Customizing `cds build`

Table of Contents

Build Configurations

`cds build` runs *build tasks* on your project folders to prepare them for deployment. Build tasks compile *source files* (typically CDS sources) and create required artifacts, for example, EDMX files or SAP HANA design-time artifacts.

Build tasks are derived from the CDS configuration and project context. By default, CDS models are resolved from these sources:

- *db/*, *srv/*, *app/* — default root folders
- *fts/* and its subfolders when using feature toggles
- CDS model folders and files defined by required services, including built-in ones
  - Examples: persistent queue or MTX-related services

- Explicit `src` folder configured in the build task

Feature toggle folders and required built-in service models will also be added if user-defined models have been configured as a `model` option in your build tasks.

↳ *Learn more about `cds.resolve`*

If custom build tasks are configured, those properties have precedence

For example, you want to configure the *src* folder and add the default models. To achieve this, do not define the *model* option in your build task:

package.json

jsonc

```jsonc
{
  "build": {
    "target": "gen",
    "tasks": [
      {
        "for": "nodejs",
        "src": "srv",
        "options": { /* no "model" entry here */ }
      }
    ]
  }
}
```

This way, the model paths will still be dynamically determined, but the *src* folder is taken from the build task configuration. You still benefit from the automatic determination of models – for example when adding a new external services or when CAP is changing any built-in service defaults.

To control which tasks `cds build` executes, you can add them as part of your project configuration in *package.json* or *.cdsrc.json*, as outlined in the following chapter.

Properties

Build Task

The following build tasks represent the default configuration dynamically determined by `cds build` for a minimal Node.js project when executing `cds build --production` or `cds build --profile production` :

package.json

```json
{
  "build": {
    "target": "gen",
    "tasks": [
      { "for": "hana", "src": "db", "options": {"model": ["db","srv"] } },
      { "for": "nodejs", "src": "srv", "options": {"model": ["db","srv"] } }
    ]
  }
}
```

The executed build tasks are logged to the command line

You can use them as a blue print – copy & paste them into your CDS configuration and adapt them to your needs.

The `for` property defines the executed build task type creating its part of the deployment layout. Currently supported types are:

Property	Description
`hana`	SAP HANA Development Infrastructure (HDI) artifacts ↳ *Learn more about **configuring SAP HANA***
`nodejs`	Node.js applications
`java`	Java applications
`mtx-sidecar`	MTX-enabled projects *with* sidecar architecture. ↳ *Learn more about **Multitenant Saas Application Deployment***
`mtx`	MTX-enabled projects *without* sidecar architecture (Node.js only). Required services are served by the Node.js application itself.
`mtx-extension`	MTX extension project (*extension.tgz*), which is required for extension activation using `cds push` . Extension point restrictions defined by the SaaS app provider are validated by default. If any restriction is violated the build aborts and the errors are logged.

Property	Description
	The build task is created by default for projects that have `"cds": { "extends": "\<SaaS app name\>" }` configured in their *package.json*. ↳ *Learn more about **Extending and Customizing SaaS Solutions***

Additional types may be supported by build plugin contributions.

Customization

Build tasks can be customized using the following properties:

Property	Description
`src`	Source folder of module to be built.
`dest`	Optional destination of the module's build destination, relative to the enclosing project. The *src* folder is used by default.
`options`	`model` : *string* or *array of string* The given list of folders or individual *.cds* file names is resolved based on the current working directory or project folder passed to `cds build`. CDS built-in models (prefix *@sap/cds**) are added by default to the user-defined list of models.

Note: Alternatively you can execute build tasks and pass the described arguments to the command line. See also `cds build --help` for further details.

Build Target Folder

If you want to change the default target folder, use the `cds.build.target` ☼ property. It is resolved based on the root folder of your project.

Node.js

Node.js projects use the folder *./gen* below the project root as build target folder by default.
Relevant source files from *db* or *srv* folders are copied into this folder, which makes it self-contained and ready for deployment. The default folder names can be changed with

the `cds.folders.db`, `cds.folders.srv`, `cds.folders.app` configuration. Or you can go for individual build task configuration for full flexibility.

Project files like *.cdsrc.json* or *.npmrc* located in the *root* folder or in the *srv* folder of your project are copied into the application's deployment folder (default *gen/srv*). Files located in the *srv* folder have precedence over the corresponding files located in the project root directory. As a consequence these files are used when deployed to production. Make sure that the folders do not contain one of these files by mistake. Consider using profiles `development` or `production` in order to distinguish environments. CDS configuration that should be kept locally can be defined in a file *.cdsrc-private.json*.

The contents of the *node_modules* folder is *not* copied into the deployment folder. For security reasons the files *default-env.json* and *.env* are also not copied into the deployment folder.

You can verify the CDS configuration settings that become effective in `production` deployments. Executing `cds env --profile production` in the deployment folder *gen/srv* will log the CDS configuration used in production environment.

↳ *Learn more about* `cds env get`

Note: `cds build` provides `options` you can use to switch the copy behavior of specific files on or off on build task level:

package.json

```json
{
  "build": {
    "tasks": [
      { "for": "nodejs", "options": { "contentCdsrcJson": false, "contentNpmr
      { "for": "hana", "options": { "contentNpmrc": false } }
    ]
  }
}
```

npm Workspace Support beta

Use CLI option `--ws-pack` to enable tarball based deployment of npm workspace dependencies. Workspaces are typically used to manage multiple local packages within a singular top-level root package. Such a setup is often referred to as a monorepo .

As an effect, your workspace dependencies can be deployed to SAP BTP without them being published to an npm registry before.

Behind the scenes, `cds build --ws-pack` creates a tarball in folder *gen/srv* for each workspace dependency of your project that has a `*` version identifier. Dependencies in *gen/package.json* will be adapted to point to the correct tarball file URL:

package.json

```jsonc
{
  "dependencies": {
    "some-package": "^1",  // regular package
    "some-workspace": "*"  // workspace dependency, marked as such via "*"
  }
}
```

Packaging of the tarball content is based on the rules of the `npm pack` command:

- Files and folders defined in *.gitignore* will not be added
- If an optional `files` field is defined in the workspace's *package.json*, only those files will be added.

Java

Java projects use the project's root folder *./* as build target folder by default.
This causes `cds build` to create the build output below the individual source folders. For example, *db/src/gen* contains the build output for the *db/* folder. No source files are copied to *db/src/gen* because they're assumed to be deployed from their original location, the *db/* folder itself.

Implement a Build Plugin since @sap/cds-dk 7.5.0

CDS already offers build plugins to create deployment layouts for the most use cases. However, you will find cases where these plugins are not enough and you have to develop your own. This section shows how such a build plugin can be implemented and how it can be used in projects.

Build plugins are run by `cds build` to generate the required deployment artifacts. Build tasks hold the actual project specific configuration. The task's `for` property value has to match the build plugin ID.

The following description uses the postgres build plugin as reference implementation. It combines runtime and design-time integration in a single plugin `@cap-js/postgres` .

Add Build Logic

A build plugin is a Node.js module complying to the CDS plugin architecture. It must be registered to the CDS build system inside a top-level cds-plugin.js file:

cds-plugin.js

```js
const cds = require('@sap/cds')
const { fs, path } = cds.utils;

cds.build?.register?.('postgres', class PostgresBuildPlugin extends cds.build
  static taskDefaults = { src: cds.env.folders.db }
  static hasTask() {
    return cds.requires.db?.kind === 'postgres';
  }
  init() {
    this.task.dest = path.join(this.task.dest, 'pg');
  }
  async build() {
    const model = await this.model();
    if (!model) return;

    await this.write(cds.compile.to.json(model)).to(path.join('db', 'csn.json

    if (fs.existsSync(path.join(this.task.src, 'data'))) {
      await this.copy(data).to(path.join('db', 'data'))
    }
    . . .
  }
})
```

Notes:

- The build plugin id has to be unique. In the previous snippet, the ID is `postgres` .
- `cds.build` will be `undefined` in non-build CLI scenarios or if the `@sap/cds-dk` package isn't installed (globally or locally as a `devDependency` of the project).

CDS offers a base build plugin implementation, which you can extend to implement your own behavior. The following methods are called by the build system in this sequence:

- `static taskDefaults` - defines default settings for build tasks of this type. For database related plugins the default `src` folder value `cds.folders.db` ☼ should be used, while for cds services related plugins `cds.folders.srv` ☼ .
- `static hasTask()` - determines whether the plugin should be called for the running `cds build` command, returns *true* by default. This will create a build task with default settings defined by `taskDefaults` and settings calculated by the framework.
- `init()` - can be used to initialize properties of the plugin, for example, changing the default build output directory defined by the property `dest` .
- `async clean` - deletes existing build output, folder `this.task.dest` is deleted by default.
- `async build` - performs the build.

The CDS build system auto-detects all required build tasks by invoking the static method `hasTask` on each registered build plugin.

The compiled CSN model can be accessed using the asynchronous methods `model()` or `basemodel()` .

- The method `model()` returns a CSN model for the scope defined by the `options.model` setting. If feature toggles are enabled, this model also includes any toggled feature enhancements.
- To get a CSN model without features, use the method `baseModel()` instead. The model can be used as input for further model processing, like `to.edmx` , `to.hdbtable` , `for.odata` , etc.
- Use `cds.reflect` to access advanced query and filter functionality on the CDS model.

Add build task type to cds schema   since @sap/cds-dk 7.6.0

In addition you can also add a new build task type provided by your plugin. This build task type will then be part of code completion suggestions for `package.json` and `.cdsrc.json` files.

↳ *Learn more about schema contributions here.*

Write Build Output

The `cds.build.Plugin` class provides methods for copying or writing contents to the file system:

postgres/lib/build.js

```js
await this.copy(path.join(this.task.src, 'package.json')).to('package.json');
await this.write({
  dependencies: { '@sap/cds': '^9', '@cap-js/postgres': '^2' },
  scripts: { start: 'cds-deploy' }
}).to('package.json');
```

These `copy` and `write` methods ensure that build output is consistently reported in the console log. Paths are relative to the build task's `dest` folder.

Handle Errors

Messages can be issued using the `pushMessage` method:

postgres/lib/build.js

```js
const { Plugin } = cds.build
const { INFO, WARNING } = Plugin

this.pushMessage('Info message', INFO);
this.pushMessage('Warning message', WARNING);
```

These messages are sorted and filtered according to the CLI parameter `log-level`. They will be logged after the CDS build has been finished. A `BuildError` can be thrown in case of severe errors. In case of any CDS compilation errors, the entire build process is aborted and a corresponding message is logged. The `messages` object can be accessed using `this.messages`. When accessing the compiler API it should be passed as an option - otherwise compiler messages won't get reported.

Run the Plugin

In the application's *package.json*, add a dependency to your plugin package to the list of `devDependencies`.

> Only use *`dependencies`* if the plugin also provides *runtime integration*, which is the case for the *`@cap-js/postgres`* plugin.

package.json

```jsonc
"dependencies": {
  "@cap-js/postgres": "^2"
}
```

The CDS build system by default auto-detects all required build tasks. Alternatively, users can run or configure required build tasks in the very same way as for the built-in tasks.

```sh
cds build
cds build --for postgres
```

```json
"tasks": [
  { "for": "nodejs" },
  { "for": "postgres" }
]
```

> See also the command line help for further details using *`cds build --help`* .

Test-Run Built Projects Locally

The artifacts deployed to the various cloud platforms are generated in the *`gen/srv/`* folder. So, to test the application as it runs on the cloud start your application from the *`gen/srv/`* folder:

```sh
cds build       # to create the build results, followed by either:

cd gen/srv && npx cds-serve
# or:
cd gen/srv && npm start
# or:
npx cds-serve -p gen/srv
```

Edit this page

Last updated: 13/08/2025, 12:40

Previous page
Deploy with CI/CD

Next page
Health Checks

Was this page helpful?

👍 👎