Creating libraries

This page provides a conceptual overview of how to create and publish new libraries to extend Angular functionality.

If you find that you need to solve the same problem in more than one application (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 applications that your company builds.

Getting started

Use the Angular CLI to generate a new library skeleton in a new workspace with the following commands.

```
ng new my-workspace --no-create-application
cd my-workspace
ng generate library my-lib
```

NAMING YOUR LIBRARY

You should be very careful when choosing the name of your library if you want to publish it later in a public package registry such as npm. See Publishing your library.

Skip to main content

hat is prefixed with `ng-`, such as `ng-library`. The `ng-` prefix is a reserved keyword used from the Angular framework and its libraries. The `ngx-` prefix is preferred as a convention used to denote that the library is suitable for use with Angular. It is also an excellent indication to consumers of the registry to differentiate between libraries of different JavaScript frameworks.

The `ng generate` command creates the `projects/my-lib` folder in your workspace, which contains a component and a service inside an NgModule.

> For more details on how a library project is structured, refer to the Library project files section of the Project File Structure guide.
>
> Use the monorepo model to use the same workspace for multiple projects. See Setting up for a multi-project workspace.

When you generate a new library, 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-angular:ng-
packagr",
        ...
```

Skip to main content project with CLI commands:

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

Notice that the configured builder for the project is different from the default builder for application projects. This builder, among other things, ensures that the library is always built with the AOT compiler.

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 application into a library

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

Skip to main content

ch as components and pipes should be designed as ing 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.
- 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 as Angular Material, for instance), you must configure your library with those dependencies.
- Consider how you provide services to client applications.

Skip to main content

hould declare their own providers, rather than providers in the NgModule or a component. Declaring a provider makes that service *tree-shakable*. This practice lets the compiler 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()` design patterns provided by the RouterModule.
- If your library provides optional services that might not be used by all client applications, support proper tree-shaking for that case by using the lightweight token design pattern.

Integrating with the CLI using code-generation schematics

A library typically includes *reusable code* that defines components, services, and other Angular artifacts (pipes, directives) that you import into a project. A library is packaged into an npm package for publishing and sharing. 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 new component with `ng generate component`. A schematic that is packaged with a library can, for example, provide the Angular CLI with the information it needs to generate a component that configures and uses a particular feature, or set of features, defined in that library. One example of this is Angular Material's navigation schematic which configures the CDK's BreakpointObserver and uses it with Material's MatSideNav and MatToolbar components.

Skip to main content e following kinds of schematics:

- 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) 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.

What you include in your library depends on your task. For example, you could define a schematic to create a dropdown that is pre-populated with canned data to show how to add it to an application. If you want a dropdown that would contain different passed-in values each time, your library could define a schematic to create it with a given configuration. Developers could then use `ng generate` to configure an instance for their own application.

Suppose you want to read a configuration file and then generate a form based on that configuration. If that form needs additional customization by the developer who is using your library, it might work best as a schematic. However, if the form 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.

For more information, see Schematics Overview and Schematics for Libraries.

Publishing your library

Skip to main content nd the npm package manager to build and publish package.

Angular CLI uses a tool called ng-packagr to create packages from your compiled code that can be published to npm. See Building libraries with Ivy for information on the distribution formats supported by `ng-packagr` and guidance on how to choose the right format for your library.

You should always build libraries for distribution using the `production` configuration. This ensures that generated output uses the appropriate optimizations and the correct package format for npm.

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

Managing assets in a library

In your Angular library, the distributable can include additional assets like theming files, Sass mixins, or documentation (like a changelog). For more information copy assets into your library as part of the build and embed assets in component styles.

Skip to main content

additional assets like Sass mixins or pre-compiled CSS. You need to add these manually to the conditional "exports" in the `package.json` of the primary entrypoint.

`ng-packagr` will merge handwritten `"exports"` with the auto-generated ones, allowing for library authors to configure additional export subpaths, or custom conditions.

```
"exports": {
  ".": {
    "sass": "./_index.scss",
  },
  "./theming": {
    "sass": "./_theming.scss"
  },
  "./prebuilt-themes/indigo-pink.css": {
    "style": "./prebuilt-themes/indigo-pink.css"
  }
}
```

The above is an extract from the @angular/material distributable.

Peer dependencies

Angular libraries should list any `@angular/*` dependencies the library depends on 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.

Skip to main content

Using your own library in applications

You don't have to publish your library to the npm package manager to use it the same workspace, but you do have to build it first.

To use your own library in an application:

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

  ```
  ng build my-lib
  ```

- In your applications, 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 application from npm. For instance, if you clone your git repository and run `npm install`, your editor shows the `my-lib` imports as missing if you haven't yet built your library.

Skip to main content

> t something from a library in an Angular application, 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.
>
> For more information, see Path mapping overview.

If you find that changes to your library are not reflected in your application, your application 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 development environment. To take advantage of this feature add the `--watch` flag to the build command:

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

Skip to main content

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

- The build system for applications, `@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 application 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.

Publishing libraries

There are two distribution formats to use when publishing a library:

- partial-Ivy **(recommended)**—contains portable code that can be consumed by Ivy applications built with any version of Angular from v12 onwards.
- full-Ivy—contains private Angular Ivy instructions, which are not guaranteed to work across different versions of Angular. This format requires that the library and application are built with the *exact* same version of Angular. This format is useful for environments where all library and application code is built directly from source.

use the partial-Ivy format as it is stable between ular.

Avoid compiling libraries with full-Ivy code if you are publishing to npm because the generated Ivy instructions are not part of Angular's public API, and so might change between patch versions.

Ensuring library version compatibility

The Angular version used to build an application should always be the same or greater than the Angular versions used to build any of its dependent libraries. For example, if you had a library using Angular version 13, the application that depends on that library should use Angular version 13 or later. Angular does not support using an earlier version for the application.

If you intend to publish your library to npm, compile with partial-Ivy code by setting `"compilationMode": "partial"` in `tsconfig.prod.json`. This partial format is stable between different versions of Angular, so is safe to publish to npm. Code with this format is processed during the application build using the same version of the Angular compiler, ensuring that the application and all of its libraries use a single version of Angular.

Avoid compiling libraries with full-Ivy code if you are publishing to npm because the generated Ivy instructions are not part of Angular's public API, and so might change between patch versions.

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

Skip to main content

Consuming partial-Ivy code outside the Angular CLI

An application installs many Angular libraries from npm into its `node_modules` directory. However, the code in these libraries cannot be bundled directly along with the built application as it is not fully compiled. To finish compilation, use the Angular linker.

For applications that don't use the Angular CLI, the linker is available as a Babel plugin. The plugin is to be imported from `@angular/compiler-cli/linker/babel`.

The Angular linker Babel plugin supports build caching, meaning that libraries only need to be processed by the linker a single time, regardless of other npm operations.

Example of integrating the plugin into a custom Webpack build by registering the linker as a Babel plugin using babel-loader.

webpack.config.mjs

```
import linkerPlugin from '@angular/compiler-
cli/linker/babel';

export default {
  // ...
  module: {
    rules: [
      {
        test: /\.m?js$/,
        use: {
          loader: 'babel-loader',
          options: {
            plugins: [linkerPlugin],
            compact: false,
            cacheDirectory: true,
          }
        }
      }
    ]
  }
  // ...
}
```

> The Angular CLI integrates the linker plugin automatically, so if consumers of your library are using the CLI, they can install Ivy-native libraries from npm without any additional configuration.

Last reviewed on Wed Nov 03 2021