The Angular Library Series - Creating a Library with Angular CLI

Angular libraries just got a lot easier to create thanks to the combination of Angular CLI and ng-packagr.

Todd Palmer Follow
May 28, 2018 · 11 min read

Metso Library in Tampere, Finland

UPDATE: This article was originally written for Angular version 6. Since then Angular 7 has been released. I have verified that all the examples in this article are still valid for version 7.

· · ·

Many of the improvements in **Angular 6** were to **Angular CLI**. The one I have really been looking forward to is the integration of Angular CLI with ng-packagr to generate and build Angular libraries. **ng-packagr** is a fantastic tool created by David Herges that transpiles your library to the Angular Package Format.

In this article we will walk through the details of creating an Angular library. Also, I will highlight some rules that will help you get your library started correctly so you won't run into issues later.

For your convenience I have created a GitHub repository at t-palmer/example-ng6-lib with the completed code.

Introduction

When we use `ng new` , Angular CLI creates a new **workspace** for us.

In our Angular **workspace** we are going to have two projects:

- A library project
  This is the library of components and services that we want to provide. This is the code that we could publish to **npm** for example.

- An application project
  This will be a test harness for our library. Sometimes this application is used as documentation and example usage of the library.

There will also be a third project for **End to End** testing that **Angular CLI** generates for us by default which we will ignore in this article.

Now that we have a high level view of what our Angular workspace will look like, let's set some specific goals for this tutorial:

Goals

- Use Angular CLI to create a workspace with the same name as our intended Angular library: **example-ng6-lib**

- We will have a test application for our example-ng6-lib library named:
  example-ng6-lib-app

- In our example-ng6-lib workspace generate an Angular library named:
  example-ng6-lib

- Our Angular library will have the prefix of **enl** to remind us of **E**xample **N**g6 **L**ibrary.

- We will test our **example-ng6-lib** by importing it as a library into our **example-ng6-lib-app** application.

Angular 6

At the time of this writing Angular 6 is still quite new. So, there are a couple of the changes that will affect this tutorial.

The Angular CLI version number has been synchronized with Angular: jumping from version 1.7 to **version 6.0.0**.

The Angular CLI configuration file angular-cli.json has been replaced with **angular.json**.

Angular CLI now generates a **workspace** that directly supports multiple projects.

Creating an Angular Workspace

Our first goal was to create an **Angular workspace** named **example-ng6-lib**.

For Angular 7

Angular 7 added the very useful flag `--createApplication`. If you are using Angular 7, you should follow the approach I describe in my article:
Angular Workspace: No Application for You!
and **not** the approach below for Angular 6 where we rename the Workspace.

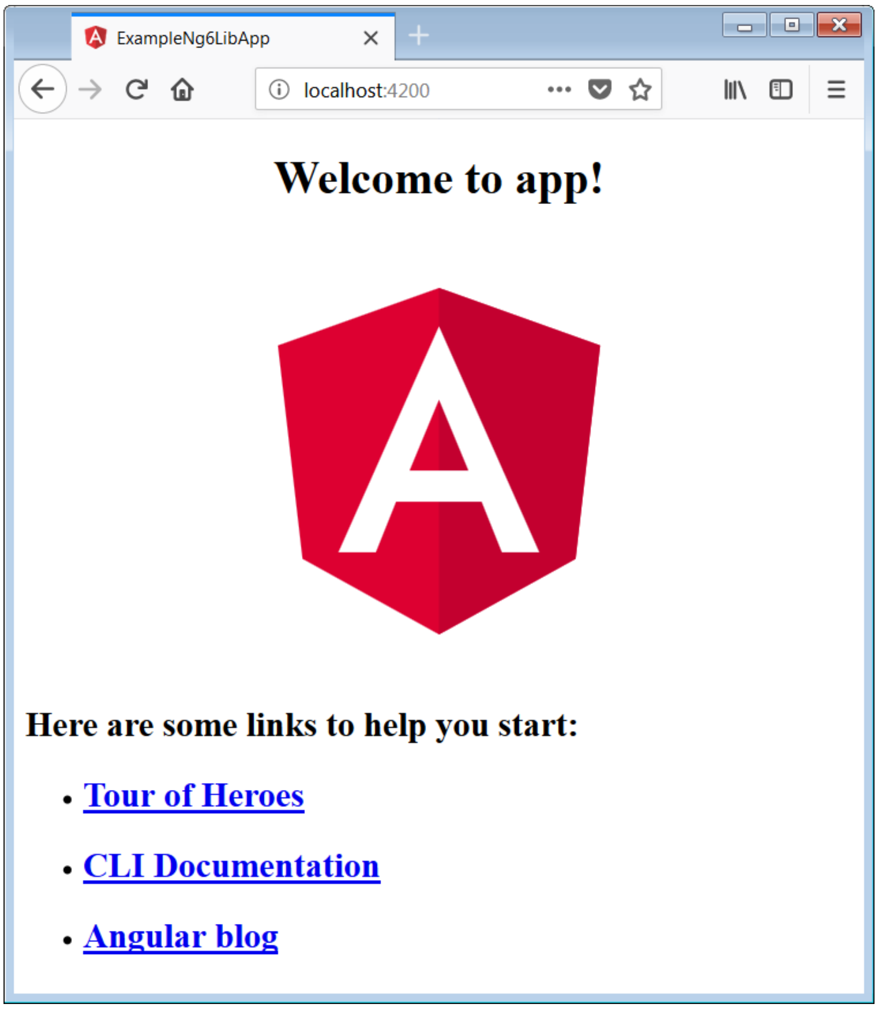
For Angular 6

Because of how the projects work in **Angular 6**, we need to create the Angular Workspace in a bit of a round-about way. We need to create a workspace named **example-ng6-lib-app** and then rename it to **example-ng6-lib**:

```
ng new example-ng6-lib-app
rename example-ng6-lib-app example-ng6-lib
cd example-ng6-lib
ng serve
```

If you need to support IE, see my article: Angular and Internet Explorer.

When we point our browser at:
http://localhost:4200/
we see the starter Angular application that we all know and love.



Welcome to app!

Here are some links to help you start:

- Tour of Heroes
- CLI Documentation
- Angular blog

Angular 6 Configuration: angular.json

Before we move on to creating our library let’s take a quick look at the new Angular configuration file: **angular.json**.

The old angular-cli.json has been replaced by angular.json. Also, the contents have somewhat changed.

The main thing to see is the **projects** object. It has one entry for each project.

```
"projects": {
  "example-ng6-lib-app": {
      ...
    },
    "example-ng6-lib-app-e2e": {
      ...
    }
  },
```

Currently, we have two projects:

- **example-ng6-lib-app**: This is our application that we use as a test harness for our library.
- **example-ng6-lib-app-e2e**: This is the default project for end to end testing. During this article, you can safely ignore this project.

Remember, we told the Angular CLI to create the workspace named as:
example-ng6-lib-app

It then created the default application for us named **example-ng6-lib-app**. This leaves us room to name our library project: **example-ng6-lib**. Once we create our library we will see another project added to the projects object.

> *ALWAYS: Create your workspace using the name of your library-app. Then rename it to the name of your library.*

Generating a Library Module

Now we can generate a new library called **example-ng6-lib** in our workspace.

```
ng generate library example-ng6-lib --prefix=enl
```

Notice we used the `--prefix` flag because we want our library components to be distinct. If we don't, Angular CLI will use `lib` by default.

> *ALWAYS: Use a prefix when generating a library.*

One of the great things about the Angular CLI **generate** command is that it always tells you what files it affects:

```
$ ng generate library example-ng6-lib --prefix=enl
```

```
CREATE projects/example-ng6-lib/karma.conf.js (968 bytes)
CREATE projects/example-ng6-lib/ng-package.json (191 bytes)
CREATE projects/example-ng6-lib/ng-package.prod.json (164
bytes)
CREATE projects/example-ng6-lib/package.json (175 bytes)
CREATE projects/example-ng6-lib/src/test.ts (700 bytes)
CREATE projects/example-ng6-lib/src/public_api.ts (191
bytes)
CREATE projects/example-ng6-lib/tsconfig.lib.json (769
bytes)
CREATE projects/example-ng6-lib/tsconfig.spec.json (246
bytes)
CREATE projects/example-ng6-lib/tslint.json (317 bytes)
CREATE projects/example-ng6-lib/src/lib/example-ng6-
lib.module.ts (261 bytes)
CREATE projects/example-ng6-lib/src/lib/example-ng6-
lib.component.spec.ts (679 bytes)
CREATE projects/example-ng6-lib/src/lib/example-ng6-
lib.component.ts (281 bytes)
CREATE projects/example-ng6-lib/src/lib/example-ng6-
lib.service.spec.ts (418 bytes)
CREATE projects/example-ng6-lib/src/lib/example-ng6-
lib.service.ts (142 bytes)
UPDATE angular.json (4818 bytes)
UPDATE package.json (1724 bytes)
UPDATE tsconfig.json (471 bytes)
```

Here is a quick summary of what the generate library command does:

- Adds a new **example-ng6-lib project** for our library in angular.json
- Adds dependencies for ng-packagr to our package.json
- Adds a reference to the example-ng6-lib build path in tsconfig.json
- Creates sources for our library in projects/example-ng6-lib

Because this is Angular in Depth let's actually take an in depth look at each of these items.

example-ng6-lib project in angular.json

Take a look at **angular.json**. Especially notice that in the `projects` object we have a new project: **example-ng6-lib**.

```
"projects": {
  "example-ng6-lib-app": {

  },
  "example-ng6-lib-app-e2e": {

  },
  "example-ng6-lib": {
    "root": "projects/example-ng6-lib",
    "sourceRoot": "projects/example-ng6-lib/src",
    "projectType": "library",
    "prefix": "enl",
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-ng-packagr:b
        "options": {
          "tsConfig": "projects/example-ng6-lib/tsconf
          "project": "projects/example-ng6-lib/ng-pack
        },
        "configurations": {
          "production": {
            "project": "projects/example-ng6-lib/ng-pa
          }
        }
      },
      "test": {
        "builder": "@angular-devkit/build-angular:karm
        "options": {
          "main": "projects/example-ng6-lib/src/test.t
          "tsConfig": "projects/example-ng6-lib/tsconf
          "karmaConfig": "projects/example-ng6-lib/kar
```

Some of the key elements to notice are:

`root`
This points to our library project's root folder.

`sourceRoot`
This points to root of our library's actual source code.

`projectType`
This specifies this is a `library` as opposed to our other two projects which are of type: `application` .

`prefix`
This is the prefix identifier that we will use in the selectors of our components. Remember, we specified **enl** when we generated the library. You are probably familiar with the **app** prefix that tells us which components belong to our main application.

`architect`
This object has sections that specify how Angular CLI handles `build` , `test` , and `lint` for our project. Notice that in the build section the builder makes use of **ng-packagr**.

ng-packagr dependency in package.json

When generating our library Angular CLI realizes that it needs **ng-packagr**. So, it adds it to our **devDependencies** in our workspace package.json:

```
"ng-packagr": "^3.0.0-rc.2",
```

build path in tsconfig.json

When testing **example-ng6-lib** we want to be able to import it like a library and not just another set of files that are part of our application. Typically, when we use a 3rd party library we use `npm install` and the library gets deployed to our **node-modules** folder.

Although, example-ng6-lib won't be in **node-modules**, it will be built to a sub-folder in our workspace's **dist** folder. Angular CLI adds this folder to our **tsconfig.json** which makes it available for import as a library.

Here is the path that it adds:

```
"paths": {
  "example-ng6-lib": [
    "dist/example-ng6-lib"
  ]
}
```

example-ng6-lib sources

The **src** folder for our library is contained in `projects/example-ng6-lib`. In our library Angular CLI created a new module with a service and a component. Also, looking there we see a few more files:

`package.json`
This is the package.json file specifically for our library. This is the one that gets published with our library as an **npm** package. When people install our library using npm this specifies its dependencies.

`public_api.ts`
This is known as the **entry file**. It specifies what parts of our library are visible externally. Now you may be asking "But Todd, isn't that what `export` does in our modules?" Well yes, but it's a little more complicated than that. We will look at this in more detail later.
Note: as of Angular CLI 7.3 this file changed to `public-api.ts`.

`ng-package.json`
This is the configuration file for **ng-packagr**. In the "old days" we needed to be familiar with its contents. Now, thanks to the new Angular CLI, it's enough to know that it tells ng-packagr where to find our entry file and where to build our library.

Building the Library

Before we can use our newly generated library we need to build it:

```
ng build example-ng6-lib
```

This builds our library to the folder:
`example-ng6-lib-app\dist\example-ng6-lib`

Beginning with version 6.1, Angular always does a production build of our library. If you are still using version 6.0.x you will want to use the `--prod` flag when building your library.

Using the Library in Our Application

One of the central ideas of building a library is that we typically have an application we build along with our library in order to test it.

In our case we have our **example-ng6-lib-app** that will use our library.

Let's try a simple test using our library in our example-ng6-lib-app. To do this we will import our example-ng6-lib's module. Then we'll display the default component that Angular CLI created for us in the library.

Importing the example-ng6-lib Module

Let's modify our **AppModule** in: **src\app\app.module.ts**

Add the **ExampleNg6LibModule** to the `imports` array. Your IDE might think it is helping you out by trying to import the file directly. Don't trust it. You want to import the module in the application using the library by name like this:

```
import { ExampleNg6LibModule } from 'example-ng6-lib';
```

This works because when importing a library by name, Angular CLI looks first in the **tsconfig.json paths** and then in node_modules.

> *ALWAYS: In your test application import using your library by name and NOT the individual files.*

Your **app.module.ts** file should look like this:

```
import { BrowserModule } from '@angular/platform-brows
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { ExampleNg6LibModule } from 'example-ng6-lib';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ExampleNg6LibModule
```

Displaying the example-ng6-lib Component

To keep things simple let's just add the default generated component from our library to our AppComponent template in:
src\app\app.component.html

You can just replace the bottom half of the AppComponent template with:

```
<enl-example-ng6-lib></enl-example-ng6-lib>
```

Your **src\app\app.component.html** should look like this:

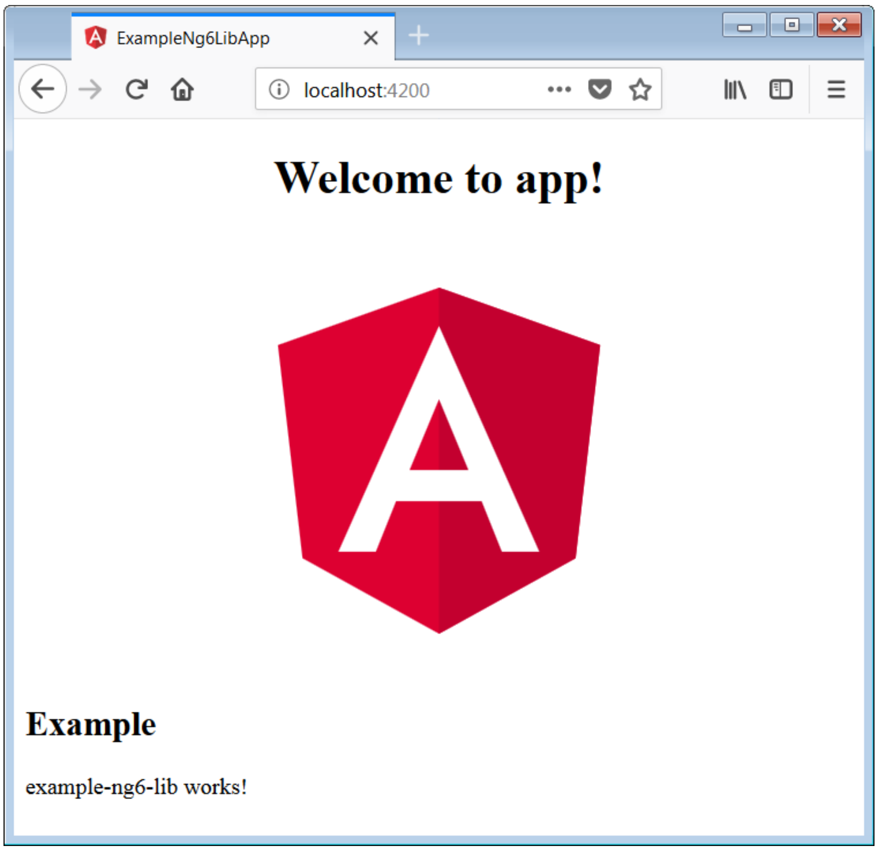
```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="data:image/s
</div>
```

Running Our Application

As always we can run our application using:

```
ng serve
```

And now when we point our browser at:
http://localhost:4200/
we should see the test for our component from our library.



Welcome to app!

Example

example-ng6-lib works!

Expanding Our Library

Now we know how to build our library and run our application using a component from the library. Let's expand our library and see what we need to do to add another component.

Here are the steps we will go through:

1. Generate a new component in our library.
2. Add the component to our library module's **exports**.
3. Add the component to our entry file.
4. Rebuild our library after we make changes to it.
5. Use the new component in our application

Generating a library component

When generating a component for our library we use the `--project` flag to tell Angular CLI that we want it to generate the component in our library project. Let's generate a simple component in our library and call it **foo**:

```
ng generate component foo --project=example-ng6-lib
```

True to form, Angular CLI tells us exactly what it did:

```
CREATE projects/example-ng6-
lib/src/lib/foo/foo.component.html (22 bytes)
CREATE projects/example-ng6-
lib/src/lib/foo/foo.component.spec.ts (607 bytes)
CREATE projects/example-ng6-lib/src/lib/foo/foo.component.ts
(257 bytes)
CREATE projects/example-ng6-
lib/src/lib/foo/foo.component.css (0 bytes)
UPDATE projects/example-ng6-lib/src/lib/example-ng6-
lib.module.ts (347 bytes)
```

Now we have a new component in our library and Angular CLI also added it to the `declarations` array of our library's module in the file: `projects\example-ng6-lib\src\lib\example-ng6-lib.module.ts`

Exporting the component from our library's module

We need to add the **Foo Component** to the exports of our library module. If we don't, we will get a template parse error telling us `"enl-foo" is not a known element` when we try to include the component in our application.

So in the **example-ng6-lib.module.ts** file add `FooComponent` to the `exports` array. Your `ExampleNg6LibModule` should now look like this:

```
import { NgModule } from '@angular/core';
import { ExampleNg6LibComponent } from './example-ng6-
import { FooComponent } from './foo/foo.component';

@NgModule({
  imports: [
  ],
  declarations: [
    ExampleNg6LibComponent,
    FooComponent
  ],
  exports: [
```

Adding the component to the entry file

As we noted before our library project has an **entry file** that defines its public API: `projects\example-ng6-lib\src\public_api.ts`

We need to add the following line to our **entry file** to tell ng-packagr that this component class should be exposed to the users of our library:

```
export * from './lib/foo/foo.component';
```

You're probably thinking this is a bit redundant because we already added our component to the exports in the module. OK, it is true that the `<enl-foo></enl-foo>` element is usable in our application's template even without adding it to our entry file. However, the **FooComponent class** itself won't be exported.

I ran the following test so you don't have to: I added a reference to my FooComponent class like `fooComponent: FooComponent;` in my **app.component.ts** without adding the foo.component file to my entry file. I then re-built the library. When I ran `ng serve`, it did the right thing and failed fast with a `Module has no exported member 'FooComponent'` error.

So the rule is:

> *FOR COMPONENTS:*
> *Using export makes the element visible.*
> *Adding it to the entry file makes the class visible.*

So after adding the line for the new component, your **public_api.ts entry file** should look like this:

```
/*
 * Public API Surface of example-ng6-lib
 */

export * from './lib/example-ng6-lib.service';
export * from './lib/example-ng6-lib.component';
```

Rebuilding our library

After making the changes, we need to rebuild our library with:

```
ng build example-ng6-lib
```

We are doing this manually. However, **Angular CLI** version `6.2` added an incremental build functionality. Every time a file changes **Angular CLI** performs a partial build that emits the amended files. To use the new watch functionality you can use:

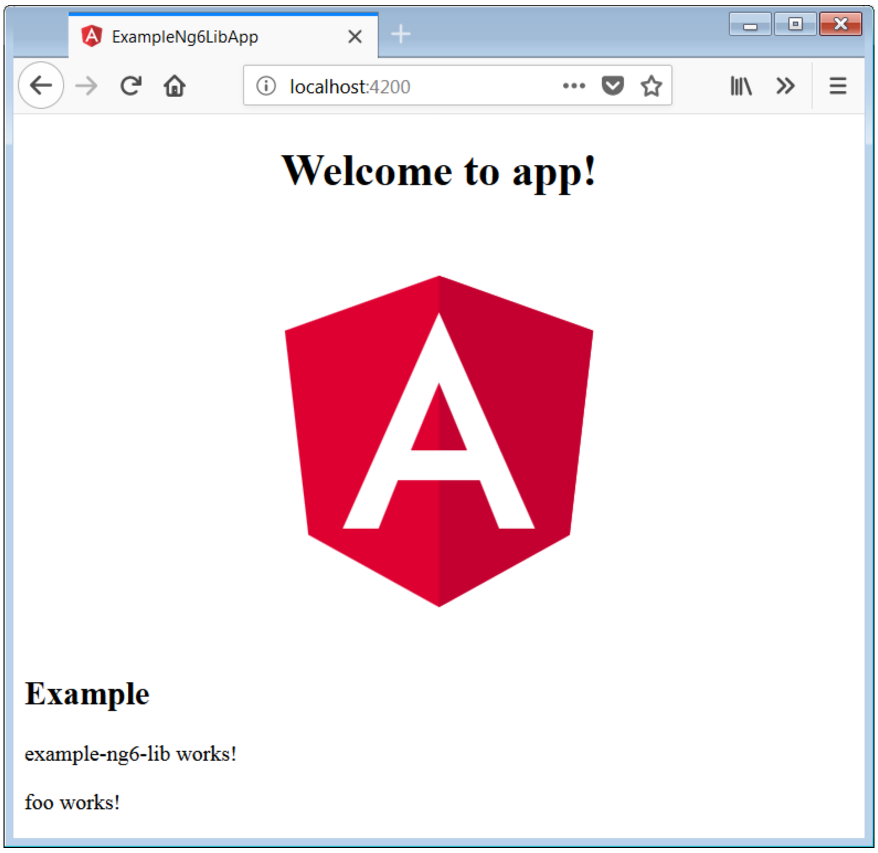
```
ng build example-ng6-lib --watch
```

Using our new library component

Finally, add the element `<enl-foo></enl-foo>` as the last line of your **app.component.html** file. It should look something like this:

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo" src="data:image/s
</div>
<h2>Example</h2>
```

Fire up that `ng serve` and point your browser at: http://localhost:4200/



Welcome to app!

Example

example-ng6-lib works!

foo works!

And there we see our new library component.

Looking Ahead

In Part 2 of this series we discuss building, packaging, and actually using our generated library in another application.

The Angular Library Series - Building and Packaging

Building, packaging, and actually using our Angular library in a separate application.

blog.angularindepth.com

If there is anything I could do to make this article better, please respond or send me a private note. I read and value your feedback.

. . .