

# Combining Multiple Angular Applications into a Single One



Jeffry Houser

[Follow](#)

Dec 5, 2018 · 10 min read



Cover Photo by Ricardo Gomez Angel on Unsplash

I'm Jeffry Houser, a developer in the content engineering group at Disney Streaming Services. My team builds content portals that allow editors to curate the content experiences for consumers in the video services we power.

Each service has its own unique set of challenges and complexities we've had to solve for. Today I'm going to walk you through a prototype we recently created as an experiment to allow multiple systems to feed into a single content portal. The challenge was to build separate UI applications that could be developed and deployed independent of each other, but still be combined into a single unit for deployment. Here's how we did it.

## Setup Your Angular Workspace

You'll need Angular CLI installed. Such a setup is beyond the scope of this article. [Go here and follow instructions.](#)

From there, create the Angular workspace:

ng new

You'll see something like this:

```
C:\Projects\blog\01MultipleProjects>ng new my-app --routing
CREATE my-app/angular.json (3548 bytes)
CREATE my-app/package.json (1310 bytes)
CREATE my-app/README.md (1022 bytes)
CREATE my-app/tsconfig.json (384 bytes)
CREATE my-app/tslint.json (2805 bytes)
CREATE my-app/.editorconfig (245 bytes)
CREATE my-app/.gitignore (583 bytes)
CREATE my-app/src/environments/environment.prod.ts (51 bytes)
CREATE my-app/src/environments/environment.ts (631 bytes)
CREATE my-app/src/favicon.ico (5430 bytes)
CREATE my-app/src/index.html (292 bytes)
CREATE my-app/src/main.ts (370 bytes)
CREATE my-app/src/polyfills.ts (3194 bytes)
CREATE my-app/src/test.ts (642 bytes)
CREATE my-app/src/assets/.gitkeep (0 bytes)
CREATE my-app/src/styles.css (80 bytes)
CREATE my-app/src/browserslist (375 bytes)
CREATE my-app/src/karma.conf.js (964 bytes)
CREATE my-app/src/tsconfig.app.json (194 bytes)
CREATE my-app/src/tsconfig.spec.json (282 bytes)
CREATE my-app/src/tslint.json (314 bytes)
CREATE my-app/src/app/app-routing.module.ts (245 bytes)
CREATE my-app/src/app/app.module.ts (393 bytes)
CREATE my-app/src/app/app.component.html (1173 bytes)
CREATE my-app/src/app/app.component.spec.ts (1106 bytes)
CREATE my-app/src/app/app.component.ts (287 bytes)
CREATE my-app/src/app/app.component.css (0 bytes)
CREATE my-app/e2e/protractor.conf.js (752 bytes)
CREATE my-app/e2e/src/app.e2e-spec.ts (302 bytes)
CREATE my-app/e2e/src/app.po.ts (208 bytes)
CREATE my-app/e2e/tsconfig.e2e.json (213 bytes)
```

This will create the standard Angular application architecture which includes the **node\_modules**, a default component, a routing module, and some other Angular specific files. Let's make sure everything setup correctly. Switch to your my-app directory and run the app:

```
cd my-app
ng serve
```

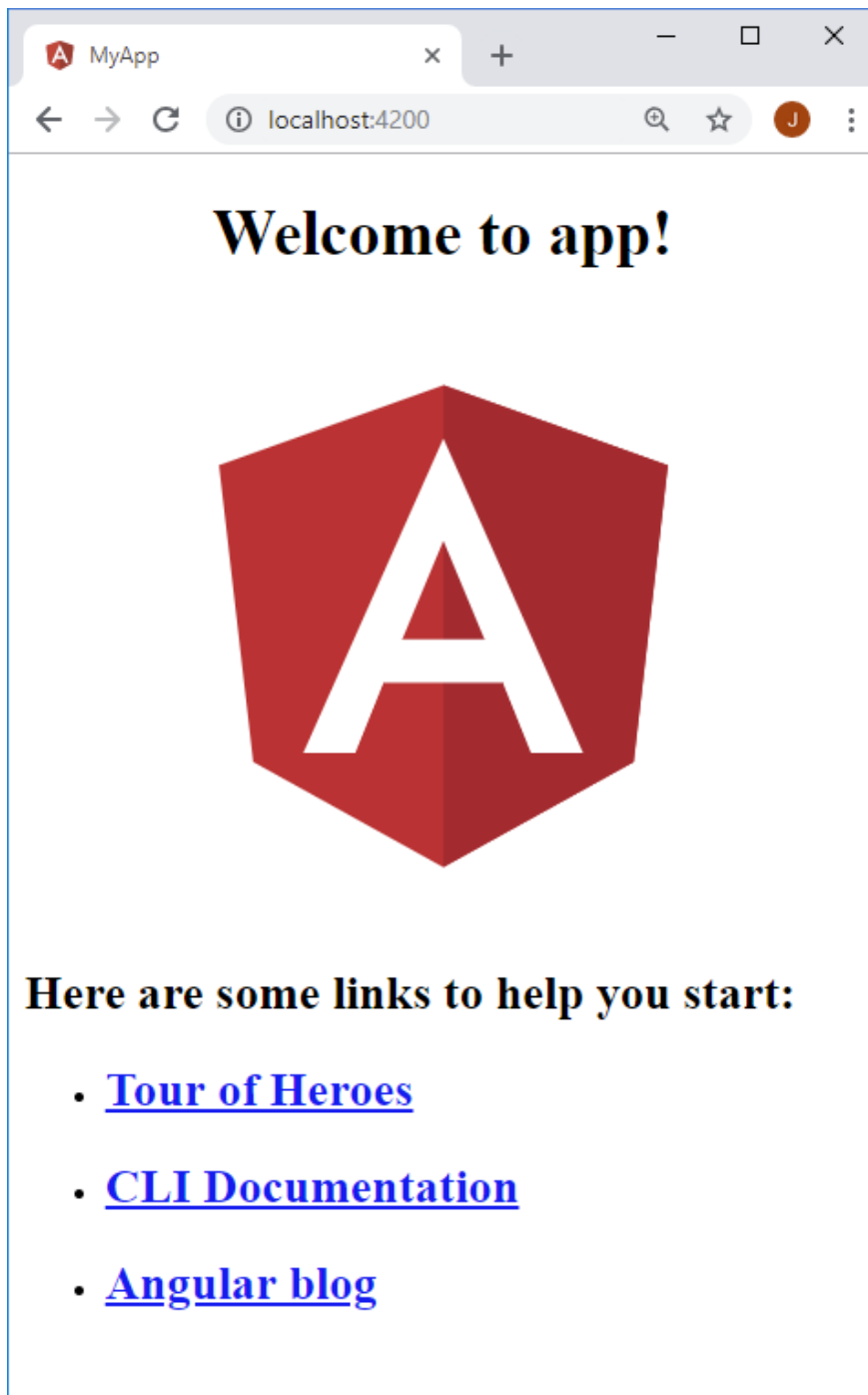
You'll see something like this:

```
C:\ng
C:\Projects\blog\01MultipleProjects>cd my-app
C:\Projects\blog\01MultipleProjects\my-app>ng serve
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

Date: 2018-10-06T00:45:08.806Z
Hash: cb183475ba1586b40fb0
Time: 4149ms
chunk {main} main.js, main.js.map (main) 12.8 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 227 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 5.22 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 15.6 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.53 MB [initial] [rendered]

@wdm: Compiled successfully.
```

Point your favorite browser to localhost:4200 to see the application running:



Angular Logo used per CC BY 4.0

You are prepared to move to the next step.

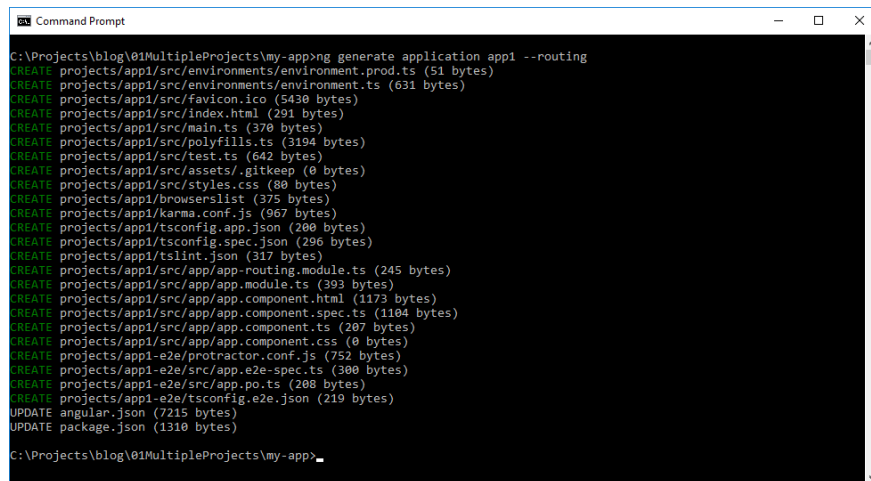
## Create your Sub Applications

When I created the initial prototype, I used the [Nrwl Extensions for Angular](#) because they had a feature to support multiple applications in the same workspace. However, the multiple application functionality is now baked directly into the Angular CLI so this post will focus on that. Instead of using `ng new` to create the workspace, we want to use `ng`

generate application`. This will create a new application, residing right next to the default application:

```
ng generate application app1 --routing
```

You'll see something like this:



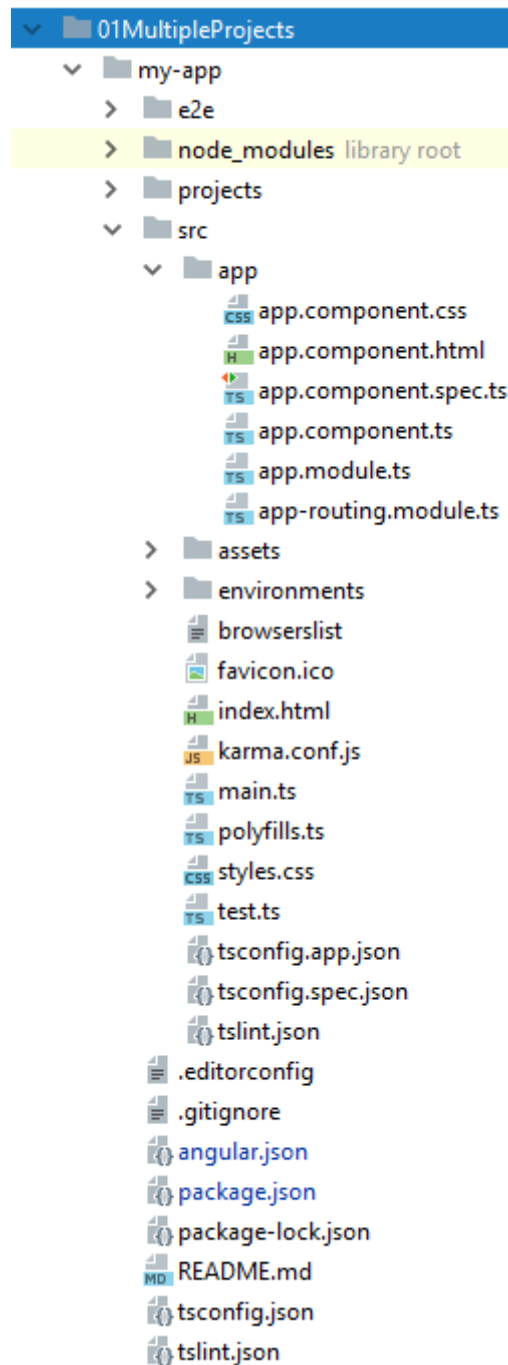
```
Command Prompt
C:\Projects\blog\01MultipleProjects\my-app>ng generate application app1 --routing
CREATE projects/app1/src/environments/environment.prod.ts (51 bytes)
CREATE projects/app1/src/environments/environment.ts (631 bytes)
CREATE projects/app1/src/favicon.ico (5430 bytes)
CREATE projects/app1/src/index.html (291 bytes)
CREATE projects/app1/src/main.ts (370 bytes)
CREATE projects/app1/src/polyfills.ts (3194 bytes)
CREATE projects/app1/src/test.ts (642 bytes)
CREATE projects/app1/src/assets/.gitkeep (0 bytes)
CREATE projects/app1/src/styles.css (80 bytes)
CREATE projects/app1/browserslist (375 bytes)
CREATE projects/app1/karma.conf.js (967 bytes)
CREATE projects/app1/tsconfig.app.json (200 bytes)
CREATE projects/app1/tsconfig.spec.json (296 bytes)
CREATE projects/app1/tslint.json (317 bytes)
CREATE projects/app1/src/app/app-routing.module.ts (245 bytes)
CREATE projects/app1/src/app/app.module.ts (303 bytes)
CREATE projects/app1/src/app/app.component.html (1173 bytes)
CREATE projects/app1/src/app/app.component.spec.ts (1104 bytes)
CREATE projects/app1/src/app/app.component.ts (207 bytes)
CREATE projects/app1/src/app/app.component.css (0 bytes)
CREATE projects/app1-e2e/protractor.conf.js (752 bytes)
CREATE projects/app1-e2e/src/app.e2e-spec.ts (300 bytes)
CREATE projects/app1-e2e/src/app.po.ts (208 bytes)
CREATE projects/app1-e2e/tsconfig.e2e.json (219 bytes)
UPDATE angular.json (7215 bytes)
UPDATE package.json (1310 bytes)
C:\Projects\blog\01MultipleProjects\my-app>
```

While we're at it, let's create a third application:

```
ng generate application app2 --routing
```

You should see something similar to the previous screenshot in your console.

Take a look at the directory listing:

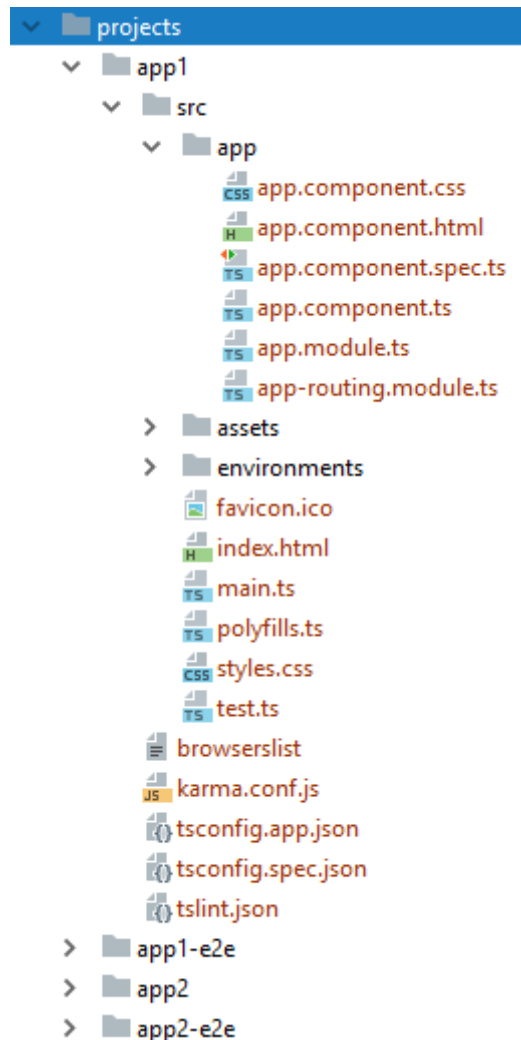


This probably looks like what you expect from an Angular project:

- **e2e:** For End to End Tests
- **node\_modules:** All the packages that had to be installed for the application to run.
- **src:** The main application's source
- **src/app:** The main application code.
- **src/assets:** The assets folder for the main application

- **src/environments:** The environment configurations

If you look closely, you'll notice a new directory, named `projects`. This is where all the sub applications go. Expand it:



The `projects` directory includes:

- **app1:** The source code for your first custom sub application
- **app1/app:** The source code for the app1.
- **app1/assets** The assets for app1.
- **app1/environments:** The environment configs for app1.
- **app1-e2e:** End to End Test for app1.
- **app2:** The source code for your second custom sub application.
- **app2/app:** The source code for the app2.

- **app2/assets** The assets for app2.
- **app2/environments**: The environment configs for app2.

You'll notice that the directory structure inside the projects directory mirrors the directory structure for this workspace's primary project. So, everything you know about building coding your main Angular project is identical when coding your subprojects.

Let's open **projects/app1/src/app/app.component.ts**. Modify the **AppComponent** Class so the title says app 1, like this:

```
export class AppComponent {  
  title = 'app1';  
}
```

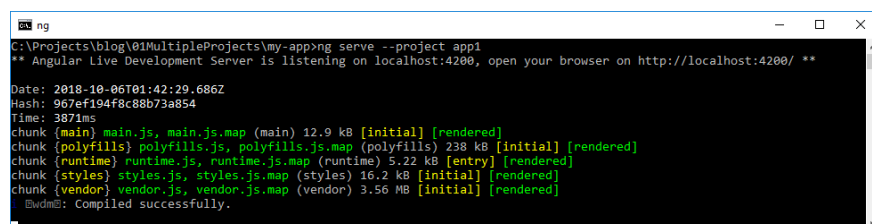
Then open up the same file in your second sub application at **projects/app2/src/app/app.component.ts**:

```
export class AppComponent {  
  title = 'app2';  
}
```

For our testing purpose, these changes will tell you which app you are running. To run one of the sub applications you just use the project argument to ng serve, like this:

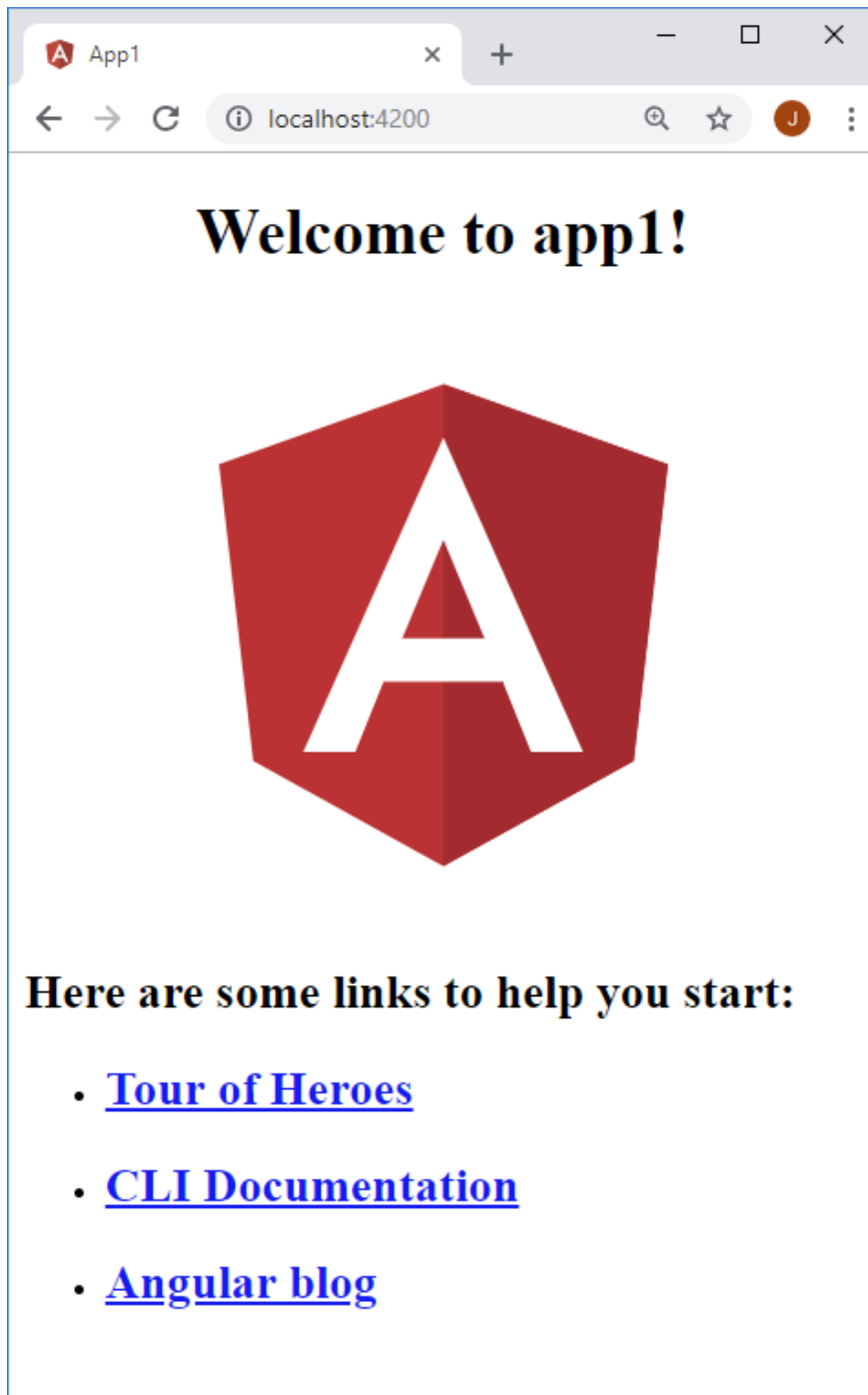
```
Ng serve -project app1
```

You'll see this in the console:



```
ng  
C:\Projects\blog\01MultipleProjects\my-app>ng serve --project app1  
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **  
  
Date: 2018-10-06T01:42:29.686Z  
Hash: 967ef194f8c88b73a854  
Time: 3871ms  
chunk {main} main.js, main.js.map (main) 12.9 kB [initial] [rendered]  
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 238 kB [initial] [rendered]  
chunk {runtime} runtime.js, runtime.js.map (runtime) 5.22 kB [entry] [rendered]  
chunk {styles} styles.js, styles.js.map (styles) 16.2 kB [initial] [rendered]  
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.56 MB [initial] [rendered]  
Browserslist: Compiled successfully.
```

That is probably boring because you've already seen it when running the main app. Load your browser to localhost:4200 and you'll see this:



This is the default Angular application component, but our title specifies that we are running the app1 code. You can run the second sub project like this:

```
ng serve -project=app2
```

You'll see a similar result in the browser. For the main project, just leave the `project` argument off your command line:

```
ng serve
```

Since all the applications use the same port, you can't run them at the same time, but that was fine for our original goals.

## Injecting Sub Apps into Main application

You've created a new workspace, with a main application, and two sub-applications, so how do you combine them? You probably know that you can inject one Angular module inside of another using the `imports` command. In fact, that is the default use of the routing module. Open up the `app.module.ts` from any one of our projects:

```
imports: [  
  BrowserModule,  
  AppRoutingModule  
],
```

Two separate modules are brought in here. That is how we want to inject all the functionality from our sub-applications into the main applications. However, there is a secondary problem. All the `@NgModule` classes, by default, are named `AppModule`. That is just going to confuse us and something we want I wanted to avoid.

Let's look at the `@NgModule` metadata for `app1`:

```
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})  
  
export class AppModule { }
```

In this same file, I'm going to add another class, **App1SharedModule**:

```
@NgModule({})
export class App1SharedModule{
  static forRoot(): ModuleWithProviders {
    return {
      ngModule: AppModule,
      providers: []
    }
  }
}
```

This class contains the **@NgModule** metadata, but with no information. The class uses the **forRoot()** convention which returns a **ModuleWithProviders** instance. The **@NgModule** returned refers back to the original **AppModule** class and metadata. The **providers** value is an empty array.

For the purposes of this sample, we'll keep the **provider** value as an empty array, but in a real application we'll definitely have some providers, so let's create a separate constant to store them:

```
const providers = []
```

As we add providers to the application, add them to this array. The **AppModule's @NgModule** metadata should reference this:

```
providers: providers,
```

As should the **App1SharedModule's forRoot()** method:

```
providers: providers
```

It is the same usage in both places. That is all the setup we need to do for app1. You can repeat this for app2.

Now open **src/app/app.module.ts**. We can just import the two new module classes we just created. In the **@NgModule** metadata:

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  App1SharedModule.forRoot(),  
  App2SharedModule.forRoot()  
],
```

With the submodules added into our main application, we get to use all the providers, components, declarations, and imports from the sub-application in the main application but can still run the sub-applications independently of the main one. Let's look at the import statements:

```
import {App1SharedModule}  
  from "../../projects/app1/src/app/app.module";  
import {App2SharedModule}  
  from "../../projects/app2/src/app/app.module";
```

Nothing special or surprising here, it just traverses the directory tree to find the **app.module**. It feels a bit odd to go out of main app directory to find imports, but is perfectly valid.

## Sharing routes

There is one “nut left to crack”! How do we share routes from the sub-applications with the main applications? Surely this is an important part of embedding one application within another, right? I found a way. First, we're going to want to create some routes in both sub-applications.

First, let's set up each sub-application with some routes and components. First, create a component in app1:

```
ng generate component view1 --project=app1
```

You'll see something like this:

```

C:\Projects\blog\01MultipleProjects\my-app>ng generate component view1 --project=app1
CREATE projects/app1/src/app/view1/view1.component.html (24 bytes)
CREATE projects/app1/src/app/view1/view1.component.spec.ts (621 bytes)
CREATE projects/app1/src/app/view1/view1.component.ts (265 bytes)
CREATE projects/app1/src/app/view1/view1.component.css (0 bytes)
UPDATE projects/app1/src/app/app.module.ts (687 bytes)
C:\Projects\blog\01MultipleProjects\my-app>

```

Let's create another view component, and a nav component, all in app1:

```

ng generate component view2 --project=app1
ng generate component nav --project=app1

```

Open up the **view1.component.html** and add in the nav. It will now look like this:

```

<app-nav></app-nav>
<p>
  app1 view1 works!
</p>

```

Do the same for **view2.component.html**:

```

<app-nav></app-nav>
<p>
  app1 view2 works!
</p>

```

In both cases, I made one mod to the default text in the component. For the purposes of this sample we aren't going to add functionality to the component beyond the default display.

Open up the **nav.component.html** file and add links to navigate between the two routes:

```

<a routerLink="/app1/one" >One</a> |
<a routerLink="/app1/two" >Two</a>

```

Finally, let's add the routes. Open up the app routing module and find the **routes** constant:

```
const routes: Routes = [];
```

By default, no routes are defined. Add these three:

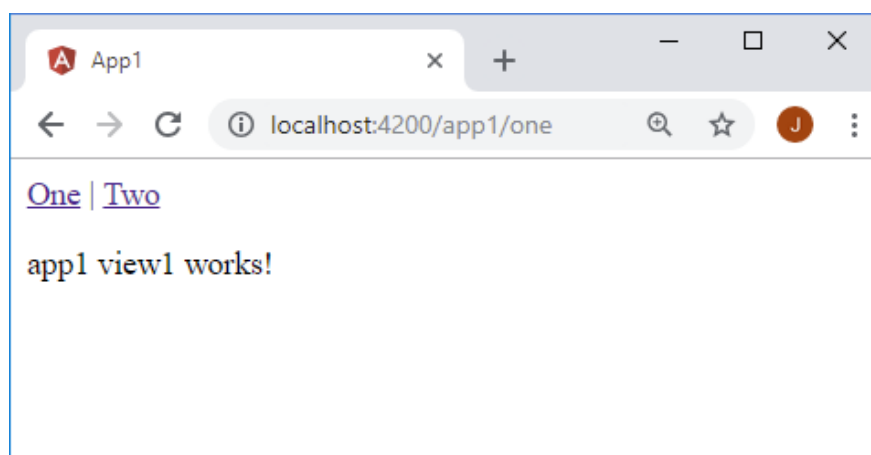
```
const routes: Routes = [  
  { path: 'app1/one', component: View1Component },  
  { path: 'app1/two', component: View2Component },  
  { path: 'app1', redirectTo: 'app1/one' }  
];
```

I set up a route for each component, **app1/one** and **app1/two**. I also set up a route for **app1** that redirected to the first screen. Note here I did not set up a catch all route, like this:

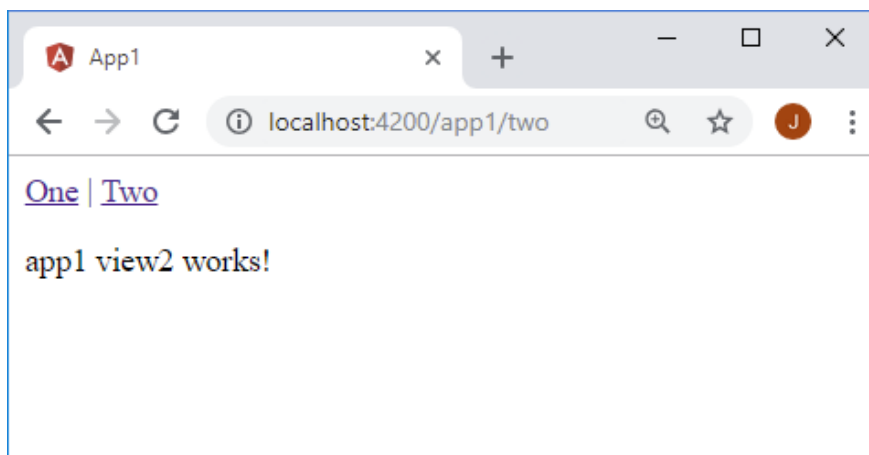
```
{ path: '**', redirectTo: '/app1' }
```

When we integrate these routes into one big main application, having multiple catch-all routes will cause issues.

Run the app and load it in the browser. Load it using `localhost:4200/app1` and you should be able to navigate between the two components:



Click the `two` link:



Now that we have this working. Now use the same steps to setup app2.  
First generate the components:

```
ng generate component view1 --project=app2
ng generate component view2 --project=app2
ng generate component nav --project=app2
```

This is the modifications to the **view1.component.html**:

```
<app-nav></app-nav>
<p>
  app2 view1 works!
</p>
```

The **view2.component.html** looks like this:

```
<app-nav></app-nav>
<p>
  app2 view2 works!
</p>
```

The only changes from app1 are the text change in the main body, from app1 to app2. In a real-world application, I would expect sub-

applications like this to have significant differences in functionality, routes, and screens, but in this example I'm keeping them simple.

Modify the nav component:

```
<a routerLink="/app2/one" >One</a> |  
<a routerLink="/app2/two" >Two</a>
```

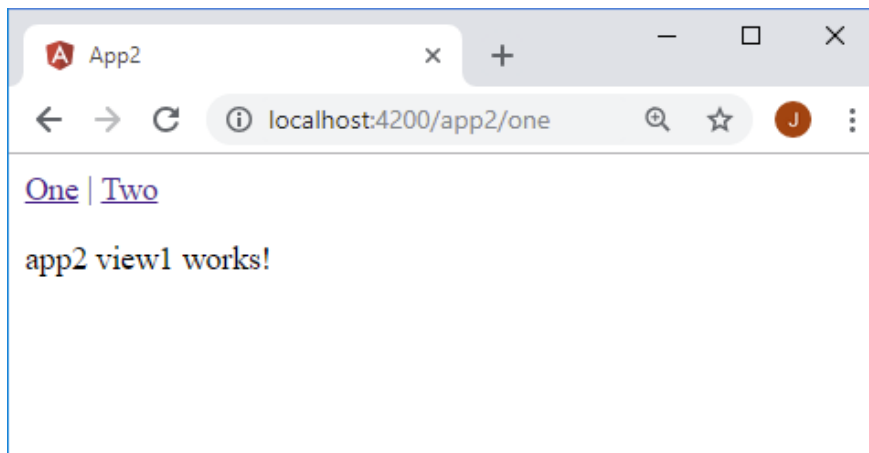
The app2 will put all pages under the app2 route. Open up the app2s routing module:

```
const routes: Routes = [  
  { path: 'app2/one', component: View1Component },  
  { path: 'app2/two', component: View2Component },  
  { path: 'app2', redirectTo: 'app2/one' },  
];
```

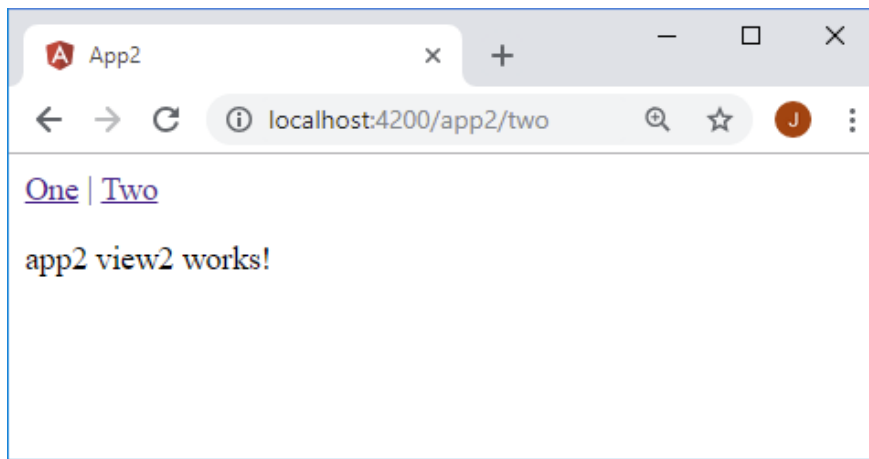
Run the app:

```
ng serve -project=app2
```

Load it at localhost:4200/app2. You'll see the first view load:



Click the 'two' link:



We now have both our sub modules set up, it is time to set up the main application to use these applications and routes

Open the app routing module and add the routes:

```
const routes: Routes = [
  {path: 'app1',
    loadChildren:
    '../..../projects/app1/src/app/app.module#App1SharedModule'},
  {path: 'app2',
    loadChildren:
    '../..../projects/app2/src/app/app.module#App2SharedModule'},
  { path: '**', redirectTo: '/app1/one' }
];
```

Instead of pointing a specific route at a specific component, we use the **loadChildren** property to load routes from the sub modules on demand. This is used as part of Angular's mechanism for lazy loading modules. The value of **loadChildren** points back at the main application that loads the modules. I set the route for the lazy loaded modules to the name of the module, **app1** or **app2**. This mirrors the base route used in the sub module. The generic redirect route directs users to the **app1/one** as a default.

Inside the **AppRoutingModule**, you'll need to add imports for both the referenced modules:

```
imports: [
  RouterModule.forRoot(routes),
  App1SharedModule.forRoot(),
```

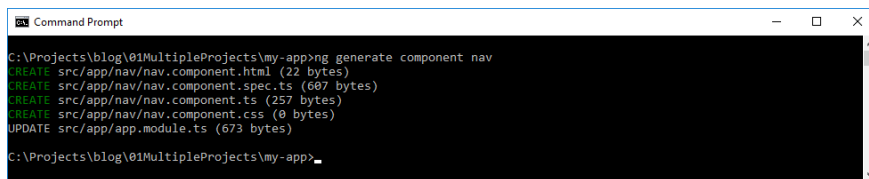
```
App2SharedModule.forRoot()
],
```

The **RouterModule.forRoot()** was already there, but the **App1SharedModule.forRoot()** and **App2SharedModule.forRoot()** are new. This Angular router module needs to know about these other routes in order to support lazy loading.

One final thing is to add a global navigation, let's follow the same pattern that was used in the sub modules:

```
ng generate component nav
```

You'll see this:



```

C:\Projects\blog\01MultipleProjects\my-app>ng generate component nav
CREATE src/app/nav/nav.component.html (22 bytes)
CREATE src/app/nav/nav.component.spec.ts (607 bytes)
CREATE src/app/nav/nav.component.ts (257 bytes)
CREATE src/app/nav/nav.component.css (0 bytes)
UPDATE src/app/app.module.ts (673 bytes)
C:\Projects\blog\01MultipleProjects\my-app>

```

Add the nav to your **app.component.html**:

```
<app-nav></app-nav>
<br/><br>
<router-outlet></router-outlet>
```

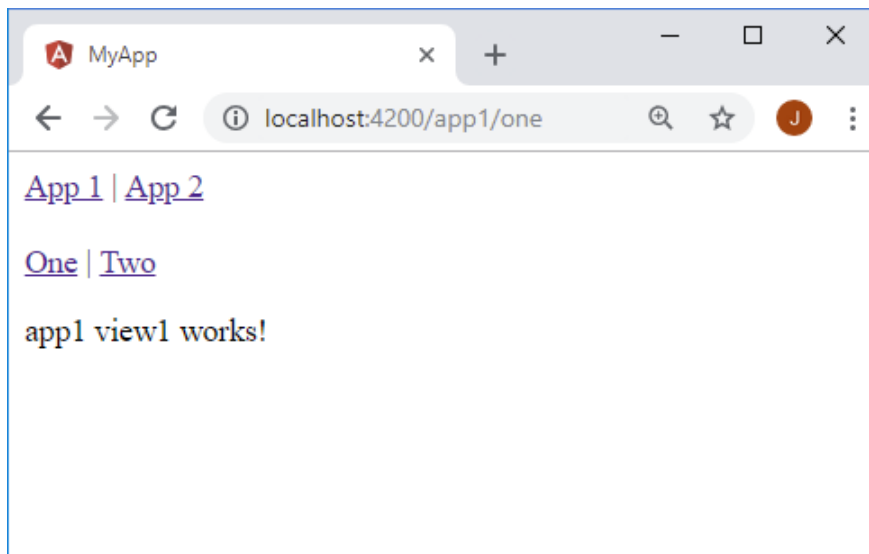
And add some links between the two sub-applications in the nav:

```
<a routerLink="/app1" >App 1</a> |
<a routerLink="/app2" >App 2</a>
```

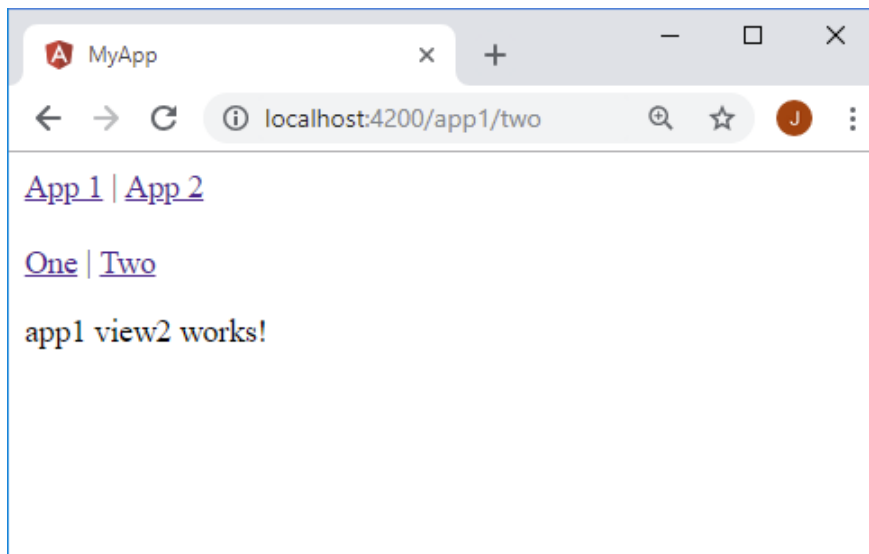
Now you should be good to go. Run:

```
ng serve
```

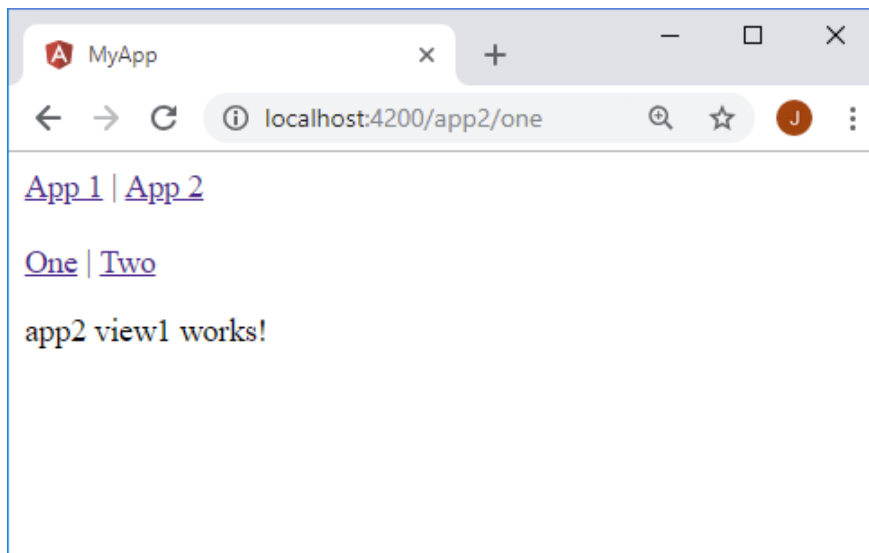
Load your browser to localhost:4200:



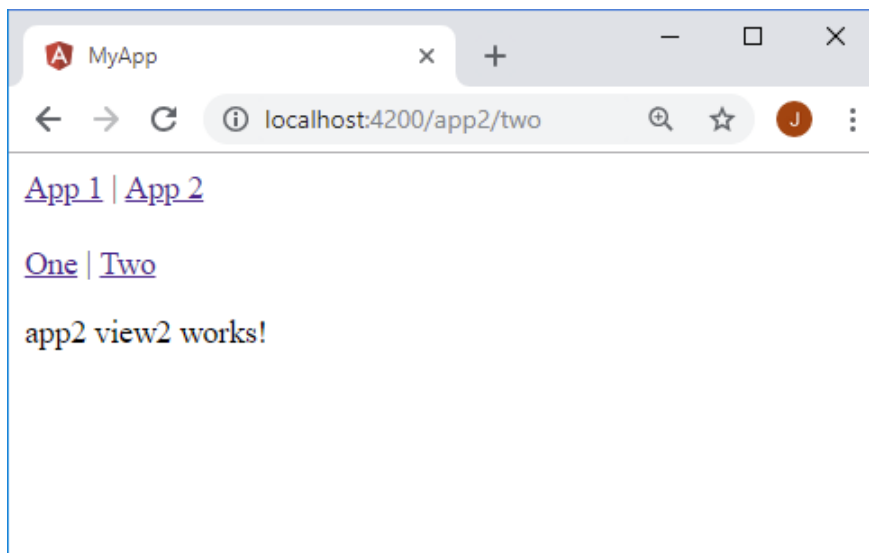
Click the links and you'll see that the routes properly change:



You can even switch to the other application:



And switch between the views there:



## Final Thoughts

I had a lot of fun building this prototype and can see advantages of composing an app with this method. It helps to separate logic and reduce cognitive load when developing the app. Being able to focus is important for the productivity of any programmer. Although we decided not to use this approach, it provided some useful leanings along the way.



