Creating libraries

Contents ›

Getting started

Refactoring parts of an app into a library

Reusable code and schematics

You can create and publish new libraries to extend Angular functionality. If you find that you need to solve the same problem in more than one app (or want to share your solution with other developers), you have a candidate for a library.

A simple example might be a button that sends users to your company website, that would be included in all apps that your company builds.

> For more details on how a library project is structured you can refer the Library Project Files

Getting started

Use the Angular CLI to generate a new library skeleton with the following command:

```
ng new my-workspace --create-application=false
cd my-workspace
ng generate library my-lib
```

> You can use the monorepo model to use the same workspace for multiple projects. See Setting up for a multi-project workspace.

This creates the `projects/my-lib` folder in your workspace, which contains a component and a service inside an NgModule. The workspace configuration file, `angular.json`, is updated with a project of type 'library'.

```
"projects": {
  ...
  "my-lib": {
    "root": "projects/my-lib",
```

```
    "sourceRoot": "projects/my-lib/src",
    "projectType": "library",
    "prefix": "lib",
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-ng-packagr:build",
        ...
```

You can build, test, and lint the project with CLI commands:

```
ng build my-lib
ng test my-lib
ng lint my-lib
```

Notice that the configured builder for the project is different from the default builder for app projects. This builder, among other things, ensures that the library is always built with the AOT compiler, without the need to specify the `--prod` flag.

To make library code reusable you must define a public API for it. This "user layer" defines what is available to consumers of your library. A user of your library should be able to access public functionality (such as NgModules, service providers and general utility functions) through a single import path.

The public API for your library is maintained in the `public-api.ts` file in your library folder. Anything exported from this file is made public when your library is imported into an application. Use an NgModule to expose services and components.

Your library should supply documentation (typically a README file) for installation and maintenance.

Refactoring parts of an app into a library

To make your solution reusable, you need to adjust it so that it does not depend on app-specific code. Here are some things to consider in migrating application functionality to a library.

- Declarations such as components and pipes should be designed as stateless, meaning they don't rely on or alter external variables. If you do rely on state, you need to evaluate every case and decide whether it is application state or state that the library would manage.
- Any observables that the components subscribe to internally should be cleaned up and disposed of during the lifecycle of those components.
- Components should expose their interactions through inputs for providing context, and outputs for communicating events to other components.
- Services should declare their own providers (rather than declaring providers in the NgModule or a component), so that they are *tree-shakable*. This allows the compiler to leave the service out of the bundle if it never gets injected into the application that imports the library. For more about this, see Tree-shakable providers.
- If you register global service providers or share providers across multiple NgModules, use the `forRoot()` and `forChild()` patterns provided by the RouterModule.
- Check all internal dependencies.

- For custom classes or interfaces used in components or service, check whether they depend on additional classes or interfaces that also need to be migrated.
- Similarly, if your library code depends on a service, that service needs to be migrated.
- If your library code or its templates depend on other libraries (such a Angular Material, for instance), you must configure your library with those dependencies.

Reusable code and schematics

A library typically includes *reusable code* that defines components, services, and other Angular artifacts (pipes, directives, and so on) that you simply import into a project. A library is packaged into an npm package for publishing and sharing, and this package can also include schematics that provide instructions for generating or transforming code directly in your project, in the same way that the CLI creates a generic skeleton app with `ng generate component`. A schematic that is combined with a library can, for example, provide the Angular CLI with the information it needs to generate a particular component defined in that library.

What you include in your library is determined by the kind of task you are trying to accomplish. For example, if you want a dropdown with some canned data to show how to add it to your app, your library could define a schematic to create it. For a component like a dropdown that would contain different passed-in values each time, you could provide it as a component in a shared library.

Suppose you want to read a configuration file and then generate a form based on that configuration. If that form will need additional customization by the user, it might work best as a schematic. However, if the forms will always be the same and not need much customization by developers, then you could create a dynamic component that takes the configuration and generates the form. In general, the more complex the customization, the more useful the schematic approach.

Integrating with the CLI

A library can include schematics that allow it to integrate with the Angular CLI.

- Include an installation schematic so that `ng add` can add your library to a project.
- Include generation schematics in your library so that `ng generate` can scaffold your defined artifacts (components, services, tests, and so on) in a project.
- Include an update schematic so that `ng update` can update your library's dependencies and provide migrations for breaking changes in new releases.

To learn more, see Schematics Overview and Schematics for Libraries.

Publishing your library

Use the Angular CLI and the npm package manager to build and publish your library as an npm package.

Before publishing a library to NPM, build it using the `--prod` flag which will use the older compiler and runtime known as View Engine instead of Ivy.

```
ng build my-lib --prod
cd dist/my-lib
npm publish
```

If you've never published a package in npm before, you must create a user account. Read more in Publishing npm Packages.

> For now, it is not recommended to publish Ivy libraries to NPM because Ivy generated code is not backward compatible with View Engine, so apps using View Engine will not be able to consume them. Furthermore, the internal Ivy instructions are not yet stable, which can potentially break consumers using a different Angular version from the one used to build the library.
>
> When a published library is used in an Ivy app, the Angular CLI will automatically convert it to Ivy using a tool known as the Angular compatibility compiler (`ngcc`). Thus, publishing your libraries using the View Engine compiler ensures that they can be transparently consumed by both View Engine and Ivy apps.

Managing assets in a library

Starting with version 9.x of the ng-packagr tool, you can configure the tool to automatically copy assets into your library package as part of the build process. You can use this feature when your library needs to publish optional theming files, Sass mixins, or documentation (like a changelog).

- Learn how to copy assets into your library as part of the build.
- Learn more about how to use the tool to embed assets in CSS.

Linked libraries

While working on a published library, you can use npm link to avoid reinstalling the library on every build.

The library must be rebuilt on every change. When linking a library, make sure that the build step runs in watch mode, and that the library's `package.json` configuration points at the correct entry points. For example, `main` should point at a JavaScript file, not a TypeScript file.

Use TypeScript path mapping for peer dependencies

Angular libraries should list all `@angular/*` dependencies as peer dependencies. This ensures that when modules ask for Angular, they all get the exact same module. If a library lists `@angular/core` in `dependencies` instead of `peerDependencies`, it might get a different Angular module instead, which would cause your application to break.

While developing a library, you must install all peer dependencies through `devDependencies` to ensure that the library compiles properly. A linked library will then have its own set of Angular libraries that it uses for building, located in its `node_modules` folder. However, this can cause problems while building or running your application.

To get around this problem you can use TypeScript path mapping to tell TypeScript that it should load some modules from a specific location. List all the peer dependencies that your library uses in the workspace TypeScript configuration file `./tsconfig.json`, and point them at the local copy in the app's `node_modules` folder.

```
{
  "compilerOptions": {
    // ...
    // paths are relative to `baseUrl` path.
    "paths": {
```

```
      "@angular/*": [
        "./node_modules/@angular/*"
      ]
    }
  }
}
```

This mapping ensures that your library always loads the local copies of the modules it needs.

Using your own library in apps

You don't have to publish your library to the npm package manager in order to use it in your own apps, but you do have to build it first.

To use your own library in an app:

- Build the library. You cannot use a library before it is built.

```
ng build my-lib
```

- In your apps, import from the library by name:

```
import { myExport } from 'my-lib';
```

Building and rebuilding your library

The build step is important if you haven't published your library as an npm package and then installed the package back into your app from npm. For instance, if you clone your git repository and run `npm install`, your editor will show the `my-lib` imports as missing if you haven't yet built your library.

> When you import something from a library in an Angular app, Angular looks for a mapping between the library name and a location on disk. When you install a library package, the mapping is in the `node_modules` folder. When you build your own library, it has to find the mapping in your `tsconfig` paths.
>
> Generating a library with the Angular CLI automatically adds its path to the `tsconfig` file. The Angular CLI uses the `tsconfig` paths to tell the build system where to find the library.

If you find that changes to your library are not reflected in your app, your app is probably using an old build of the library.

You can rebuild your library whenever you make changes to it, but this extra step takes time. *Incremental builds* functionality improves the library-development experience. Every time a file is changed a partial build is performed that emits the amended files.

Incremental builds can be run as a background process in your dev environment. To take advantage of this feature add the `--watch` flag to the build command:

```
ng build my-lib --watch
```

The CLI `build` command uses a different builder and invokes a different build tool for libraries than it does for applications.

- The build system for apps, `@angular-devkit/build-angular`, is based on `webpack`, and is included in all new Angular CLI projects.
- The build system for libraries is based on `ng-packagr`. It is only added to your dependencies when you add a library using `ng generate library my-lib`.

The two build systems support different things, and even where they support the same things, they do those things differently. This means that the TypeScript source can result in different JavaScript code in a built library than it would in a built application.

For this reason, an app that depends on a library should only use TypeScript path mappings that point to the *built library*. TypeScript path mappings should *not* point to the library source `.ts` files.

Version 10.0.10-local+sha.84d1ba792b.