that module.

\* `yarn awslint list` prints all rules (details and rationale in the guidelines doc)

\* `scripts/foreach.sh yarn awslint` will start linting the entire repo, progressively. Rerun `scripts/foreach.sh` after fixing to continue.

\* `lerna run awslint --no-bail --stream 2> awslint.txt` will run \_\_awslint\_\_ in all modules and collect all results into awslint.txt

\* `lerna run awslint -- -i <RULE>` will run awslint throughout the repo and

evaluate only the rule specified [awslint README](./packages/awslint/README.md)

for details on include/exclude rule patterns.

### cfn2ts

This tool is used to generate our low-level CloudFormation resources

(L1/`CfnFoo`). It is executed as part of the build step of all modules in the

AWS Construct Library.

The tool consults the `cdk-build.cloudformation` key in `package.json` to

determine which CloudFormation namespace this library represents (e.g.

`AWS::EC2` is the namespace for `aws-ec2`). We maintain strict 1:1 relationship

between those.

Each module also has an npm script called `cfn2ts`:

\* `yarn cfn2ts`: generates L1 for a specific module

\* `lerna run cfn2ts`: generates L1 for the entire repo

### scripts/foreach.sh

This wonderful tool allows you to execute a command for all modules in this repo

in topological order, but has the incredible property of being stateful. this

means that if a command fails, you can fix the issue and resume from where you

left off.

To start a session, run:

```console

$ scripts/foreach.sh COMMAND

```

This will execute "COMMAND" for each module in the repo (cwd will be the directory of the module).

if a task fails, it will stop, and then to resume, simply run `foreach.sh` again (with or without the same command).

To reset the session (either when all tasks finished or if you wish to run a different session), run:

```console

$ scripts/foreach.sh --reset

```

If you wish to run a command only against a module's dependency closure, use:

```console

$ cd packages/my-module

$ ../scripts/foreach.sh --up COMMAND

```

This will execute `COMMAND` against `my-module` and all it's deps (in a topological order of course).

### Jetbrains support (WebStorm/IntelliJ)

This project uses lerna and utilizes symlinks inside nested node\_modules directories. You may encounter an issue during

indexing where the IDE attempts to index these directories and keeps following links until the process runs out of

available memory and crashes. To fix this, you can run ```node ./scripts/jetbrains-remove-node-modules.js``` to exclude

these directories.

## Workflows

This section includes step-by-step descriptions of common workflows.

### Full clean build

Clone the repo:

```console

$ git clone https://github.com/aws/aws-cdk.git

$ cd aws-cdk

```

If you already have a local repo and you want a fresh build, run `git clean -fdx` from the root.

Install and build:

```console

$ ./install.sh

$ yarn build

```

If you also wish to package to all languages, make sure you have all the [toolchains](#getting-started) and now run:

```

$ ./pack.sh

```

> NOTE: in local builds, pack.sh will finish but will fail with an error

> indicating the build artifacts use the marker version (`0.0.0`). This is

> normal, and you can trust the output in `dist/` despite the failure. This is a

> protection we have to make sure we don't accidentally release artifacts with

> the marker version.

### Full Docker build

Clone the repo:

```console

$ git clone https://github.com/aws/aws-cdk.git

$ cd aws-cdk

```

If you already have a local repo and you want a fresh build, run `git clean -fdx` from the root.

Build the docker image:

```console

$ docker build -t aws-cdk .

```

This allows you to run the CDK in a CDK-compatible directory with a command like:

```console

$ docker run -v $(pwd):/app -w /app aws-cdk <CDK ARGS>

```

### Partial build

In many cases, you don't really need to build the entire project. Say you want to work on the `@aws-cdk/aws-ec2` module:

```console

$ yarn install

$ cd packages/@aws-cdk/aws-ec2

$ ../../../scripts/buildup

```

Note that `buildup` uses `foreach.sh`, which means it's resumable. If your build fails and you wish to resume, just run

`buildup --resume`. If you wish to restart, run `buildup` again.

### Partial pack

Packing involves generating CDK code in the various target languages, and packaged up ready to be published to the

respective package managers. Once in a while, these will need to be generated either to test the experience of a new

feature, or reproduce a packaging failure.

Before running this, make sure either that the CDK module and all of its dependencies are already built. See [Partial

build](#partial-build) or [Full clean build](#full-clean-build).

To package a specific module, say the `@aws-cdk/aws-ec2` module:

```console

$ cd <root-of-cdk-repo>

$ docker run --rm --net=host -it -v $PWD:$PWD -w $PWD jsii/superchain

docker$ cd packages/@aws-cdk/aws-ec2

docker$ ../../../scripts/foreach.sh --up yarn run package

docker$ exit

```

The `dist/` folder within each module contains the packaged up language artifacts.

### Quick Iteration

After you've built the modules you want to work on once, use `yarn watch` for each module that you are modifying.

Watch the EC2 and IAM modules in a second terminal session:

```console

$ cd packages/@aws-cdk/aws-ec2

$ yarn watch & # runs in the background

$ cd packages/@aws-cdk/aws-iam

$ yarn watch & # runs in the background

```

Code...

Now to test, you can either use `yarn test` or invoke nodeunit/jest directly:

Running nodeunit tests directly on a module

```console

$ cd packages/@aws-cdk/aws-iam

$ nodeunit test/test.\*.js

<BOOM>

```

Running jest tests directly on a module

```console

$ cd packages/@aws-cdk/aws-iam

$ jest test/\*test.js

<BOOM>

```

### Linking against this repository

The script `./link-all.sh` can be used to generate symlinks to all modules in this repository under some `node\_module`

directory. This can be used to develop against this repo as a local dependency.

One can use the `postinstall` script to symlink this repo:

```json

{

"scripts": {

"postinstall": "../aws-cdk/link-all.sh"

}

}

```

This assumes this repo is a sibling of the target repo and will install the CDK as a linked dependency during

\_\_yarn install\_\_.

### Running integration tests in parallel

Integration tests may take a long time to complete. We can speed this up by running them in parallel

in different regions.

```shell

# Install GNU parallel (may require uninstall 'moreutils' if you have it)

$ apt-get install parallel

$ brew install parallel

$ scripts/run-integ-parallel @aws-cdk/aws-ec2 @aws-cdk/aws-autoscaling ...

```

### Visualizing dependencies in a CloudFormation Template

Use GraphViz with `template-deps-to-dot`:

```shell

$ cdk -a some.app.js synth | $awscdk/scripts/template-deps-to-dot | dot -Tpng > deps.png

```

### Adding Dependencies

The root [package.json](./package.json) includes global devDependencies (see

[lerna docs](https://github.com/lerna/lerna#common-devdependencies)) on the topic.

\* To add a global dependency, run `yarn add <dep> --dev` at the root.

\* To add a dependency for a specific module, run `yarn add <dep>` inside the module's directory.

Guidelines:

\* We cannot accept dependencies that use non-permissive open source licenses (Apache, MIT, etc).

\* Make sure dependencies are defined using [caret

ranges](https://docs.npmjs.com/misc/semver#caret-ranges-123-025-004) (e.g. `^1.2.3`). This enables non-breaking

updates to automatically be picked up.

\* Make sure `yarn.lock` is included in your commit.

### Finding dependency cycles between packages

You can use `find-cycles` to print a list of internal dependency cycles:

```shell

$ scripts/find-cycles.sh

Cycle: @aws-cdk/aws-iam => @aws-cdk/assert => aws-cdk => @aws-cdk/aws-s3 => @aws-cdk/aws-kms => @aws-cdk/aws-iam

Cycle: @aws-cdk/assert => aws-cdk => @aws-cdk/aws-s3 => @aws-cdk/aws-kms => @aws-cdk/assert

Cycle: @aws-cdk/aws-iam => @aws-cdk/assert => aws-cdk => @aws-cdk/aws-s3 => @aws-cdk/aws-iam

Cycle: @aws-cdk/assert => aws-cdk => @aws-cdk/aws-s3 => @aws-cdk/assert

Cycle: @aws-cdk/assert => aws-cdk => @aws-cdk/aws-cloudformation => @aws-cdk/assert

Cycle: @aws-cdk/aws-iam => @aws-cdk/assert => aws-cdk => @aws-cdk/util => @aws-cdk/aws-iam

Cycle: @aws-cdk/aws-sns => @aws-cdk/aws-lambda => @aws-cdk/aws-codecommit => @aws-cdk/aws-sns

Cycle: @aws-cdk/aws-sns => @aws-cdk/aws-lambda => @aws-cdk/aws-codecommit => @aws-cdk/aws-codepipeline => @aws-cdk/aws-sns

```

### Updating all Dependencies

To update all dependencies (without bumping major versions):

1. Obtain a fresh clone from "master".

2. Run `yarn install`

3. Run `./scripts/update-dependencies.sh --mode full` (use `--mode semver` to avoid bumping major versions)

4. Submit a Pull Request.

### Running CLI integration tests

The CLI package (`packages/aws-cdk`) has some integration tests that aren't

run as part of the regular build, since they have some particular requirements.

See the [CLI CONTRIBUTING.md file](packages/aws-cdk/CONTRIBUTING.md) for

more information on running those tests.

### Changing Cloud Assembly Schema

If you plan on making changes to the `cloud-assembly-schema` package, make sure you familiarize yourself with

its own [contribution guide](./packages/@aws-cdk/cloud-assembly-schema/CONTRIBUTING.md)

### API Compatibility Checks

All stable APIs in the CDK go through a compatibility check during build using

the [jsii-diff] tool. This tool downloads the latest released version from npm

and verifies that the APIs in the current build have not changed in a breaking

way.

[jsii-diff]: https://www.npmjs.com/package/jsii-diff

Compatibility checks always run as part of a full build (`yarn build`).

You can use `yarn compat` to run compatibility checks for all modules:

```shell

(working directory is repo root)

$ yarn build

$ yarn compat

```

You can also run `compat` from individual package directories:

```shell

$ cd packages/@aws-cdk/aws-sns

$ yarn build

$ yarn compat

```

The only case where it is legitimate to break a public API is if the existing

API is a bug that blocked the usage of a feature. This means that by breaking

this API we will not break anyone, because they weren't able to use it. The file

`allowed-breaking-changes.txt` in the root of the repo is an exclusion file that

can be used in these cases.

### Examples

Examples typed in fenced code blocks (looking like `'''ts`, but then with backticks

instead of regular quotes) will be automatically extrated, compiled and translated

to other languages when the bindings are generated.

To successfully do that, they must be compilable. The easiest way to do that is using

a \*fixture\*, which looks like this:

```

'''ts fixture=with-bucket

bucket.addLifecycleTransition({ ... });

'''

```

While processing the examples, the tool will look for a file called

`rosetta/with-bucket.ts-fixture` in the package directory. This file will be

treated as a regular TypeScript source file, but it must also contain the text

`/// here`, at which point the example will be inserted. The complete file must

compile properly.

Before the `/// here` marker, the fixture should import the necessary packages

and initialize the required variables.

If no fixture is specified, the fixture with the name

`rosetta/default.ts-fixture` will be used if present. `nofixture` can be used to

opt out of that behavior.

In an `@example` block, which is unfenced, the first line of the example can

contain three slashes to achieve the same effect:

```

/\*\*

\* @example

\* /// fixture=with-bucket

\* bucket.addLifecycleTransition({ ... });

\*/

```

When including packages in your examples (even the package you're writing the

examples for), use the full package name (e.g. `import s3 =

require('@aws-cdk/aws-s3);`). The example will be compiled in an environment

where all CDK packages are available using their public names. In this way,

it's also possible to import packages that are not in the dependency set of

the current package.

For a practical example of how making sample code compilable works, see the

`aws-ec2` package.

Examples of all packages are extracted and compiled as part of the packaging

step. If you are working on getting rid of example compilation errors of a

single package, you can run `scripts/compile-samples` on the package by itself.

For now, non-compiling examples will not yet block the build, but at some point

in the future they will.

### Feature Flags

Sometimes we want to introduce new breaking behavior because we believe this is

the correct default behavior for the CDK. The problem of course is that breaking

changes are only allowed in major versions and those are rare.

To address this need, we have a feature flags pattern/mechanism. It allows us to

introduce new breaking behavior which is disabled by default (so existing

projects will not be affected) but enabled automatically for new projects

created through `cdk init`.

The pattern is simple:

1. Define a new const under

[cx-api/lib/features.ts](https://github.com/aws/aws-cdk/blob/master/packages/%40aws-cdk/cx-api/lib/features.ts)

with the name of the context key that \*\*enables\*\* this new feature (for

example, `ENABLE\_STACK\_NAME\_DUPLICATES`). The context key should be in the

form `module.Type:feature` (e.g. `@aws-cdk/core:enableStackNameDuplicates`).

2. Use `node.tryGetContext(cxapi.ENABLE\_XXX)` to check if this feature is enabled

in your code. If it is not defined, revert to the legacy behavior.

3. Add your feature flag to the `FUTURE\_FLAGS` map in

[cx-api/lib/features.ts](https://github.com/aws/aws-cdk/blob/master/packages/%40aws-cdk/cx-api/lib/features.ts).

This map is inserted to generated `cdk.json` files for new projects created

through `cdk init`.

4. In your PR title (which goes into CHANGELOG), add a `(under feature flag)` suffix. e.g:

```

fix(core): impossible to use the same physical stack name for two stacks (under feature flag)

```

5. Under `BREAKING CHANGES` in your commit message describe this new behavior:

```

BREAKING CHANGE: template file names for new projects created through "cdk init"

will use the template artifact ID instead of the physical stack name to enable

multiple stacks to use the same name. This is enabled through the flag

`@aws-cdk/core:enableStackNameDuplicates` in newly generated `cdk.json` files.

```

In the [next major version of the

CDK](https://github.com/aws/aws-cdk/issues/3398) we will either remove the

legacy behavior or flip the logic for all these features and then

reset the `FEATURE\_FLAGS` map for the next cycle.

### Versioning

All `package.json` files in this repo use a stable marker version of `0.0.0`.

This means that when you declare dependencies, you should always use `0.0.0`.

This makes it easier for us to bump a new version (the `bump.sh` script will

just update the central version and create a CHANGELOG entry) and also reduces

the chance of merge conflicts after a new version is released.

Additional scripts that take part in the versioning mechanism:

- `scripts/get-version.js` can be used to obtain the actual version of the repo.

You can use either from JavaScript code by `require('./scripts/get-version')`

or from a shell script `node -p "require('./scripts/get-version')"`.

- `scripts/get-version-marker.js` returns `0.0.0` and used to DRY the version

marker.

- `scripts/align-version.sh` and `scripts/align-version.js` are used to align

all package.json files in the repo to the official version. This script is

invoked in CI builds and should not be used inside a development environment.

## Troubleshooting

Most build issues can be solved by doing a full clean rebuild:

```shell

$ git clean -fqdx .

$ yarn build

```

However, this will be time consuming. In this section we'll describe some common issues you may encounter and some more

targeted commands you can run to resolve your issue.

\* The compiler is throwing errors on files that I renamed/it's running old tests that I meant to remove/code coverage is

low and I didn't change anything.

If you switch to a branch in which `.ts` files got renamed or deleted, the generated `.js` and `.d.ts` files from the

previous compilation run are still around and may in some cases still be picked up by the compiler or test runners.

Run the following to clear out stale build artifacts:

```shell

$ scripts/clean-stale-files.sh

```

\* I added a dependency but it's not being picked up by the build

You need to tell Lerna to update all dependencies:

```shell

$ node\_modules/.bin/lerna bootstrap

```

\* I added a dependency but it's not being picked up by a `watch` background compilation run.

No it's not. After re-bootstrapping you need to restart the watch command.

\* I added a dependency but it's not being picked up by Visual Studio Code (I still get red underlines).

The TypeScript compiler that's running has cached your dependency tree. After re-bootstrapping,

restart the TypeScript compiler.

Hit F1, type `> TypeScript: Restart TS Server`.

\* I'm doing refactorings between packages and compile times are killing me/I need to switch between

differently-verionsed branches a lot and rebuilds because of version errors are taking too long.

Our build steps for each package do a couple of things, such as generating code and generating JSII assemblies. If

you've done a full build at least once to generate all source files, you can do a quicker TypeScript-only rebuild of the

entire source tree by doing the following:

```shell

# Only works after at least one full build to generate source files

$ scripts/build-typescript.sh

# Also works to start a project-wide watch compile

$ scripts/build-typescript.sh -w

```

This does not do code generation and it does not do JSII checks and JSII assembly generation. Instead of doing a

package-by-package ordered build, it compiles all `.ts` files in the repository all at once. This takes about the same

time as it does to compile the biggest package all by itself, and on my machine is the difference between a 15

CPU-minute build and a 20 CPU-second build. If you use this methods of recompiling and you want to run the test, you

have to disable the built-in rebuild functionality of `lerna run test`:

```shell

$ CDK\_TEST\_BUILD=false lr test

```

## Debugging

### Connecting the VS Code Debugger

\*Note:\* This applies to typescript CDK application only.

To debug your CDK application along with the CDK repository,

1. Clone the CDK repository locally and build the repository. See [Workflows](#workflows) section for the different build options.

2. Build the CDK application using the appropriate npm script (typically, `yarn build`) and then run the `link-all.sh` script as so -

```

cd /path/to/cdk/app

/path/to/aws-cdk/link-all.sh

```

3. Open the CDK application (assume it's `hello-cdk` in these steps) and the CDK repository as a [VS code multi-root workspace](https://code.visualstudio.com/docs/editor/multi-root-workspaces).

4. Open the [workspace settings file](https://code.visualstudio.com/docs/editor/multi-root-workspaces#\_settings) and verify that the following two folders must already exist

```json

{

"folders": [

{ "path": "<path-to-cdk-repo>/aws-cdk" },

{ "path": "<path-to-cdk-app>/hello-cdk" }

],

}

```

5. Add the following launch configuration to the settings file -

```json

"launch": {

"configurations": [{

"type": "node",

"request": "launch",

"name": "Debug hello-cdk",

"program": "${workspaceFolder:hello-cdk}/bin/hello-cdk.js",

"cwd": "${workspaceFolder:hello-cdk}",

"console": "internalConsole",

"sourceMaps": true,

"skipFiles": [ "<node\_internals>/\*\*/\*" ],

"outFiles": [

"${workspaceFolder:aws-cdk}/\*\*/\*.js",

"${workspaceFolder:hello-cdk}/\*\*/\*.js",

],

}]

}

```

\*Go [here](https://code.visualstudio.com/docs/editor/debugging#\_launch-configurations) for more about launch configurations.\*

6. The debug view, should now have a launch configuration called 'Debug hello-cdk' and launching that will start the debugger.

7. Any time you modify the CDK app or any of the CDK modules, they need to be re-built and depending on the change the `link-all.sh` script from step#2, may need to be re-run. Only then, would VS code recognize the change and potentially the breakpoint.

### Run a CDK unit test in the debugger

If you want to run the VSCode debugger on unit tests of the CDK project

itself, do the following:

1. Set a breakpoint inside your unit test.

2. In your terminal, depending on the type of test, run either:

```

# (For tests names test.xxx.ts)

$ node --inspect-brk /path/to/aws-cdk/node\_modules/.bin/nodeunit -t 'TESTNAME'

# (For tests names xxxx.test.ts)

$ node --inspect-brk /path/to/aws-cdk/node\_modules/.bin/jest -i -t 'TESTNAME'

```

3. On the `Run` pane of VSCode, select the run configuration \*\*Attach to NodeJS\*\* and click the button.

## Related Repositories

\* [Samples](https://github.com/aws-samples/aws-cdk-examples): includes sample code in multiple languages

\* [Workshop](https://github.com/aws-samples/aws-cdk-intro-workshop): source for https://cdkworkshop.com

\* [Developer Guide](https://github.com/awsdocs/aws-cdk-guide): markdown source for developer guide

\* [jsii](https://github.com/aws/jsii): the technology we use for multi-language support. If you are looking to help us support new languages, start there.