Schematics for libraries

Contents ›

When you create an Angular library, you can provide and package it with schematics that integrate it with the Angular CLI. With your schematics, your users can use `ng add` to install an initial version of your library, `ng generate` to create artifacts defined in your library, and `ng update` to adjust their project for a new version of your library that introduces breaking changes.

All three types of schematics can be part of a collection that you package with your library.

Download the library schematics project for a completed example of the steps below.

Creating a schematics collection

To start a collection, you need to create the schematic files. The following steps show you how to add initial support without modifying any project files.

1. In your library's root folder, create a `schematics/` folder.
2. In the `schematics/` folder, create an `ng-add/` folder for your first schematic.
3. At the root level of the `schematics/` folder, create a `collection.json` file.
4. Edit the `collection.json` file to define the initial schema for your collection.

projects/my-lib/schematics/collection.json (Schematics Collection)

```
{
  "$schema": "../../../node_modules/@angular-devkit/schematics/collection-schema.json",
  "schematics": {
    "ng-add": {
      "description": "Add my library to the project.",
      "factory": "./ng-add/index#ngAdd"
    }
  }
}
```

- The `$schema` path is relative to the Angular Devkit collection schema.

- The `schematics` object describes the named schematics that are part of this collection.
- The first entry is for a schematic named `ng-add`. It contains the description, and points to the factory function that is called when your schematic is executed.

1. In your library project's `package.json` file, add a "schematics" entry with the path to your schema file. The Angular CLI uses this entry to find named schematics in your collection when it runs commands.

projects/my-lib/package.json (Schematics Collection Reference)

```
{
  "name": "my-lib",
  "version": "0.0.1",
  "schematics": "./schematics/collection.json",
}
```

The initial schema that you have created tells the CLI where to find the schematic that supports the `ng add` command. Now you are ready to create that schematic.

Providing installation support

A schematic for the `ng add` command can enhance the initial installation process for your users. The following steps will define this type of schematic.

1. Go to the /schematics/ng-add/ folder.
2. Create the main file, `index.ts`.
3. Open `index.ts` and add the source code for your schematic factory function.

projects/my-lib/schematics/ng-add/index.ts (ng-add Rule Factory)

```
import { Rule, SchematicContext, Tree } from '@angular-devkit/schematics';
import { NodePackageInstallTask } from '@angular-devkit/schematics/tasks';

// Just return the tree
export function ngAdd(options: any): Rule {
  return (tree: Tree, context: SchematicContext) => {
    context.addTask(new NodePackageInstallTask());
    return tree;
  };
}
```

The only step needed to provide initial `ng add` support is to trigger an installation task using the `SchematicContext`. The task uses the user's preferred package manager to add the library to the project's `package.json` configuration file, and install it in the project's `node_modules` directory.

In this example, the function receives the current `Tree` and returns it without any modifications. If you need to, you can do additional setup when your package is installed, such as generating files, updating configuration, or any

other initial setup your library requires.

Define dependency type

Use the `save` option of `ng-add` to configure if the library should be added to the `dependencies`, the `devDepedencies`, or not saved at all in the project's `package.json` configuration file.

projects/my-lib/package.json (ng-add Reference)

```
"ng-add": {
  "save": "devDependencies"
}
```

Possible values are:

- `false` - Don't add the package to package.json
- `true` - Add the package to the dependencies
- `"dependencies"` - Add the package to the dependencies
- `"devDependencies"` - Add the package to the devDependencies

Building your schematics

To bundle your schematics together with your library, you must configure the library to build the schematics separately, then add them to the bundle. You must build your schematics *after* you build your library, so they are placed in the correct directory.

- Your library needs a custom Typescript configuration file with instructions on how to compile your schematics into your distributed library.
- To add the schematics to the library bundle, add scripts to the library's `package.json` file.

Assume you have a library project `my-lib` in your Angular workspace. To tell the library how to build the schematics, add a `tsconfig.schematics.json` file next to the generated `tsconfig.lib.json` file that configures the library build.

1. Edit the `tsconfig.schematics.json` file to add the following content.

projects/my-lib/tsconfig.schematics.json (TypeScript Config)

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "lib": [
      "es2018",
      "dom"
    ],
    "declaration": true,
    "module": "commonjs",
    "moduleResolution": "node",
```

```
    "noEmitOnError": true,
    "noFallthroughCasesInSwitch": true,
    "noImplicitAny": true,
    "noImplicitThis": true,
    "noUnusedParameters": true,
    "noUnusedLocals": true,
    "rootDir": "schematics",
    "outDir": "../../dist/my-lib/schematics",
    "skipDefaultLibCheck": true,
    "skipLibCheck": true,
    "sourceMap": true,
    "strictNullChecks": true,
    "target": "es6",
    "types": [
      "jasmine",
      "node"
    ]
  },
  "include": [
    "schematics/**/*"
  ],
  "exclude": [
    "schematics/*/files/**/*"
  ]
}
```

- The `rootDir` specifies that your `schematics/` folder contains the input files to be compiled.
- The `outDir` maps to the library's output folder. By default, this is the `dist/my-lib` folder at the root of your workspace.

1. To make sure your schematics source files get compiled into the library bundle, add the following scripts to the `package.json` file in your library project's root folder (`projects/my-lib`).

projects/my-lib/package.json (Build Scripts)

```
{
  "name": "my-lib",
  "version": "0.0.1",
  "scripts": {
    "build": "../../node_modules/.bin/tsc -p tsconfig.schematics.json",
    "copy:schemas": "cp --parents schematics/*/schema.json ../../dist/my-lib/",
    "copy:files": "cp --parents -p schematics/*/files/** ../../dist/my-lib/",
    "copy:collection": "cp schematics/collection.json ../../dist/my-
lib/schematics/collection.json",
    "postbuild": "npm run copy:schemas && npm run copy:files && npm run
```

```
copy:collection"
  },
  "peerDependencies": {
    "@angular/common": "^7.2.0",
    "@angular/core": "^7.2.0"
  },
  "schematics": "./schematics/collection.json",
  "ng-add": {
    "save": "devDependencies"
  }
}
```

- The `build` script compiles your schematic using the custom `tsconfig.schematics.json` file.
- The `copy:*` statements copy compiled schematic files into the proper locations in the library output folder in order to preserve the file structure.
- The `postbuild` script copies the schematic files after the `build` script completes.

Providing generation support

You can add a named schematic to your collection that lets your users use the `ng generate` command to create an artifact that is defined in your library.

We'll assume that your library defines a service, `my-service`, that requires some setup. You want your users to be able to generate it using the following CLI command.

```
ng generate my-lib:my-service
```

To begin, create a new subfolder, `my-service`, in the `schematics` folder.

Configure the new schematic

When you add a schematic to the collection, you have to point to it in the collection's schema, and provide configuration files to define options that a user can pass to the command.

1. Edit the `schematics/collection.json` file to point to the new schematic subfolder, and include a pointer to a schema file that will specify inputs for the new schematic.

projects/my-lib/schematics/collection.json (Schematics Collection)

```
{
  "$schema": "../../../node_modules/@angular-devkit/schematics/collection-schema.json",
  "schematics": {
    "ng-add": {
      "description": "Add my library to the project.",
      "factory": "./ng-add/index#ngAdd"
    },
```

```
    "my-service": {
      "description": "Generate a service in the project.",
      "factory": "./my-service/index#myService",
      "schema": "./my-service/schema.json"
    }
  }
}
```

1. Go to the `<lib-root>/schematics/my-service/` folder.
2. Create a `schema.json` file and define the available options for the schematic.

projects/my-lib/schematics/my-service/schema.json (Schematic JSON Schema)

```
{
  "$schema": "http://json-schema.org/schema",
  "id": "SchematicsMyService",
  "title": "My Service Schema",
  "type": "object",
  "properties": {
    "name": {
      "description": "The name of the service.",
      "type": "string"
    },
    "path": {
      "type": "string",
      "format": "path",
      "description": "The path to create the service.",
      "visible": false
    },
    "project": {
      "type": "string",
      "description": "The name of the project.",
      "$default": {
        "$source": "projectName"
      }
    }
   },
  "required": [
    "name"
  ]
}
```

- *id*: A unique id for the schema in the collection.
- *title*: A human-readable description of the schema.

- *type*: A descriptor for the type provided by the properties.
- *properties*: An object that defines the available options for the schematic.
  Each option associates key with a type, description, and optional alias. The type defines the shape of the value you expect, and the description is displayed when the user requests usage help for your schematic.
  See the workspace schema for additional customizations for schematic options.

1. Create a `schema.ts` file and define an interface that stores the values of the options defined in the `schema.json` file.

projects/my-lib/schematics/my-service/schema.ts (Schematic Interface)

```
export interface Schema {
  // The name of the service.
  name: string;

  // The path to create the service.
  path?: string;

  // The name of the project.
  project?: string;
}
```

- *name*: The name you want to provide for the created service.
- *path*: Overrides the path provided to the schematic. The default path value is based on the current working directory.
- *project*: Provides a specific project to run the schematic on. In the schematic, you can provide a default if the option is not provided by the user.

Add template files

To add artifacts to a project, your schematic needs its own template files. Schematic templates support special syntax to execute code and variable substitution.

1. Create a `files/` folder inside the `schematics/my-service/` folder.
2. Create a file named `__name@dasherize__.service.ts.template` that defines a template you can use for generating files. This template will generate a service that already has Angular's `HttpClient` injected into its constructor.

projects/my-lib/schematics/my-service/files/__name@dasherize__.service.ts.template (Schematic Template)

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
```

```
  providedIn: 'root'
})
export class <%= classify(name) %>Service {
  constructor(private http: HttpClient) { }
}
```

- The `classify` and `dasherize` methods are utility functions that your schematic will use to transform your source template and filename.
- The `name` is provided as a property from your factory function. It is the same `name` you defined in the schema.

Add the factory function

Now that you have the infrastructure in place, you can define the main function that performs the modifications you need in the user's project.

The Schematics framework provides a file templating system, which supports both path and content templates. The system operates on placeholders defined inside files or paths that loaded in the input `Tree`. It fills these in using values passed into the `Rule`.

For details of these data structures and syntax, see the Schematics README.

1. Create the main file `index.ts` and add the source code for your schematic factory function.
2. First, import the schematics definitions you will need. The Schematics framework offers many utility functions to create and use rules when running a schematic.

projects/my-lib/schematics/my-service/index.ts (Imports)

```
import {
  Rule, Tree, SchematicsException,
  apply, url, applyTemplates, move,
  chain, mergeWith
} from '@angular-devkit/schematics';

import { strings, normalize, experimental } from '@angular-devkit/core';
```

1. Import the defined schema interface that provides the type information for your schematic's options.

projects/my-lib/schematics/my-service/index.ts (Schema Import)

```
import {
  Rule, Tree, SchematicsException,
  apply, url, applyTemplates, move,
  chain, mergeWith
} from '@angular-devkit/schematics';

import { strings, normalize, experimental } from '@angular-devkit/core';
```

```
import { Schema as MyServiceSchema } from './schema';
```

1. To build up the generation schematic, start with an empty rule factory.

projects/my-lib/schematics/my-service/index.ts (Initial Rule)

```
export function myService(options: MyServiceSchema): Rule {
  return (tree: Tree) => {
    return tree;
  };
}
```

This simple rule factory returns the tree without modification. The options are the option values passed through from the `ng generate` command.

Define a generation rule

We now have the framework in place for creating the code that actually modifies the user's application to set it up for the service defined in your library.

The Angular workspace where the user has installed your library contains multiple projects (applications and libraries). The user can specify the project on the command line, or allow it to default. In either case, your code needs to identify the specific project to which this schematic is being applied, so that you can retrieve information from the project configuration.

You can do this using the `Tree` object that is passed in to the factory function. The `Tree` methods give you access to the complete file tree in your workspace, allowing you to read and write files during the execution of the schematic.

Get the project configuration

1. To determine the destination project, use the `Tree.read()` method to read the contents of the workspace configuration file, `angular.json`, at the root of the workspace. Add the following code to your factory function.

projects/my-lib/schematics/my-service/index.ts (Schema Import)

```
import {
  Rule, Tree, SchematicsException,
  apply, url, applyTemplates, move,
  chain, mergeWith
} from '@angular-devkit/schematics';

import { strings, normalize, experimental } from '@angular-devkit/core';

import { Schema as MyServiceSchema } from './schema';

export function myService(options: MyServiceSchema): Rule {
```

```
  return (tree: Tree) => {
    const workspaceConfig = tree.read('/angular.json');
    if (!workspaceConfig) {
      throw new SchematicsException('Could not find Angular workspace configuration');
    }

    // convert workspace to string
    const workspaceContent = workspaceConfig.toString();

    // parse workspace string into JSON object
    const workspace: experimental.workspace.WorkspaceSchema =
JSON.parse(workspaceContent);
  };
}
```

- Be sure to check that the context exists and throw the appropriate error.
- After reading the contents into a string, parse the configuration into a JSON object, typed to the `WorkspaceSchema`.

1. The `WorkspaceSchema` contains all the properties of the workspace configuration, including a `defaultProject` value for determining which project to use if not provided. We will use that value as a fallback, if no project is explicitly specified in the `ng generate` command.

projects/my-lib/schematics/my-service/index.ts (Default Project)

```
if (!options.project) {
  options.project = workspace.defaultProject;
}
```

1. Now that you have the project name, use it to retrieve the project-specific configuration information.

projects/my-lib/schematics/my-service/index.ts (Project)

```
const projectName = options.project as string;

const project = workspace.projects[projectName];

const projectType = project.projectType === 'application' ? 'app' : 'lib';
```

The `workspace projects` object contains all the project-specific configuration information.

1. The `options.path` determines where the schematic template files are moved to once the schematic is applied.
   The `path` option in the schematic's schema is substituted by default with the current working directory. If the `path` is not defined, use the `sourceRoot` from the project configuration along with the `projectType`.

projects/my-lib/schematics/my-service/index.ts (Project Info)

```
if (options.path === undefined) {
  options.path = `${project.sourceRoot}/${projectType}`;
}
```

Define the rule

A `Rule` can use external template files, transform them, and return another `Rule` object with the transformed template. You can use the templating to generate any custom files required for your schematic.

1. Add the following code to your factory function.

projects/my-lib/schematics/my-service/index.ts (Template transform)

```
const templateSource = apply(url('./files'), [
  applyTemplates({
    classify: strings.classify,
    dasherize: strings.dasherize,
    name: options.name
  }),
  move(normalize(options.path as string))
]);
```

- The `apply()` method applies multiple rules to a source and returns the transformed source. It takes 2 arguments, a source and an array of rules.
- The `url()` method reads source files from your filesystem, relative to the schematic.
- The `applyTemplates()` method receives an argument of methods and properties you want make available to the schematic template and the schematic filenames. It returns a `Rule`. This is where you define the `classify()` and `dasherize()` methods, and the `name` property.
- The `classify()` method takes a value and returns the value in title case. For example, if the provided name is `my service`, it is returned as `MyService`
- The `dasherize()` method takes a value and returns the value in dashed and lowercase. For example, if the provided name is MyService, it is returned as `my-service`.
- The `move` method moves the provided source files to their destination when the schematic is applied.

1. Finally, the rule factory must return a rule.

projects/my-lib/schematics/my-service/index.ts (Chain Rule)

```
return chain([
  mergeWith(templateSource)
]);
```

The `chain()` method allows you to combine multiple rules into a single rule, so that you can perform multiple operations in a single schematic. Here you are only merging the template rules with any code executed by the schematic.

See a complete exampled of the schematic rule function.

projects/my-lib/schematics/my-service/index.ts

```
import {
  Rule, Tree, SchematicsException,
  apply, url, applyTemplates, move,
  chain, mergeWith
} from '@angular-devkit/schematics';

import { strings, normalize, experimental } from '@angular-devkit/core';

import { Schema as MyServiceSchema } from './schema';

export function myService(options: MyServiceSchema): Rule {
  return (tree: Tree) => {
    const workspaceConfig = tree.read('/angular.json');
    if (!workspaceConfig) {
      throw new SchematicsException('Could not find Angular workspace configuration');
    }

    // convert workspace to string
    const workspaceContent = workspaceConfig.toString();

    // parse workspace string into JSON object
    const workspace: experimental.workspace.WorkspaceSchema =
JSON.parse(workspaceContent);
    if (!options.project) {
      options.project = workspace.defaultProject;
    }

    const projectName = options.project as string;

    const project = workspace.projects[projectName];

    const projectType = project.projectType === 'application' ? 'app' : 'lib';

    if (options.path === undefined) {
      options.path = `${project.sourceRoot}/${projectType}`;
    }

    const templateSource = apply(url('./files'), [
      applyTemplates({
```

```
        classify: strings.classify,
        dasherize: strings.dasherize,
        name: options.name
      }),
      move(normalize(options.path as string))
    ]);

    return chain([
      mergeWith(templateSource)
    ]);
  };
}
```

For more information about rules and utility methods, see Provided Rules.

Running your library schematic

After you build your library and schematics, you can install the schematics collection to run against your project. The steps below show you how to generate a service using the schematic you created above.

Build your library and schematics

From the root of your workspace, run the `ng build` command for your library.

```
ng build my-lib
```

Then, you change into your library directory to build the schematic

```
cd projects/my-lib
npm run build
```

Link the library

Your library and schematics are packaged and placed in the `dist/my-lib` folder at the root of your workspace. For running the schematic, you need to link the library into your `node_modules` folder. From the root of your workspace, run the `npm link` command with the path to your distributable library.

```
npm link dist/my-lib
```

Run the schematic

Now that your library is installed, you can run the schematic using the `ng generate` command.

```
ng generate my-lib:my-service --name my-data
```

In the console, you will see that the schematic was run and the `my-data.service.ts` file was created in your app folder.

```
CREATE src/app/my-data.service.ts (208 bytes)
```


Version 10.0.10-local+sha.84d1ba792b.